NORTHEASTERN UNIVERSITY

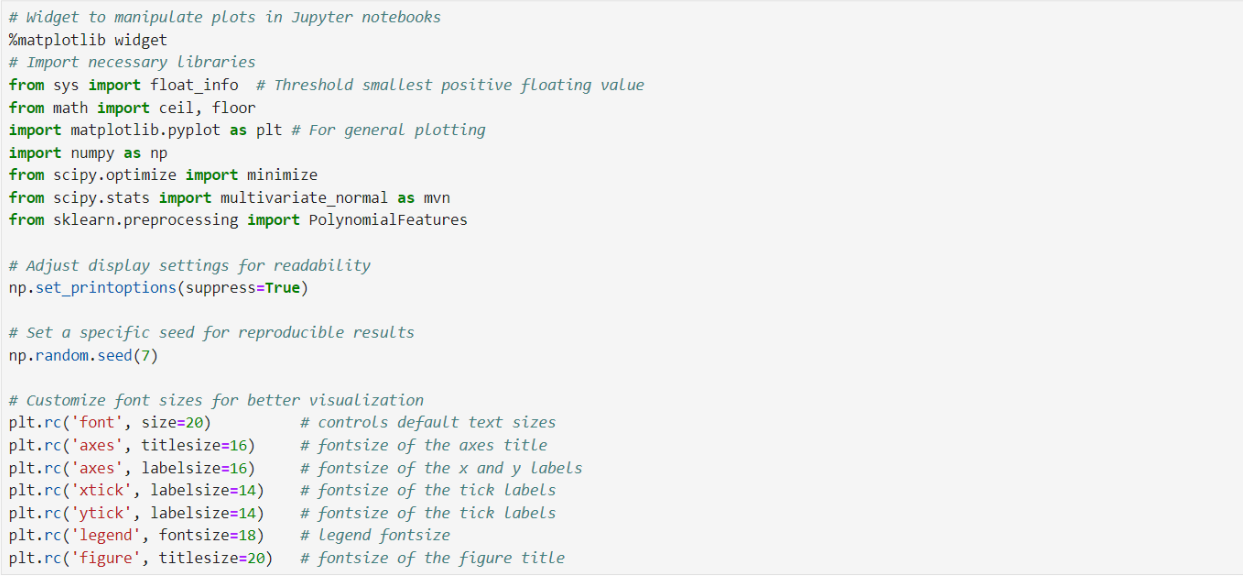
Department of Electrical and Computer Engineering

EECE 5644 Machine Learning and Pattern Recognition

Homework Assignment – 2

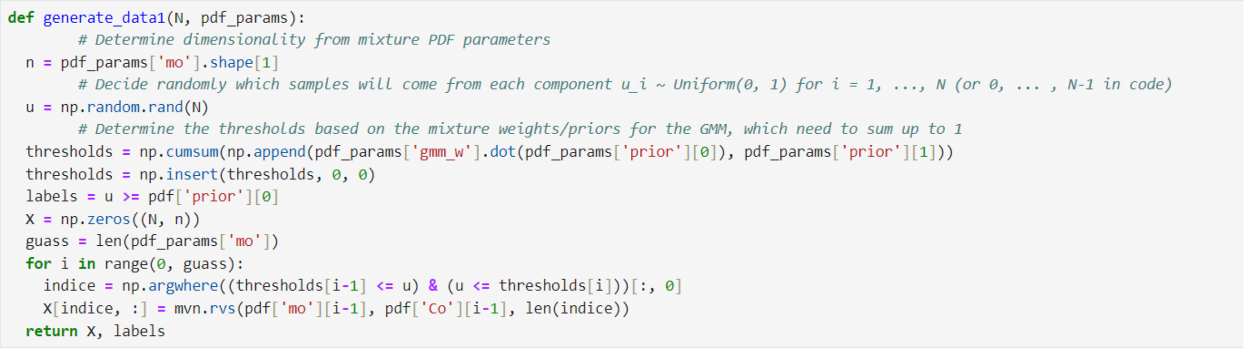
Mansi Rao Mudrakola
NUID: 002702946

Imports:



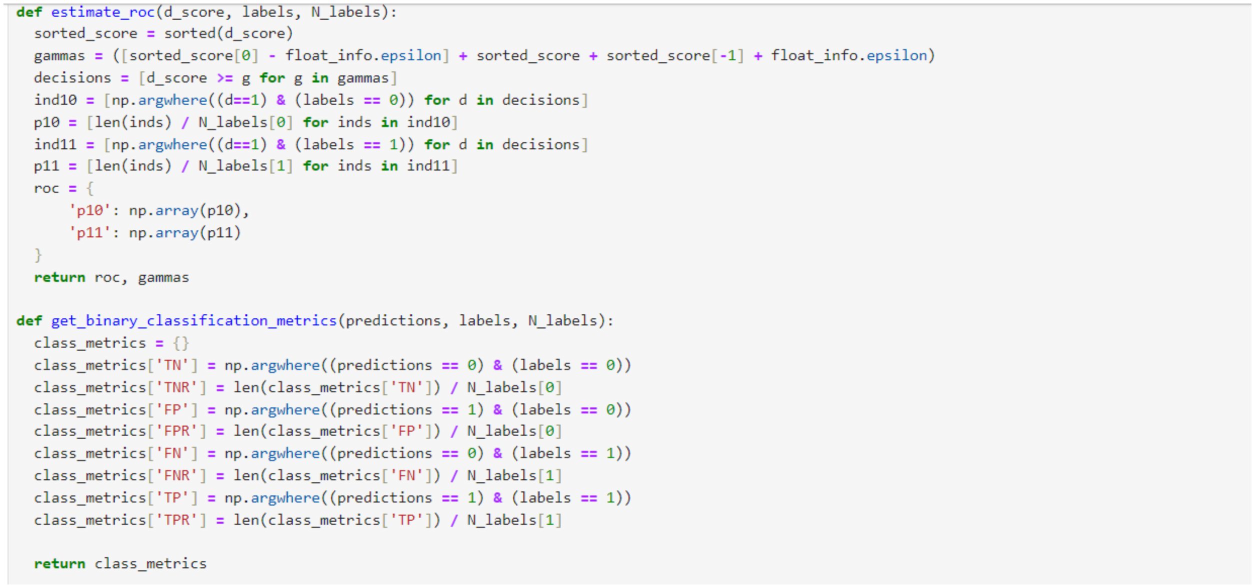
```
# Widget to manipulate plots in Jupyter notebooks
%matplotlib widget
# Import necessary libraries
from sys import float_info  # Threshold smallest positive floating value
from math import ceil, floor
import matplotlib.pyplot as plt # For general plotting
import numpy as np
from scipy.optimize import minimize
from scipy.stats import multivariate_normal as mvn
from sklearn.preprocessing import PolynomialFeatures

# Adjust display settings for readability
np.set_printoptions(suppress=True)

# Set a specific seed for reproducible results
np.random.seed(7)

# Customize font sizes for better visualization
plt.rc('font', size=20)          # controls default text sizes
plt.rc('axes', titlesize=16)     # fontsize of the axes title
plt.rc('axes', labelsize=16)     # fontsize of the x and y labels
plt.rc('xtick', labelsize=14)    # fontsize of the tick labels
plt.rc('ytick', labelsize=14)    # fontsize of the tick labels
plt.rc('legend', fontsize=18)    # legend fontsize
plt.rc('figure', titlesize=20)   # fontsize of the figure title
```

Data Generation:



```
def generate_data1(N, pdf_params):
        # Determine dimensionality from mixture PDF parameters
  n = pdf_params['mo'].shape[1]
        # Decide randomly which samples will come from each component u_i ~ Uniform(0, 1) for i = 1, ..., N (or 0, ... , N-1 in code)
  u = np.random.rand(N)
        # Determine the thresholds based on the mixture weights/priors for the GMM, which need to sum up to 1
  thresholds = np.cumsum(np.append(pdf_params['gmm_w'].dot(pdf_params['prior'][0]), pdf_params['prior'][1]))
  thresholds = np.insert(thresholds, 0, 0)
  labels = u >= pdf['prior'][0]
  X = np.zeros((N, n))
  guass = len(pdf_params['mo'])
  for i in range(0, guass):
    indice = np.argwhere((thresholds[i-1] <= u) & (u <= thresholds[i]))[:, 0]
    X[indice, :] = mvn.rvs(pdf['mo'][i-1], pdf['Co'][i-1], len(indice))
  return X, labels
```

Evaluation Functions:



```
def estimate_roc(d_score, labels, N_labels):
  sorted_score = sorted(d_score)
  gammas = ([sorted_score[0] - float_info.epsilon] + sorted_score + sorted_score[-1] + float_info.epsilon)
  decisions = [d_score >= g for g in gammas]
  ind10 = [np.argwhere((d==1) & (labels == 0)) for d in decisions]
  p10 = [len(inds) / N_labels[0] for inds in ind10]
  ind11 = [np.argwhere((d==1) & (labels == 1)) for d in decisions]
  p11 = [len(inds) / N_labels[1] for inds in ind11]
  roc = {
      'p10': np.array(p10),
      'p11': np.array(p11)
  }
  return roc, gammas

def get_binary_classification_metrics(predictions, labels, N_labels):
  class_metrics = {}
  class_metrics['TN'] = np.argwhere((predictions == 0) & (labels == 0))
  class_metrics['TNR'] = len(class_metrics['TN']) / N_labels[0]
  class_metrics['FP'] = np.argwhere((predictions == 1) & (labels == 0))
  class_metrics['FPR'] = len(class_metrics['FP']) / N_labels[0]
  class_metrics['FN'] = np.argwhere((predictions == 0) & (labels == 1))
  class_metrics['FNR'] = len(class_metrics['FN']) / N_labels[1]
  class_metrics['TP'] = np.argwhere((predictions == 1) & (labels == 1))
  class_metrics['TPR'] = len(class_metrics['TP']) / N_labels[1]

  return class_metrics
```

Question 1

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

The class priors are P(L = 0) = 0.6 and P(L = 1) = 0.4. The class class-conditional pdfs are p(x|L = 0) = w01g(x|m01, C01) + w02g(x|m02, C02) and p(x|L = 1) = w11g(x|m11, C11) + w12g(x|m12, C12), where g(x|m, C) is a multivariate Gaussian probability density function with mean vector m and covariance matrix C. The parameters of the class-conditional Gaussian pdfs are: wi1 = wi2 = 1/2 for i ∈ {1, 2}, and

$$\mathbf{m}_{01} = \left[\begin{smallmatrix}-1\\-1\end{smallmatrix}\right] \quad \mathbf{m}_{02} = \left[\begin{smallmatrix}1\\1\end{smallmatrix}\right] \quad \mathbf{m}_{11} = \left[\begin{smallmatrix}-1\\1\end{smallmatrix}\right] \quad \mathbf{m}_{12} = \left[\begin{smallmatrix}1\\-1\end{smallmatrix}\right] \quad \mathbf{C}_{ij} = \left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right] \text{ for all } \{ij\} \text{ pairs.}$$

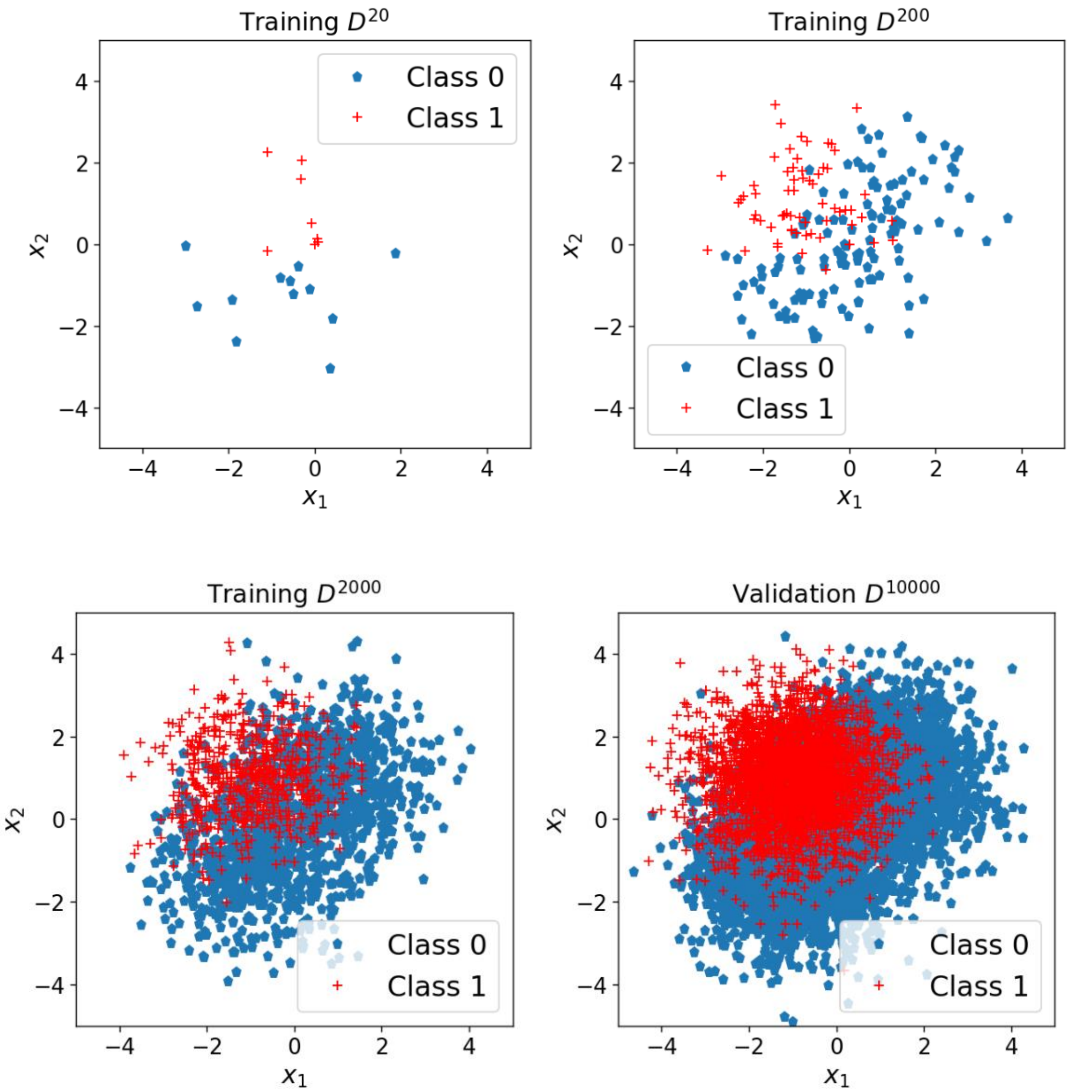
For numerical results requested below, generate the following independent datasets each con- sisting of iid samples from the specified data distribution, and in each dataset make sure to include the true class label for each sample.

- D20 train consists of 20 samples and their labels for training;
- D200 train consists of 200 samples and their labels for training;
- D2000 train consists of 2000 samples and their labels for training;
- D10K validate consists of 10000 samples and their labels for validation

Ans: The code prepares, visualizes, and compares training and validation datasets of different sizes, which is useful in assessing how the size of the dataset can impact the performance of machine learning models or algorithms. It provides a visual representation of the data and is a valuable step in data analysis and model evaluation.

```
# Define distribution parameters: a, mu and Sigma
# Define distribution parameters: a, mu and Sigma
pdf = {
    'prior': np.array([0.6, 0.4]),
    'gmm_w': np.array([0.5, 0.5, 0.5, 0.5]),
    'mo': np.array([[-1, -1], [1, 1], [-1, 1], [1, -1]]),
    'Co': np.array([[[1,0], [0, 1]],
                    [[1,0], [0, 1]],
                    [[1,0], [0, 1]],
                    [[1,0], [0, 1]]])
}

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

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

# Making lists for the samples and labels
X_train = []
labels_train = []
N_labels_train = []
# Index for axes
t = 0
for N_t in N_train:
    X_t, labels_t = generate_data1(N_t, pdf)
    X_train.append(X_t)

    labels_train.append(labels_t)
    N_labels_train.append(np.array((sum(labels_t == 0), sum(labels_t == 1))))
```

```
    # Axis fancy indexing for the sake of plotting correctly in subplots
    ax[floor(t/2), t%2].set_title(r"Training $D^{%d}$" % (N_t))
    ax[floor(t/2), t%2].plot(X_t[labels_t==0, 0], X_t[labels_t==0, 1], 'p', label="Class 0")
    ax[floor(t/2), t%2].plot(X_t[labels_t==1, 0], X_t[labels_t==1, 1], 'r+', label="Class 1")
    ax[floor(t/2), t%2].set_xlabel(r"$x_1$")
    ax[floor(t/2), t%2].set_ylabel(r"$x_2$")
    ax[floor(t/2), t%2].legend()

    t += 1

# Number of validation samples for experiments
N_valid = 10000

X_valid, labels_valid = generate_data1(N_valid, pdf)

# Count up the number of samples per class
Nl_valid = np.array((sum(labels_valid == 0), sum(labels_valid == 1)))

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

# Using validation set samples to limit axes (most samples drawn, highest odds of spanning sample space)
x1_valid_lim = (floor(np.min(X_valid[:,0])), ceil(np.max(X_valid[:,0])))
x2_valid_lim = (floor(np.min(X_valid[:,1])), ceil(np.max(X_valid[:,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()
```



Part 1: 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 D10K validate. From the decision results and true labels for this validation set, estimate and plot the ROC curve for a corresponding discriminant score for this classifier, and on the ROC curve indicate, with a special marker, the location of the min-P(error) classifier. Also report an estimate of the min-P(error) achievable, based on counts of decision- truth label pairs on D10K validate.

Ans: Binary Classification is performed using ERM (Empirical Risk Minimization) algorithm and the resulting decision boundary is plotted. Below, the discriminant_score_erm function calculates the ERM score for a given input X and a probability density function(pdf) that models the two classes. The function returns the difference between the logarithm of the likelihood of the two classes.

ERM classifier with 0-1 loss achieves minimum probability of error. The decision rule for two class ERM classifier is as below:

$$\frac{p(x|L=1)}{p(x|L=0)} \underset{\text{Decide } 0}{\overset{\text{Decide } 1}{\gtrless}} \frac{(\lambda_{10}-\lambda_{00})}{(\lambda_{01}-\lambda_{11})} \frac{p(L=0)}{p(L=1)}$$

with 0-1 loss, the decision rule reduces down to minimum probality of error (MAP) classificatio

$$\frac{p(x|L=1)}{p(x|L=0)} \underset{\text{Decide}_0}{\overset{\text{Decide}_1}{\gtrless}} \frac{p(L=0)}{p(L=1)}$$

Hence, theoretically optimal threshold γ^* that leads to minimum probability of error is given by:

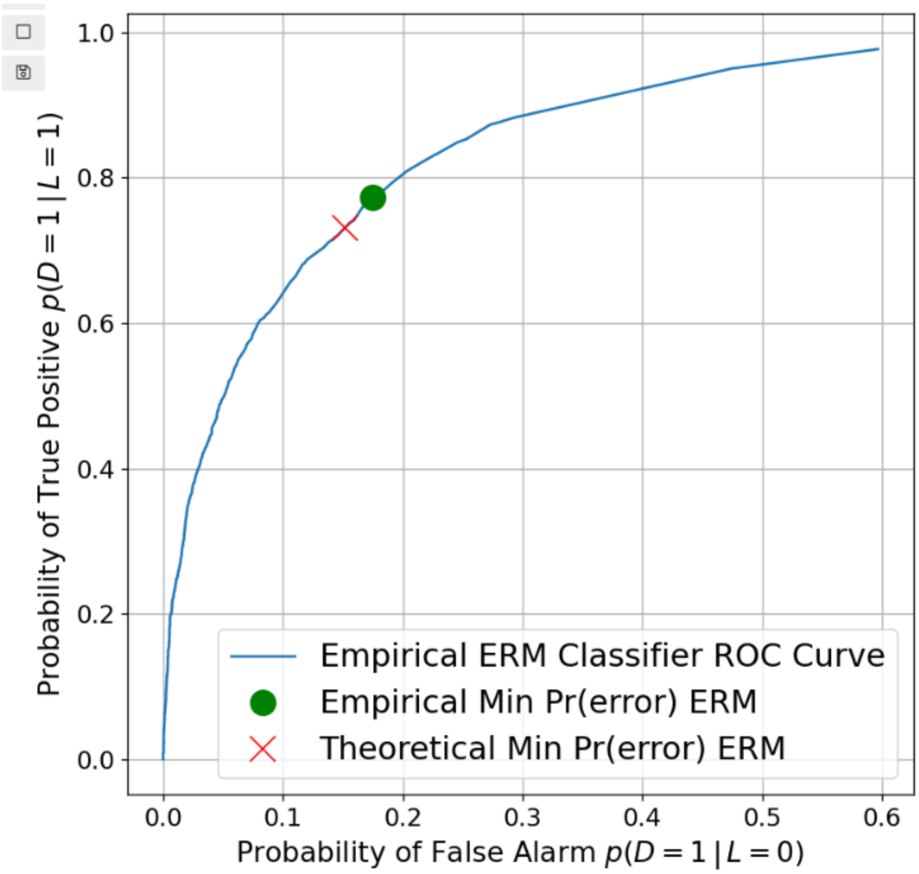
$$\gamma^* = \frac{p(L=0)}{p(L=1)} = \frac{0.6}{0.4} = 1.5$$

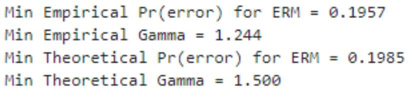
Equivalent decision rule can be obtained by taking log on both sides,

$$\ln p(x|L=1) - \ln p(x|L=0) \underset{\text{Decide}_0}{\overset{\text{Decide}_1}{\gtrless}} \ln \gamma^*$$

```python
def discriminant_score_erm(X, dist_params):
    class_lld_0 = (dist_params['gmm_w'][0]*mvn.pdf(X, dist_params['mo'][0], dist_params['Co'][0])
                   + dist_params['gmm_w'][1]*mvn.pdf(X, dist_params['mo'][1], dist_params['Co'][1]))
    class_lld_1 = mvn.pdf(X, dist_params['mo'][2], dist_params['Co'][2])
    erm_scores = np.log(class_lld_1) - np.log(class_lld_0)
    return erm_scores

disc_erm_scores = discriminant_score_erm(X_valid, pdf)
roc_erm, gammas_empirical = estimate_roc(disc_erm_scores, labels_valid, Nl_valid)

fig_roc, ax_roc = plt.subplots(figsize=(8, 8));

ax_roc.plot(roc_erm['p10'], roc_erm['p11'], 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 $p(D=1\,|\,L=1)$")

# ROC returns FPR vs TPR, but prob error needs FNR so take 1-TPR
# Pr(error; γ) = p(D = 1|L = 0; γ)p(L = 0) + p(D = 0|L = 1; γ)p(L = 1)
prob_error_empirical = np.array((roc_erm['p10'], 1 - roc_erm['p11'])).T.dot(Nl_valid / N_valid)

# Min prob error for the empirically-selected gamma thresholds
min_prob_error_empirical = np.min(prob_error_empirical)
min_ind_empirical = np.argmin(prob_error_empirical)

# Compute theoretical gamma as log-ratio of priors (0-1 loss) -> MAP classification rule
gamma_map = pdf['prior'][0] / pdf['prior'][1]
decisions_map = disc_erm_scores >= np.log(gamma_map)

class_metrics_map = get_binary_classification_metrics(decisions_map, labels_valid, Nl_valid)
# To compute probability of error, we need FPR and FNR
min_prob_error_map = np.array((class_metrics_map['FPR'] * pdf['prior'][0] +
                               class_metrics_map['FNR'] * pdf['prior'][1]))
```

```python
# Plot theoretical and empirical
ax_roc.plot(roc_erm['p10'][min_ind_empirical], roc_erm['p11'][min_ind_empirical], 'go', label="Empirical Min Pr(error) ERM",
            markersize=14)
ax_roc.plot(class_metrics_map['FPR'], class_metrics_map['TPR'], 'rx', label="Theoretical Min Pr(error) ERM", markersize=14)

plt.grid(True)
plt.legend()
plt.show()

print("Min Empirical Pr(error) for ERM = {:.4f}".format(min_prob_error_empirical))
print("Min Empirical Gamma = {:.3f}".format(np.exp(gammas_empirical[min_ind_empirical])))

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





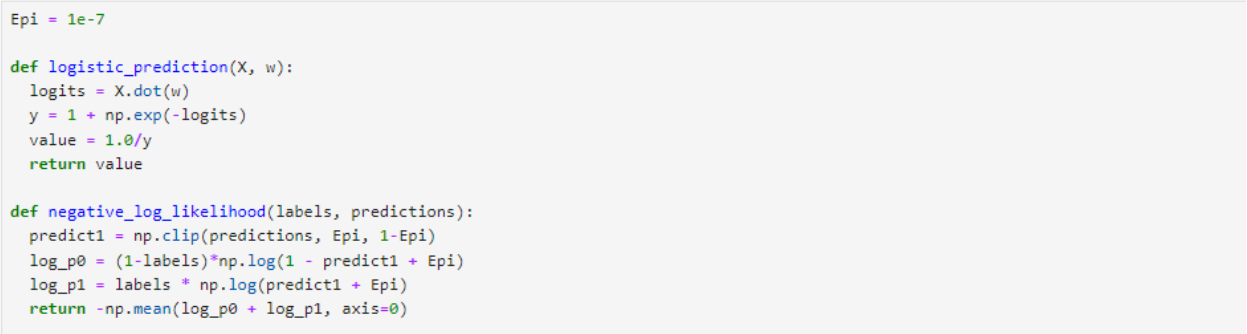
```
Min Empirical Pr(error) for ERM = 0.1957
Min Empirical Gamma = 1.244
Min Theoretical Pr(error) for ERM = 0.1985
Min Theoretical Gamma = 1.500
```

Part 2: (a) Using the maximum likelihood parameter estimation technique train three separate logistic-linear-function-based approximations of class label posterior functions given a sample. For each approximation use one of the three training datasets D20 train, D200 train, D2000 train. When optimizing the parameters, specify the optimization problem as minimization of the negative-log- likelihood of the training dataset, and use your favorite numerical optimization approach, such as gradient descent or Matlab's fminsearch. Determine how to use these class-label-posterior approx- imations to classify a sample in order to approximate the minimum-P(error) classification rule; apply these three approximations of the class label posterior function on samples in D10K validate, and estimate the probability of error that these three classification rules will attain (using counts of decisions on the validation set).

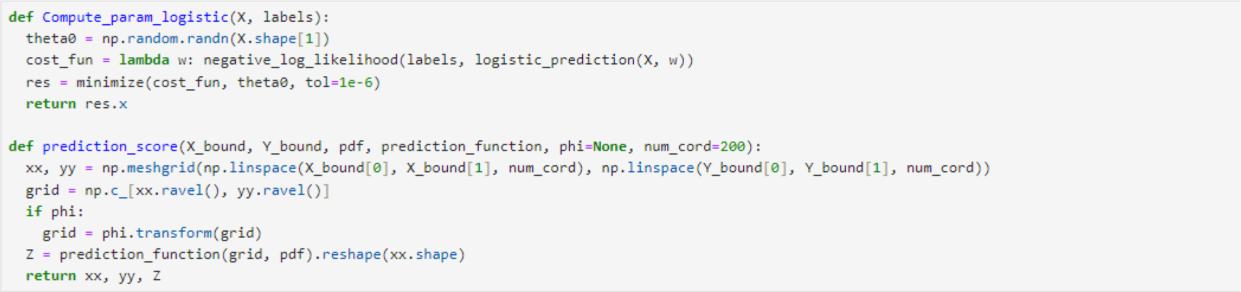
Ans: The code defines the value of Epi as 1e-7, which is used to avoid taking the logarithm of zero in the negative_log_likelihood function. The logistic_prediction function takes in the input data X and the weights w, and returns the predicted probabilities of the target variable using the logistic function. The negative_log_likelihood function takes in the true labels and predicted probabilities and calculates the negative log-likelihood loss.

The Compute_param_logistic function takes in the input data and true labels, initializes the weights randomly, and uses the minimize function to find the optimal weights that minimize the negative log-likelihood loss. The prediction_score_grid function takes in the x and y boundaries of the plot, a trained logistic regression model, and a data transformation function phi. It generates a grid of coordinates, applies the data transformation function phi if provided, and returns the predicted probabilities for the grid points. The logistic_classifier function takes in the plot axis, the input data X, the trained weights w, the true labels, the number of labels N labels, and the data transformation function phi. It generates the predicted labels, calculates the binary classification metrics, plots the data points, and adds the contour plot of the predicted probabilities.

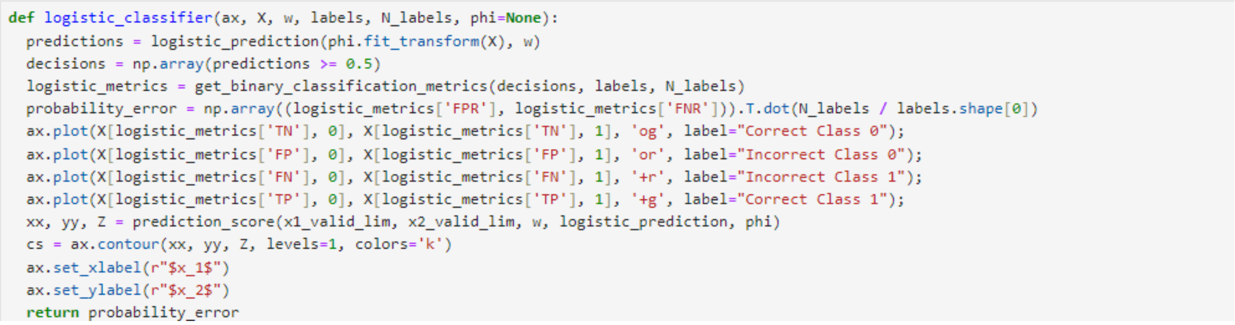
For each of the three training sets, the code trains a logistic regression model using the Compute_param_logistic function and plots the data points and the decision boundary using the logistic_classifier function.



```
Epi = 1e-7

def logistic_prediction(X, w):
  logits = X.dot(w)
  y = 1 + np.exp(-logits)
  value = 1.0/y
  return value

def negative_log_likelihood(labels, predictions):
  predict1 = np.clip(predictions, Epi, 1-Epi)
  log_p0 = (1-labels)*np.log(1 - predict1 + Epi)
  log_p1 = labels * np.log(predict1 + Epi)
  return -np.mean(log_p0 + log_p1, axis=0)
```



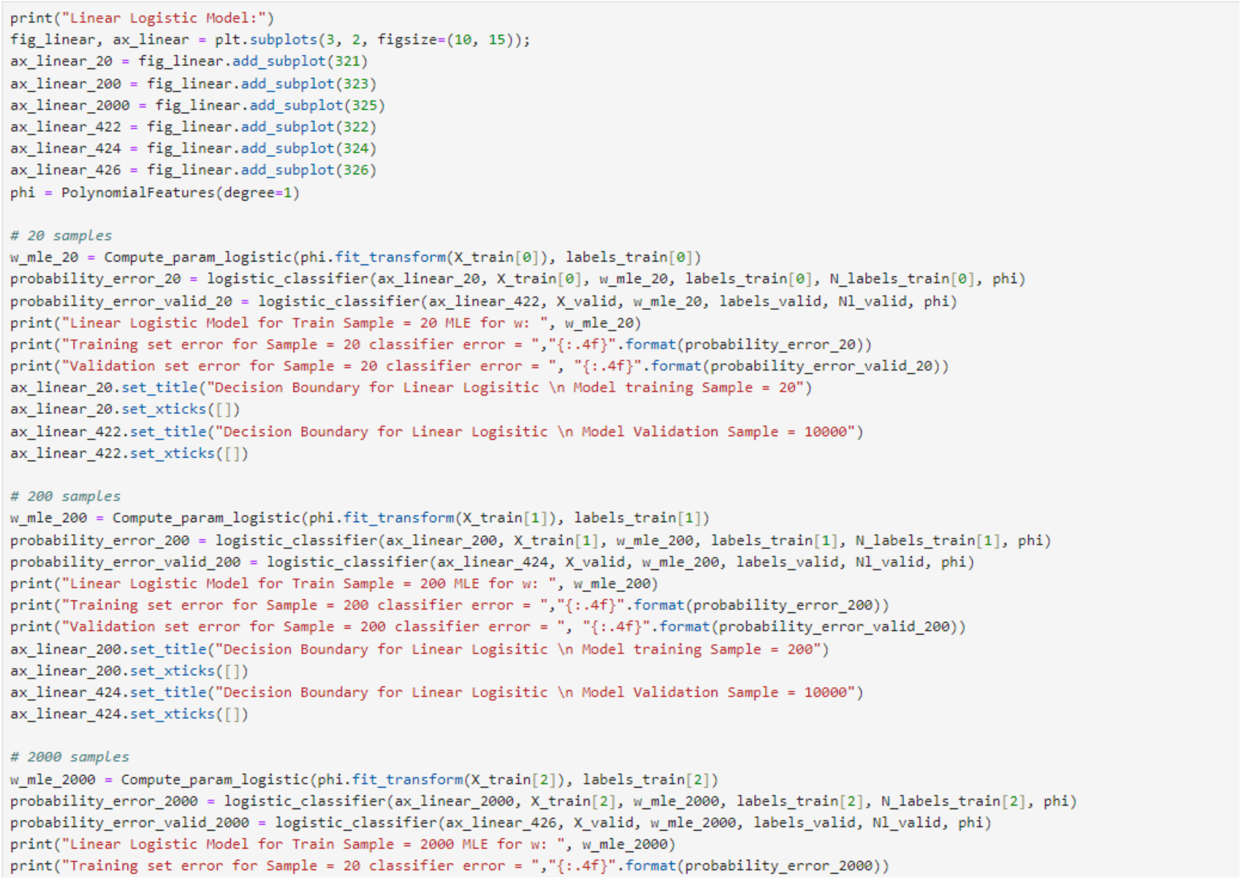
```
def Compute_param_logistic(X, labels):
  theta0 = np.random.randn(X.shape[1])
  cost_fun = lambda w: negative_log_likelihood(labels, logistic_prediction(X, w))
  res = minimize(cost_fun, theta0, tol=1e-6)
  return res.x

def prediction_score(X_bound, Y_bound, pdf, prediction_function, phi=None, num_cord=200):
  xx, yy = np.meshgrid(np.linspace(X_bound[0], X_bound[1], num_cord), np.linspace(Y_bound[0], Y_bound[1], num_cord))
  grid = np.c_[xx.ravel(), yy.ravel()]
  if phi:
    grid = phi.transform(grid)
  Z = prediction_function(grid, pdf).reshape(xx.shape)
  return xx, yy, Z
```



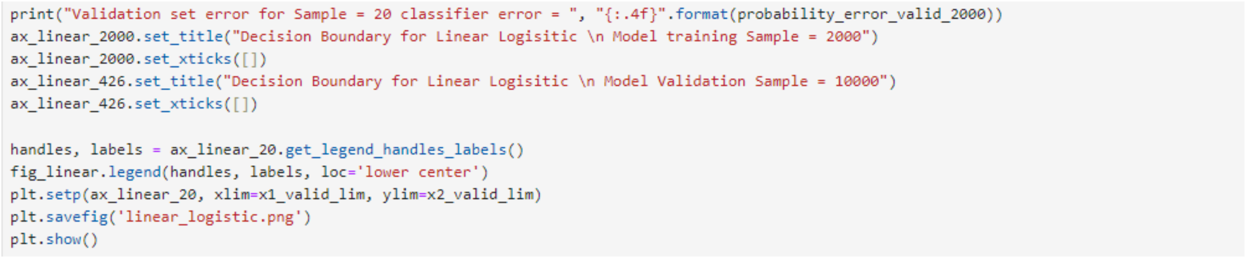
```
def logistic_classifier(ax, X, w, labels, N_labels, phi=None):
  predictions = logistic_prediction(phi.fit_transform(X), w)
  decisions = np.array(predictions >= 0.5)
  logistic_metrics = get_binary_classification_metrics(decisions, labels, N_labels)
  probability_error = np.array((logistic_metrics['FPR'], logistic_metrics['FNR'])).T.dot(N_labels / labels.shape[0])
  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");
  xx, yy, Z = prediction_score(x1_valid_lim, x2_valid_lim, w, logistic_prediction, phi)
  cs = ax.contour(xx, yy, Z, levels=1, colors='k')
  ax.set_xlabel(r"$x_1$")
  ax.set_ylabel(r"$x_2$")
  return probability_error
```

Logistic-Linear Model

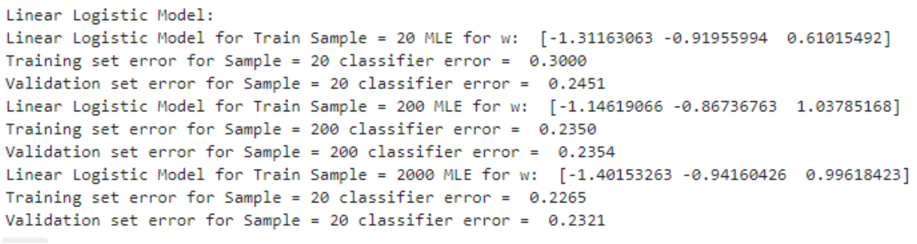
The Logistic-Linear model helps to minimize the negative log likelihood loss on the 3 different Dtrain subsets. Below, the training procedure for this model can be seen:



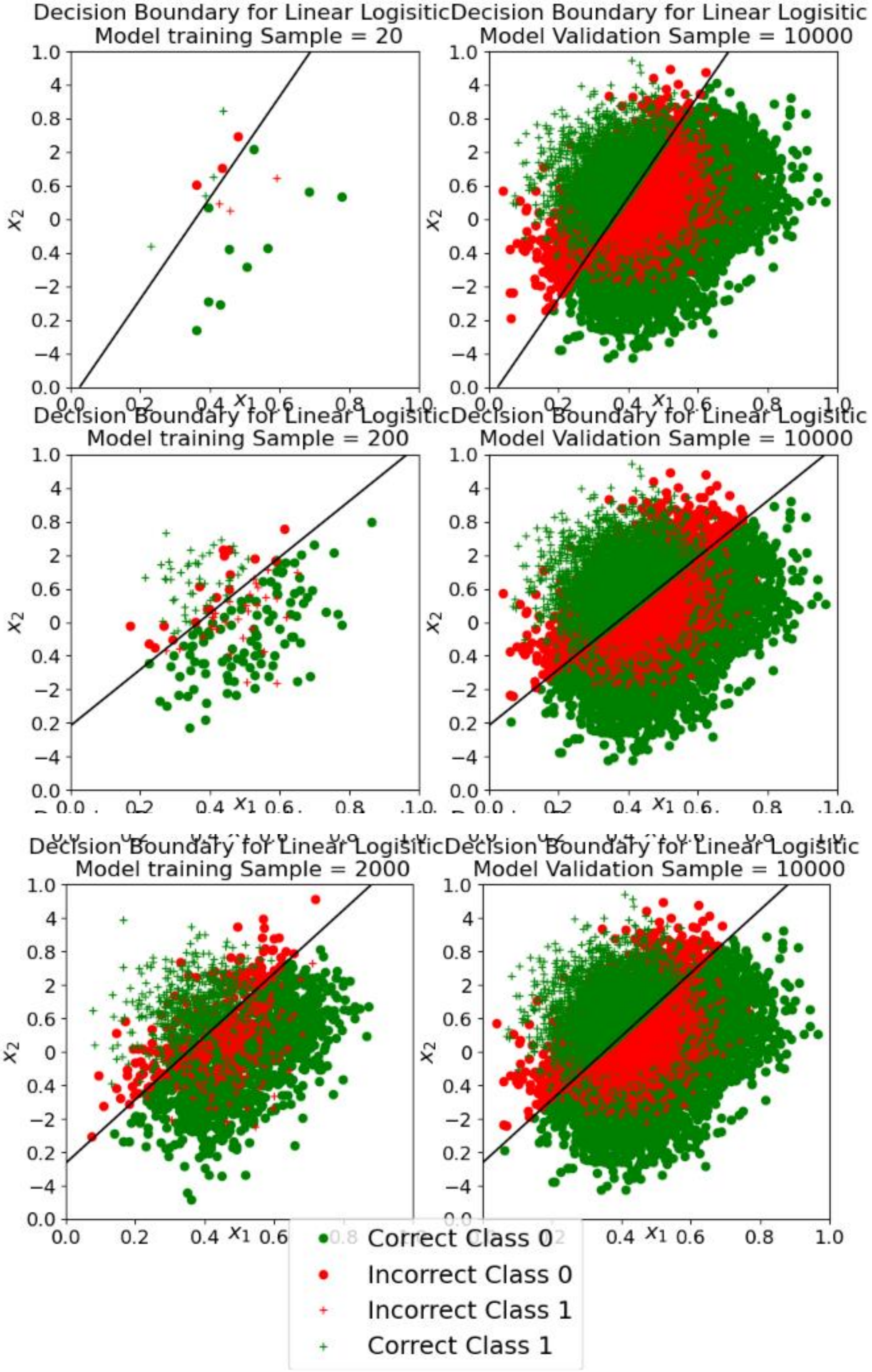
```python
print("Linear Logistic Model:")
fig_linear, ax_linear = plt.subplots(3, 2, figsize=(10, 15));
ax_linear_20 = fig_linear.add_subplot(321)
ax_linear_200 = fig_linear.add_subplot(323)
ax_linear_2000 = fig_linear.add_subplot(325)
ax_linear_422 = fig_linear.add_subplot(322)
ax_linear_424 = fig_linear.add_subplot(324)
ax_linear_426 = fig_linear.add_subplot(326)
phi = PolynomialFeatures(degree=1)

# 20 samples
w_mle_20 = Compute_param_logistic(phi.fit_transform(X_train[0]), labels_train[0])
probability_error_20 = logistic_classifier(ax_linear_20, X_train[0], w_mle_20, labels_train[0], N_labels_train[0], phi)
probability_error_valid_20 = logistic_classifier(ax_linear_422, X_valid, w_mle_20, labels_valid, Nl_valid, phi)
print("Linear Logistic Model for Train Sample = 20 MLE for w: ", w_mle_20)
print("Training set error for Sample = 20 classifier error = ","{:.4f}".format(probability_error_20))
print("Validation set error for Sample = 20 classifier error = ", "{:.4f}".format(probability_error_valid_20))
ax_linear_20.set_title("Decision Boundary for Linear Logisitic \n Model training Sample = 20")
ax_linear_20.set_xticks([])
ax_linear_422.set_title("Decision Boundary for Linear Logisitic \n Model Validation Sample = 10000")
ax_linear_422.set_xticks([])

# 200 samples
w_mle_200 = Compute_param_logistic(phi.fit_transform(X_train[1]), labels_train[1])
probability_error_200 = logistic_classifier(ax_linear_200, X_train[1], w_mle_200, labels_train[1], N_labels_train[1], phi)
probability_error_valid_200 = logistic_classifier(ax_linear_424, X_valid, w_mle_200, labels_valid, Nl_valid, phi)
print("Linear Logistic Model for Train Sample = 200 MLE for w: ", w_mle_200)
print("Training set error for Sample = 200 classifier error = ","{:.4f}".format(probability_error_200))
print("Validation set error for Sample = 200 classifier error = ", "{:.4f}".format(probability_error_valid_200))
ax_linear_200.set_title("Decision Boundary for Linear Logisitic \n Model training Sample = 200")
ax_linear_200.set_xticks([])
ax_linear_424.set_title("Decision Boundary for Linear Logisitic \n Model Validation Sample = 10000")
ax_linear_424.set_xticks([])

# 2000 samples
w_mle_2000 = Compute_param_logistic(phi.fit_transform(X_train[2]), labels_train[2])
probability_error_2000 = logistic_classifier(ax_linear_2000, X_train[2], w_mle_2000, labels_train[2], N_labels_train[2], phi)
probability_error_valid_2000 = logistic_classifier(ax_linear_426, X_valid, w_mle_2000, labels_valid, Nl_valid, phi)
print("Linear Logistic Model for Train Sample = 2000 MLE for w: ", w_mle_2000)
print("Training set error for Sample = 20 classifier error = ","{:.4f}".format(probability_error_2000))
```



```python
print("Validation set error for Sample = 20 classifier error = ", "{:.4f}".format(probability_error_valid_2000))
ax_linear_2000.set_title("Decision Boundary for Linear Logisitic \n Model training Sample = 2000")
ax_linear_2000.set_xticks([])
ax_linear_426.set_title("Decision Boundary for Linear Logisitic \n Model Validation Sample = 10000")
ax_linear_426.set_xticks([])

handles, labels = ax_linear_20.get_legend_handles_labels()
fig_linear.legend(handles, labels, loc='lower center')
plt.setp(ax_linear_20, xlim=x1_valid_lim, ylim=x2_valid_lim)
plt.savefig('linear_logistic.png')
plt.show()
```



```
Linear Logistic Model:
Linear Logistic Model for Train Sample = 20 MLE for w:  [-1.31163063 -0.91955994  0.61015492]
Training set error for Sample = 20 classifier error =  0.3000
Validation set error for Sample = 20 classifier error =  0.2451
Linear Logistic Model for Train Sample = 200 MLE for w:  [-1.14619066 -0.86736763  1.03785168]
Training set error for Sample = 200 classifier error =  0.2350
Validation set error for Sample = 200 classifier error =  0.2354
Linear Logistic Model for Train Sample = 2000 MLE for w:  [-1.40153263 -0.94160426  0.99618423]
Training set error for Sample = 20 classifier error =  0.2265
Validation set error for Sample = 20 classifier error =  0.2321
```



Decision Boundary for Linear Logisitic Model training Sample = 20

Decision Boundary for Linear Logisitic Model Validation Sample = 10000

Decision Boundary for Linear Logisitic Model training Sample = 200

Decision Boundary for Linear Logisitic Model Validation Sample = 10000

Decision Boundary for Linear Logisitic Model training Sample = 2000

Decision Boundary for Linear Logisitic Model Validation Sample = 10000

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

Ans: We predict the logistic-linear regression model to perform worse and theoretically optimal classifiers since the decision border between the two overlapping Gaussians defining this dataset is nonlinear. A dictionary called pdf, which seems to contain parameters for a probability distribution, is defined at the start of the code. 'Prior', 'gmm_w','mo' (mean vectors), and 'Co' (covariance matrices) are among the parameters. Usually, these parameters are used to a Gaussian Mixture Model (GMM) or other probabilistic model that is comparable.

The Maximum Likelihood Estimates (MLE) for the weight vector w for each model and sample size are also provided. For the Linear Logistic Model, we see that as the training sample size increases, the error rates on both the training and validation sets decrease. However, even for the largest sample size (2000), the error rates are still relatively high (training error of 0.2265 and validation error of 0.2321). This suggests that a linear decision boundary is not very effective at separating the two classes in the given dataset.

Using the theoretical classifier's performance on the validation set as a guideline, with Pr(error)=0.1957, then the logistic-linear classifier's error estimates of ∼0.4 across all trained models is significantly worse (by more than a factor of two). The classifier is also ineffectual at differentiating between these Gaussian class-conditional PDFs, regardless of the volume of data trained on (quite identical performance on the validation set for the D200 and D2000 models). This is seen by the linear decision boundaries.

Question 2

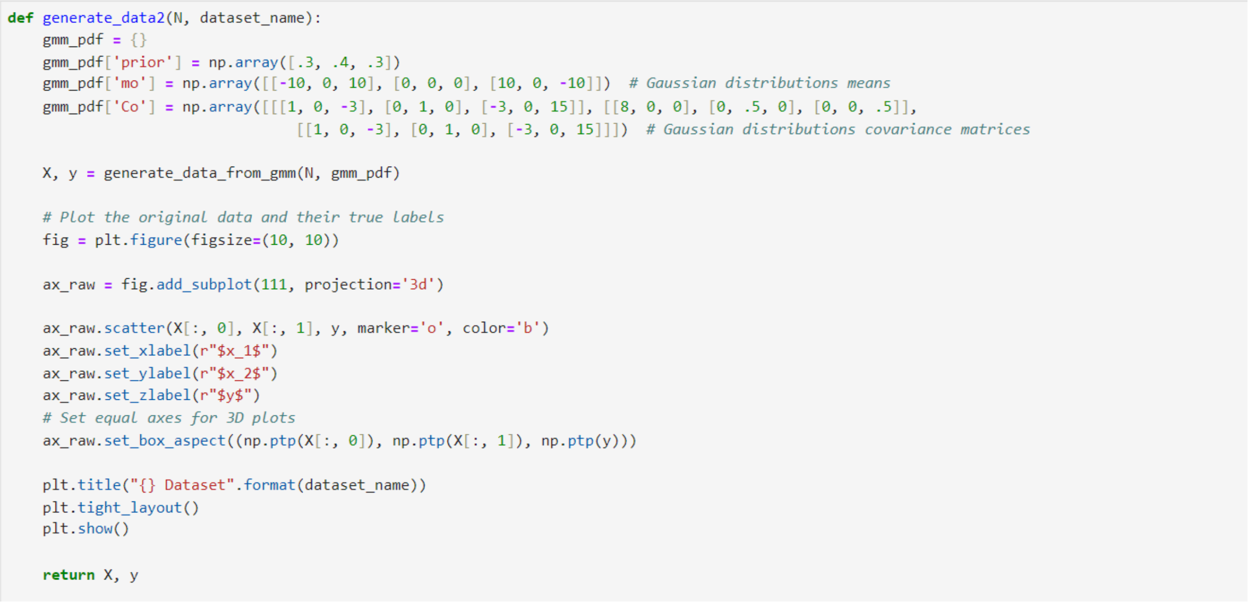
Assume that scalar-real y and two-dimensional real vector x are related to each other according to y = c(x, w) + v, where c(., w) is a cubic polynomial in x with coefficients w and v is a random Gaussian random scalar with mean zero and σ 2-variance.

Given a dataset D = (x1, y1), . . . , (xN , yN) with N samples of (x, y) pairs, with the assumption that these samples are independent and identically distributed according to the model, derive two estimators for w using maximum-likelihood (ML) and maximum-a-posteriori (MAP) parameter estimation approaches as a function of these data samples. For the MAP estimator, assume that w has a zero-mean Gaussian prior with covariance matrix γI.

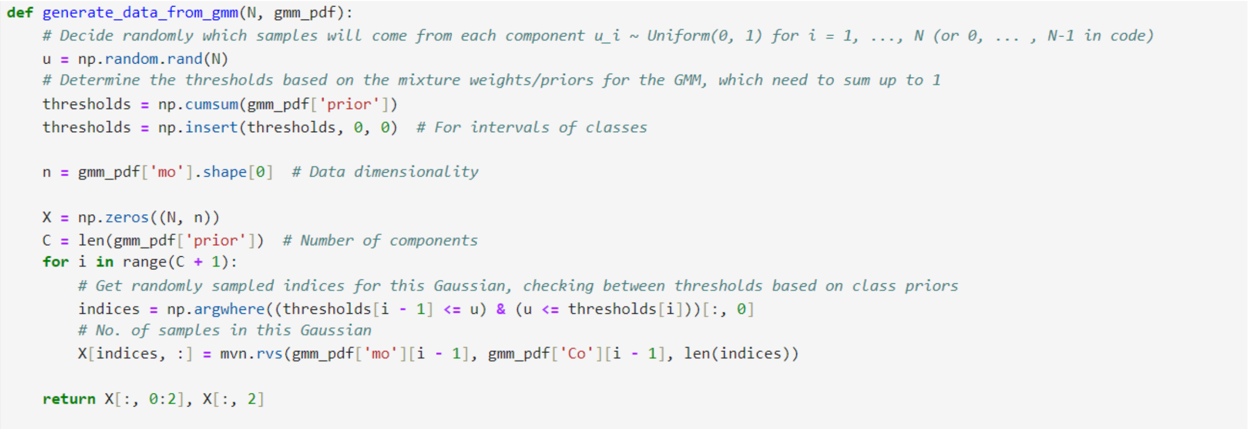
Ans: generateDataFromGMM(N, gmm pdf) is a function that generates a dataset of size N from a Gaussian Mixture Model (GMM) specified by the dictionary gmm pdf. The GMM is defined by the prior probabilities of the classes, the mean vectors of the Gaussians, and their covariance matrices. The function returns a tuple of two arrays: the first array contains the 2D coordinates of the data points, and the second array contains the class label.

Generate_data(N) is a function that generates a dataset of size N with three classes. The dataset is created by calling generateDataFromGMM() with a pre-defined GMM. The function returns the same tuple as generateDataFromGMM().

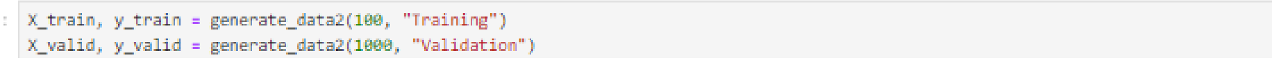
prediction_score(X bound, Y bound, pdf, prediction function, phi=None, num cord=100) is a function that evaluates the performance of a prediction function on a grid of points. The grid is defined by the x and y coordinate bounds X bound and Y bound, and the number of points in each direction num cord. The prediction function takes two arguments: a 2D array of points and a dictionary pdf that defines the GMM for the problem. If phi is not None, the points are transformed using a polynomial feature expansion. The function returns three arrays: the x and y coordinates of the points in the grid, and the predicted class labels for each point.



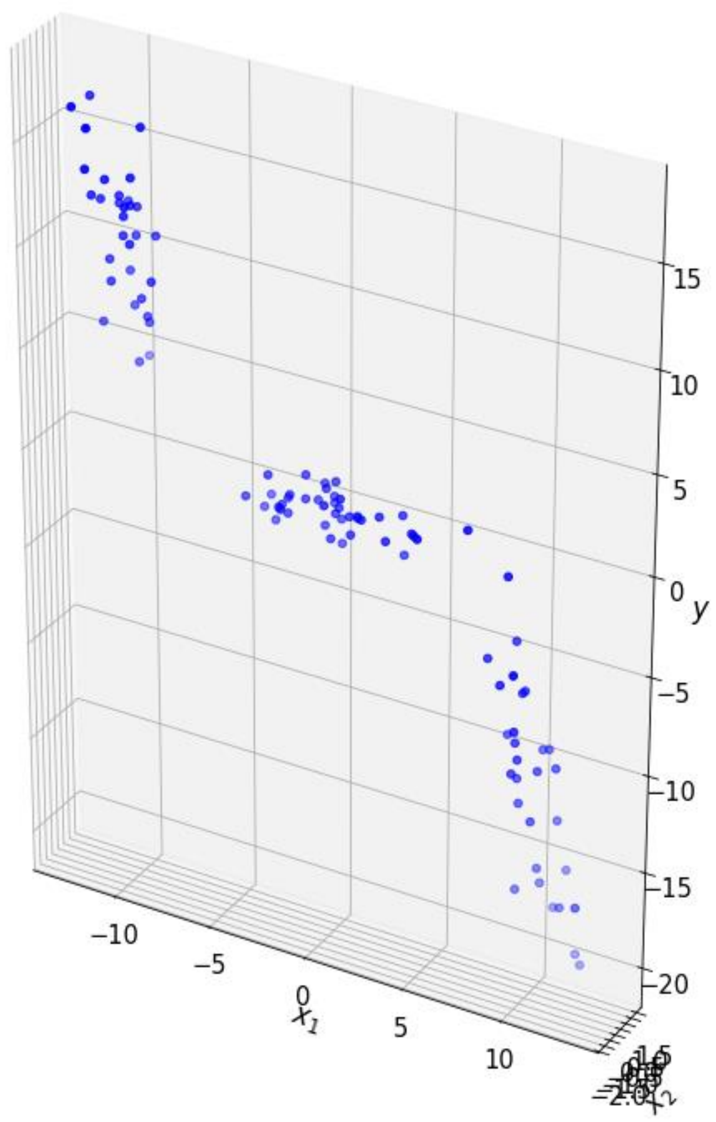
```python
def generate_data2(N, dataset_name):
    gmm_pdf = {}
    gmm_pdf['prior'] = np.array([.3, .4, .3])
    gmm_pdf['mo'] = np.array([[-10, 0, 10], [0, 0, 0], [10, 0, -10]])  # Gaussian distributions means
    gmm_pdf['Co'] = np.array([[[1, 0, -3], [0, 1, 0], [-3, 0, 15]], [[8, 0, 0], [0, .5, 0], [0, 0, .5]],
                             [[1, 0, -3], [0, 1, 0], [-3, 0, 15]]])  # Gaussian distributions covariance matrices

    X, y = generate_data_from_gmm(N, gmm_pdf)

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

    ax_raw = fig.add_subplot(111, projection='3d')

    ax_raw.scatter(X[:, 0], X[:, 1], y, marker='o', color='b')
    ax_raw.set_xlabel(r"$x_1$")
    ax_raw.set_ylabel(r"$x_2$")
    ax_raw.set_zlabel(r"$y$")
    # Set equal axes for 3D plots
    ax_raw.set_box_aspect((np.ptp(X[:, 0]), np.ptp(X[:, 1]), np.ptp(y)))

    plt.title("{} Dataset".format(dataset_name))
    plt.tight_layout()
    plt.show()

    return X, y
```



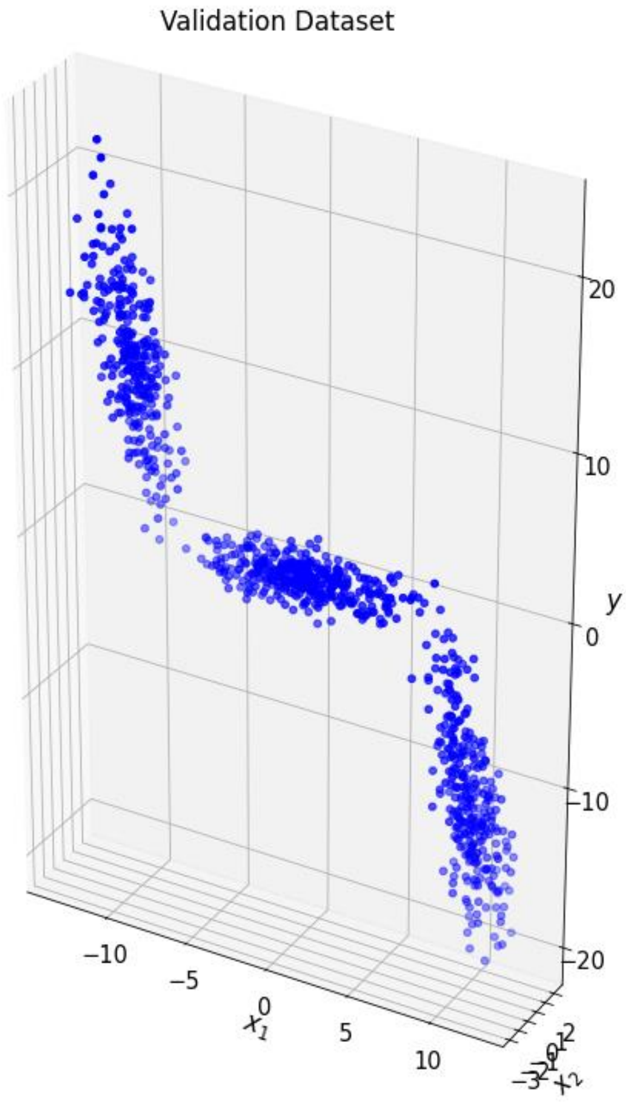
```python
def generate_data_from_gmm(N, gmm_pdf):
    # Decide randomly which samples will come from each component u_i ~ Uniform(0, 1) for i = 1, ..., N (or 0, ... , N-1 in code)
    u = np.random.rand(N)
    # Determine the thresholds based on the mixture weights/priors for the GMM, which need to sum up to 1
    thresholds = np.cumsum(gmm_pdf['prior'])
    thresholds = np.insert(thresholds, 0, 0)  # For intervals of classes

    n = gmm_pdf['mo'].shape[0]  # Data dimensionality

    X = np.zeros((N, n))
    C = len(gmm_pdf['prior'])  # Number of components
    for i in range(C + 1):
        # Get randomly sampled indices for this Gaussian, checking between thresholds based on class priors
        indices = np.argwhere((thresholds[i - 1] <= u) & (u <= thresholds[i]))[:, 0]
        # No. of samples in this Gaussian
        X[indices, :] = mvn.rvs(gmm_pdf['mo'][i - 1], gmm_pdf['Co'][i - 1], len(indices))

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



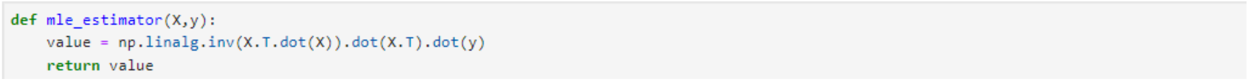
```python
X_train, y_train = generate_data2(100, "Training")
X_valid, y_valid = generate_data2(1000, "Validation")
```



Training Dataset



mle estimator(X,y): This function takes in a matrix X and a vector y, and computes the maximum likelihood estimator for linear regression coefficients using the formula inv(X.T.dot(X)).dot(X.T).dot(y), where inv() represents matrix inverse and dot() represents matrix multiplication. It returns the computed values.

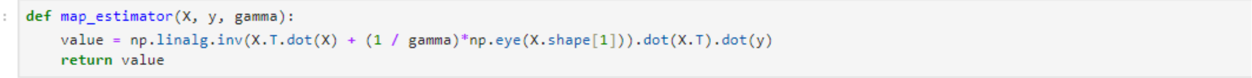


```
def mle_estimator(X,y):
    value = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
    return value
```

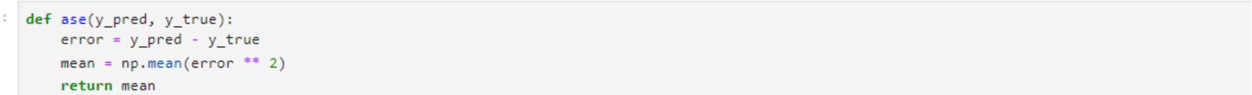
Having derived the estimator expressions, implement them in code and apply to the dataset generated by the attached Matlab script. Using the training dataset, obtain the ML estimator and the MAP estimator for a variety of γ values ranging from 10−m to 10n. Evaluate each trained model by calculating the average-squared error between the y values in the validation samples and model estimates of these using c(., wtrained).

Ans: map estimator(X, y, gamma): This function takes in a matrix X, a vector y, and a regularization parameter gamma, and computes the maximum a posteriori (MAP) estimator for linear regression coefficients using the formula inv(X.T.dot(X) + (1 / gamma)*np.eye(X.shape[1])).dot(X.T).dot(y), where inv() represents matrix inverse, dot() represents matrix multiplication, and np.eye() creates an identity matrix. It returns the computed values.

ase(y pred, y true): This function takes in two vectors y pred and y true, representing the predicted and true values of the target variable, respectively. It computes the average squared error (ASE) between the predicted and true values using the formula mean((y pred - y true) ** 2), where mean() computes the average of the given vector. It returns the computed value



```
def map_estimator(X, y, gamma):
    value = np.linalg.inv(X.T.dot(X) + (1 / gamma)*np.eye(X.shape[1])).dot(X.T).dot(y)
    return value
```

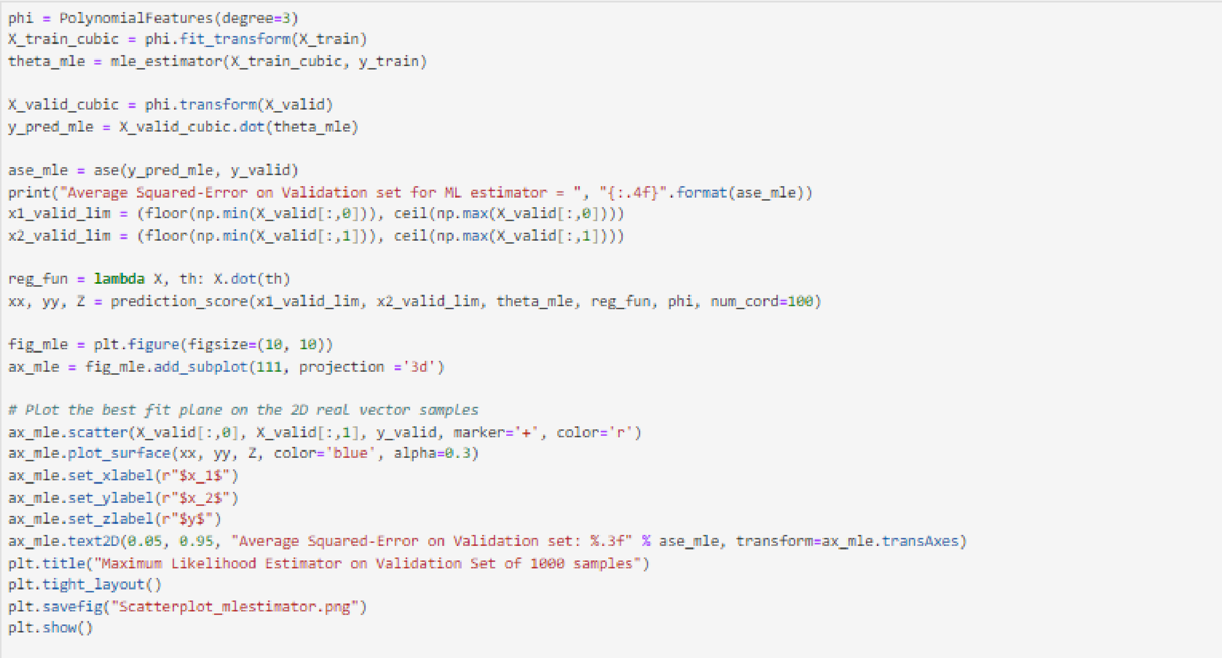


```
def ase(y_pred, y_true):
    error = y_pred - y_true
    mean = np.mean(error ** 2)
    return mean
```

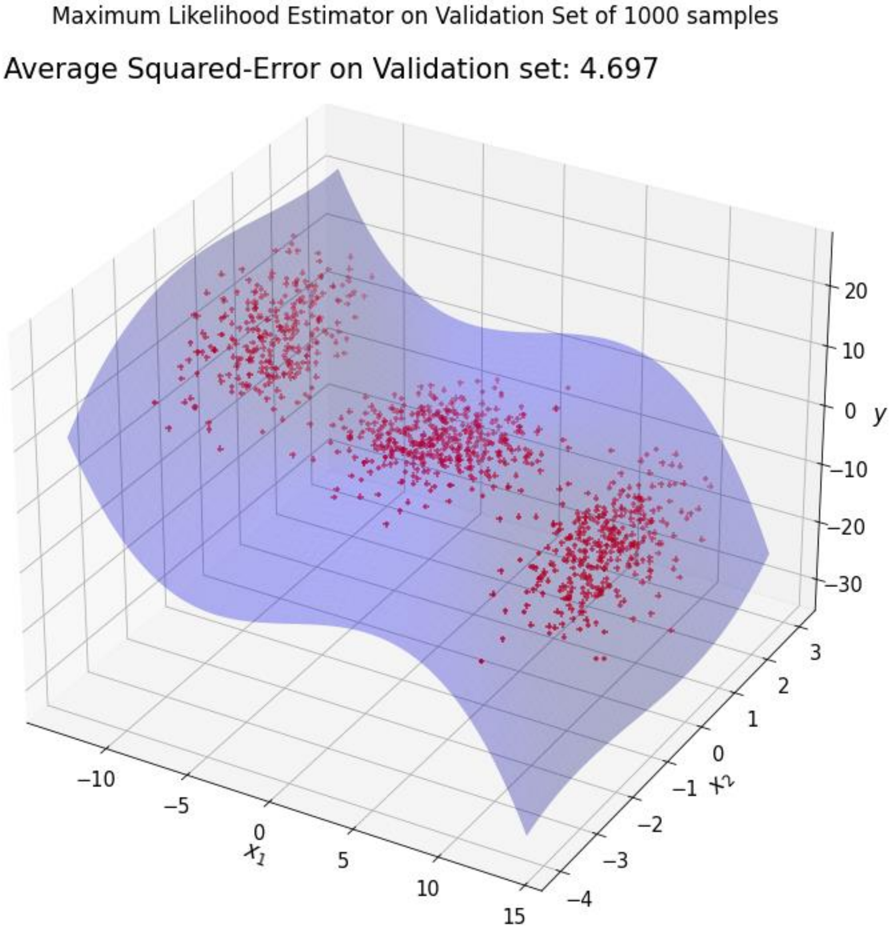
The PolynomialFeatures() function is used to transform the input features into higher degree polynomials. The degree of the polynomial is set to 3. The transformed features are used to train a maximum likelihood estimator (MLE) using the mle estimator() function. The MLE estimates the coefficients of the polynomial regression model.

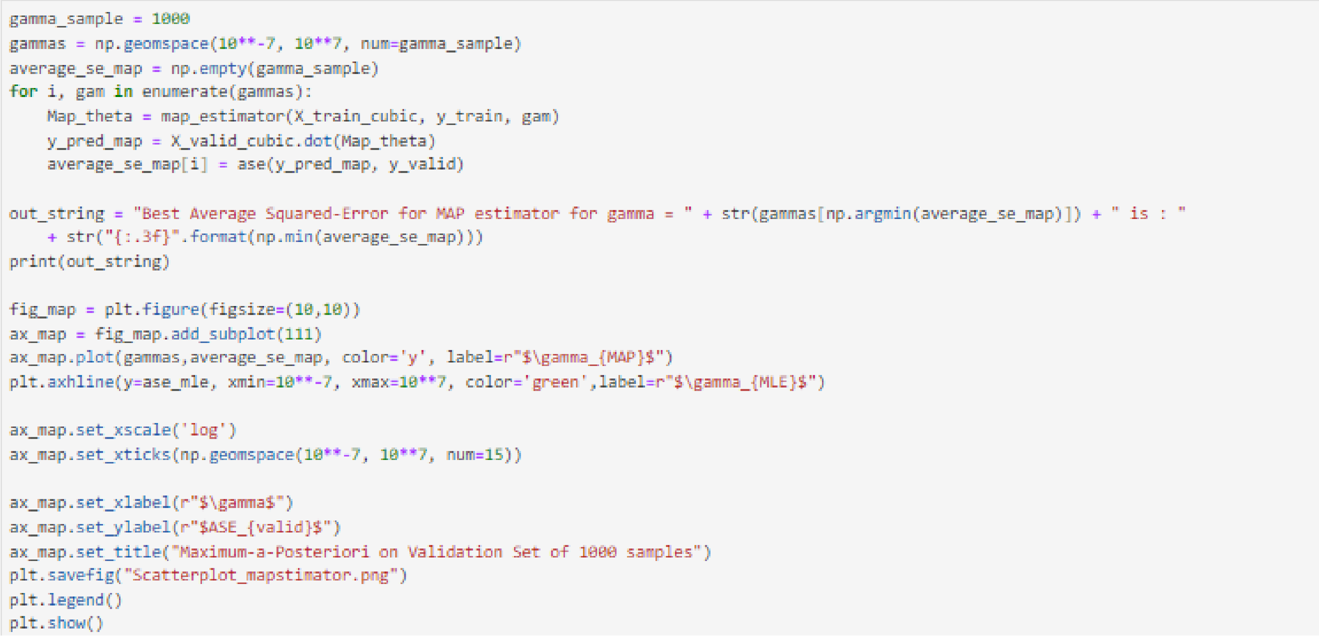
The MLE estimator is used to predict the output variable for the validation set, and the average squared error (ASE) between the predicted values and the actual values is computed using the ase() function. The ASE for the MLE estimator is printed.

A surface plot is created using the predicted values and the original validation set. Next, a maximum a posteriori (MAP) estimator is trained using the map estimator() function with a range of hyperparameters gamma. The hyperparameter that yields the lowest ASE on the validation set is chosen as the best hyperparameter for the MAP estimator. The ASE values for different hyperparameters are plotted against the corresponding hyperparameters using a log scale on the x-axis. Overall, the code performs polynomial regression on a dataset and compares the performance of the MLE and MAP estimators.

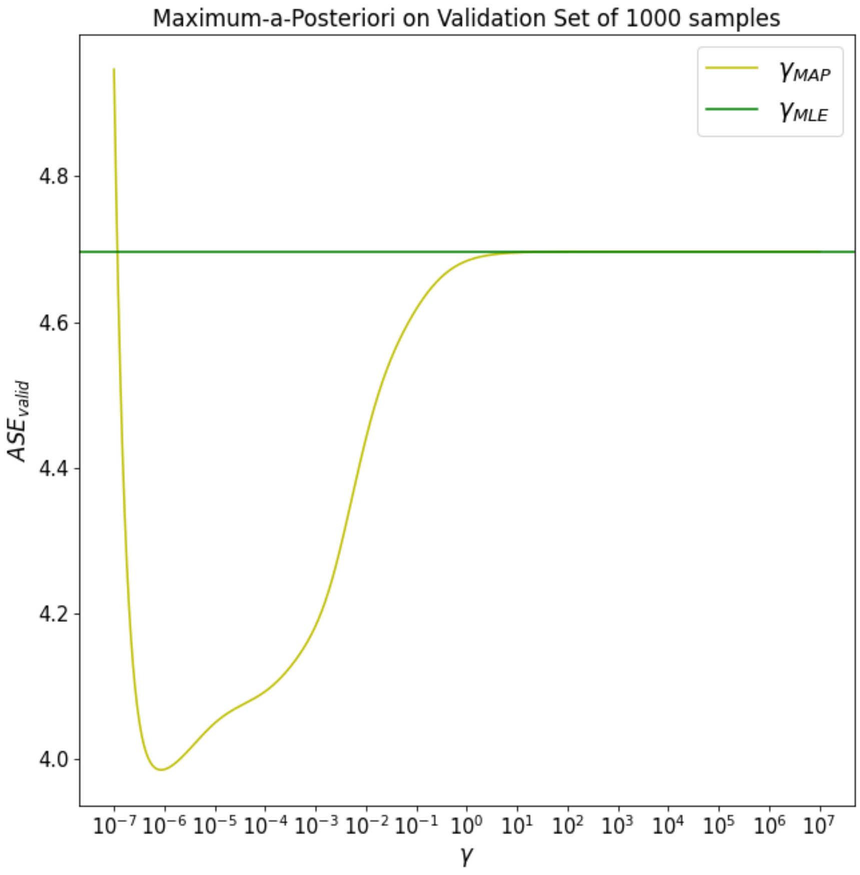


```
phi = PolynomialFeatures(degree=3)
X_train_cubic = phi.fit_transform(X_train)
theta_mle = mle_estimator(X_train_cubic, y_train)

X_valid_cubic = phi.transform(X_valid)
y_pred_mle = X_valid_cubic.dot(theta_mle)

ase_mle = ase(y_pred_mle, y_valid)
print("Average Squared-Error on Validation set for ML estimator = ", "{:.4f}".format(ase_mle))
x1_valid_lim = (floor(np.min(X_valid[:,0])), ceil(np.max(X_valid[:,0])))
x2_valid_lim = (floor(np.min(X_valid[:,1])), ceil(np.max(X_valid[:,1])))

reg_fun = lambda X, th: X.dot(th)
xx, yy, Z = prediction_score(x1_valid_lim, x2_valid_lim, theta_mle, reg_fun, phi, num_cord=100)

fig_mle = plt.figure(figsize=(10, 10))
ax_mle = fig_mle.add_subplot(111, projection ='3d')

# Plot the best fit plane on the 2D real vector samples
ax_mle.scatter(X_valid[:,0], X_valid[:,1], y_valid, marker='+', color='r')
ax_mle.plot_surface(xx, yy, Z, color='blue', alpha=0.3)
ax_mle.set_xlabel(r"$x_1$")
ax_mle.set_ylabel(r"$x_2$")
ax_mle.set_zlabel(r"$y$")
ax_mle.text2D(0.05, 0.95, "Average Squared-Error on Validation set: %.3f" % ase_mle, transform=ax_mle.transAxes)
plt.title("Maximum Likelihood Estimator on Validation Set of 1000 samples")
plt.tight_layout()
plt.savefig("Scatterplot_mlestimator.png")
plt.show()
```





```
gamma_sample = 1000
gammas = np.geomspace(10**-7, 10**7, num=gamma_sample)
average_se_map = np.empty(gamma_sample)
for i, gam in enumerate(gammas):
    Map_theta = map_estimator(X_train_cubic, y_train, gam)
    y_pred_map = X_valid_cubic.dot(Map_theta)
    average_se_map[i] = ase(y_pred_map, y_valid)

out_string = "Best Average Squared-Error for MAP estimator for gamma = " + str(gammas[np.argmin(average_se_map)]) + " is : "
    + str("{:.3f}".format(np.min(average_se_map)))
print(out_string)

fig_map = plt.figure(figsize=(10,10))
ax_map = fig_map.add_subplot(111)
ax_map.plot(gammas,average_se_map, color='y', label=r"$\gamma_{MAP}$")
plt.axhline(y=ase_mle, xmin=10**-7, xmax=10**7, color='green',label=r"$\gamma_{MLE}$")

ax_map.set_xscale('log')
ax_map.set_xticks(np.geomspace(10**-7, 10**7, num=15))

ax_map.set_xlabel(r"$\gamma$")
ax_map.set_ylabel(r"$ASE_{valid}$")
ax_map.set_title("Maximum-a-Posteriori on Validation Set of 1000 samples")
plt.savefig("Scatterplot_mapstimator.png")
plt.legend()
plt.show()
```



Discussion:

How does your MAP-trained model perform on the validation set as γ is varied? How is the MAP estimate related to the ML estimate? Describe your experiments, visualize and quantify your analyses (e.g. average squared error on validation dataset as a function of hyperparameter γ) with data from these experiments.

Ans: The ML estimator and MAP estimator are two different methods used to estimate the parameters of a model given some training data. The ML estimator uses the maximum likelihood principle to find the values of the parameters that maximize the likelihood of observing the given data. In other words, it tries to find the parameters that make the observed data the most likely under the assumed model. In this code, the ML estimator is used to find the parameters of a linear regression model. The MAP estimator, on the other hand, is a Bayesian approach that uses prior knowledge about the parameters to find the posterior distribution of the parameters given the data. It tries to find the values of the parameters that maximize the posterior probability. In this code, the MAP estimator is used with a regularization term (gamma) to find the parameters of a regularized linear regression model. To compare the performance of the two estimators, the Average Squared-Error (ASE) is calculated on a validation set. The ASE is a measure of the quality of the model's predictions, and it measures the average of the squared differences between the predicted values and the true values on the validation set.

Overall, the MAP estimator is expected to perform better than the ML estimator when the data is noisy or when the number of features is large, because it can incorporate prior knowledge to avoid overfitting. However, it may require tuning of the regularization parameter (gamma) to achieve the best performance.

Question 3

Part 1:

A vehicle at true position [xT , yT]T in 2-dimensional space is to be localized using distance (range) measurements to K reference (landmark) coordinates {[x1, y1]T, . . . , [xi, yi]T, . . . , [xK , yK]T}. These range measurements are ri = dTi + ni for i ∈ {1, . . . , K}, where dTi = ‖[xT , yT]T − [xi, yi]T‖ is the true distance between the vehicle and the ith reference point, and ni is a zero mean Gaussian distributed measurement noise with known variance σ_i^2 . The noise in each measurement is independent from the others.

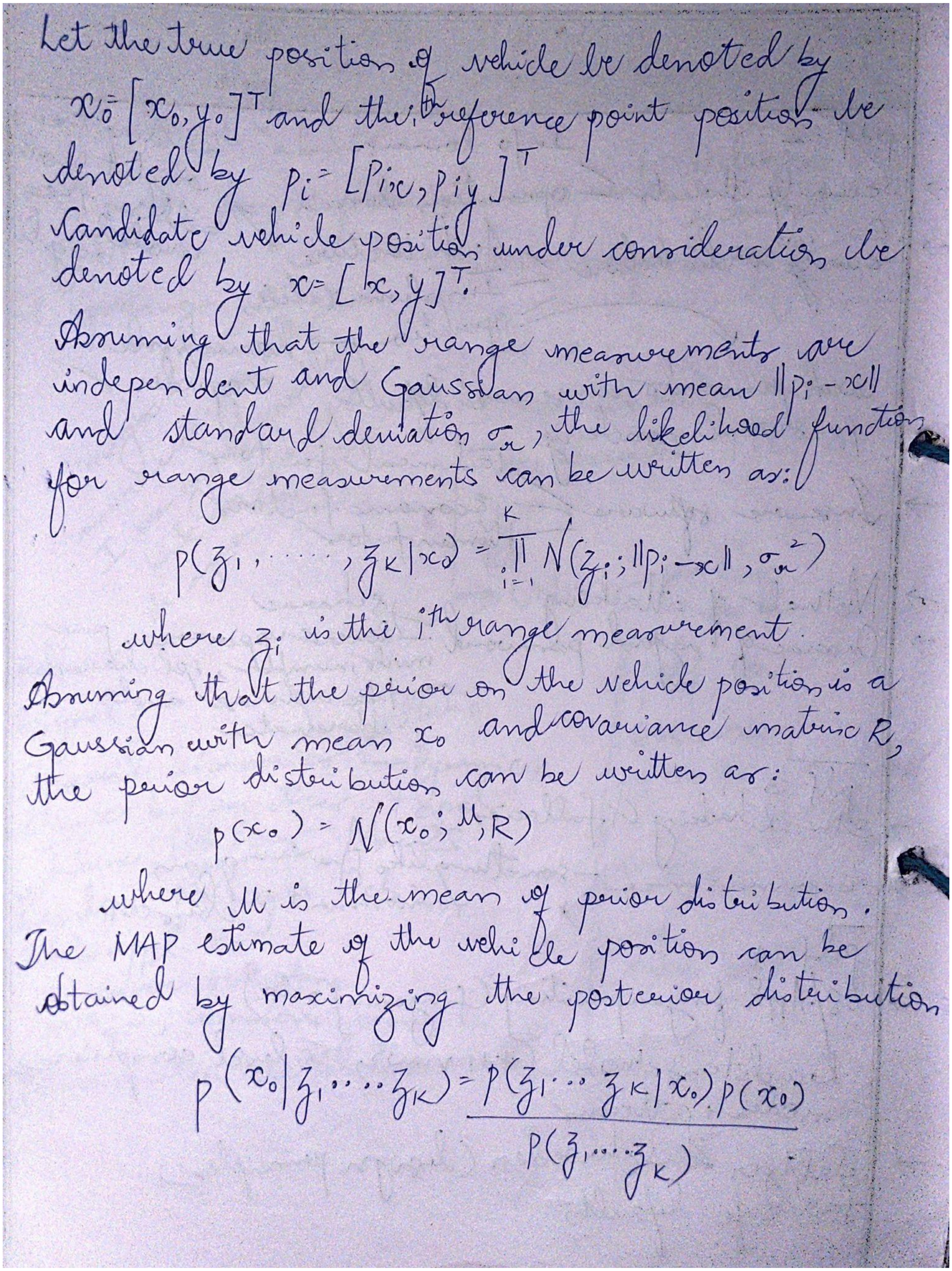
Assume that we have the following prior knowledge regarding the position of the vehicle:

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

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

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

Ans:



Let the true position of vehicle be denoted by $x_0 = [x_0, y_0]^T$ and the i^{th} reference point position be denoted by $p_i = [p_{ix}, p_{iy}]^T$

Candidate vehicle position under consideration be denoted by $x = [x, y]^T$.

Assuming that the range measurements are independent and Gaussian with mean $\|p_i - x\|$ and standard deviation σ_r, the likelihood function for range measurements can be written as:

$$p(z_1, \ldots\ldots, z_K | x_0) = \prod_{i=1}^{K} N(z_i; \|p_i - x\|, \sigma_r^2)$$

where z_i is the i^{th} range measurement.

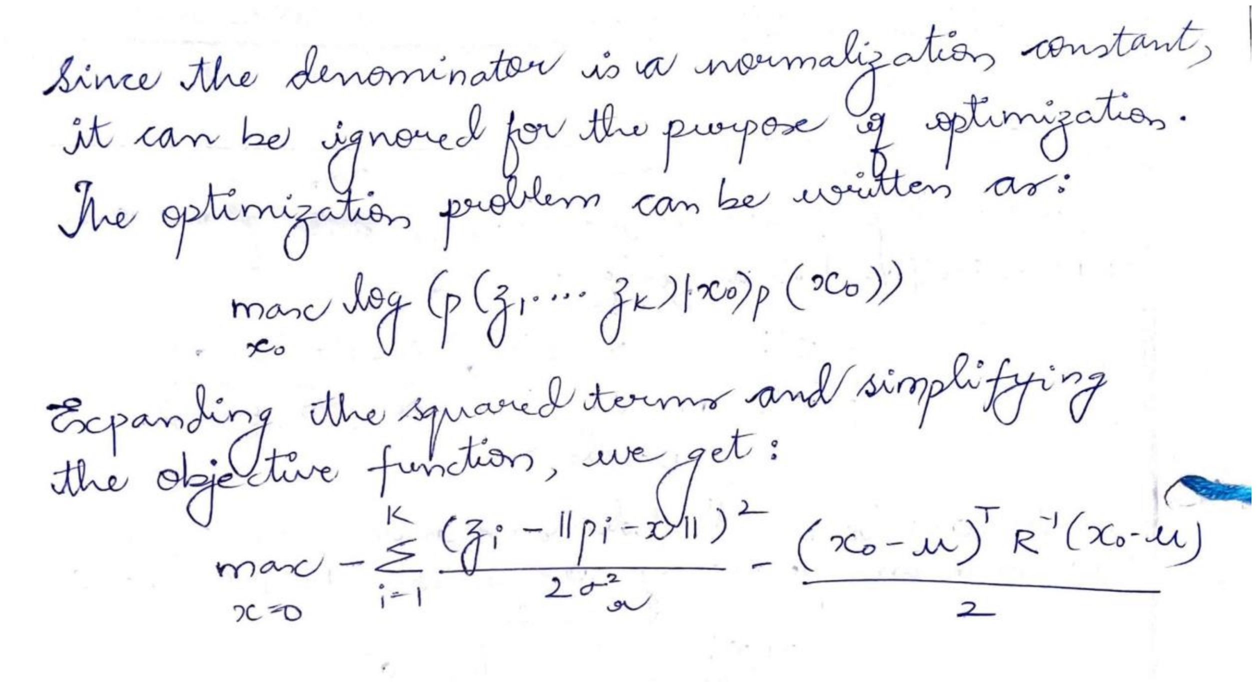
Assuming that the prior on the vehicle position is a Gaussian with mean x_0 and covariance matrix R, the prior distribution can be written as:

$$p(x_0) = N(x_0; \mu, R)$$

where μ is the mean of prior distribution.

The MAP estimate of the vehicle position can be obtained by maximizing the posterior distribution

$$p(x_0 | z_1 \ldots\ldots z_K) = \frac{p(z_1 \ldots z_K | x_0)\, p(x_0)}{p(z_1 \ldots\ldots z_K)}$$



Since the denominator is a normalization constant, it can be ignored for the purpose of optimization. The optimization problem can be written as:

$$\max_{x_0} \log (p(z_1 \dots z_K)|x_0)p(x_0))$$

Expanding the squared terms and simplifying the objective function, we get:

$$\max_{x=0} -\sum_{i=1}^{K} \frac{(z_i - \|p_i - x\|)^2}{2\sigma_r^2} - \frac{(x_0 - \mu)^T R^{-1}(x_0 - \mu)}{2}$$

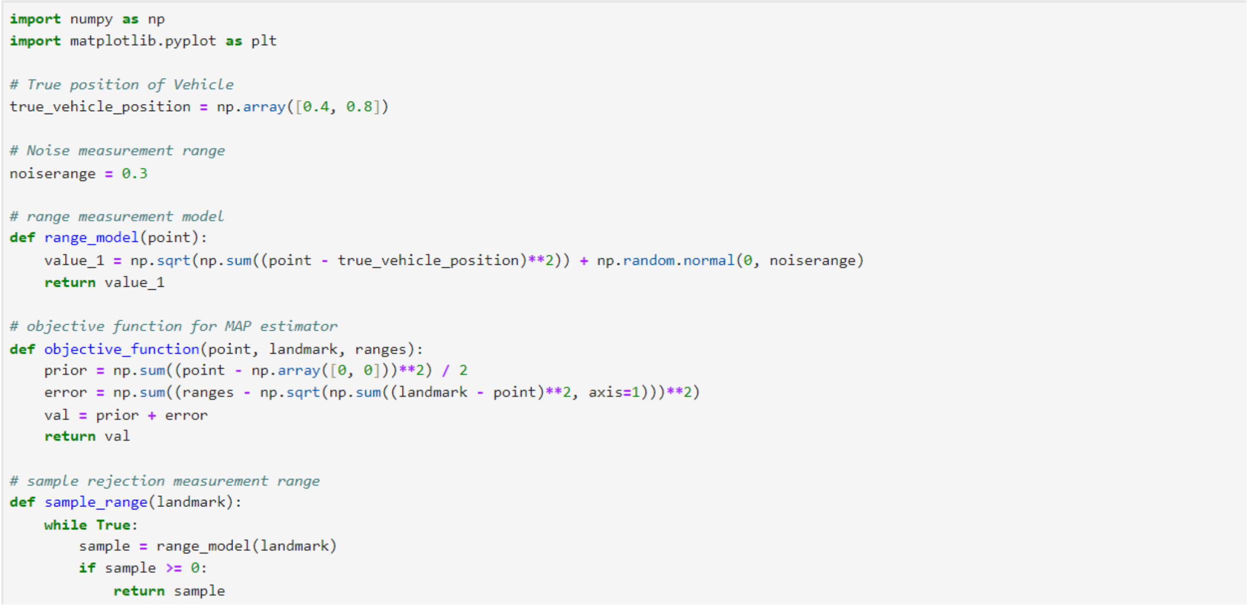
Part 2: Implement the following as computer code: Set the true vehicle location to be inside the circle with unit radious centered at the origin. For each K ∈ {1, 2, 3, 4} repeat the following. Place evenly spaced K landmarks on a circle with unit radius centered at the origin. Set measurement noise 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).

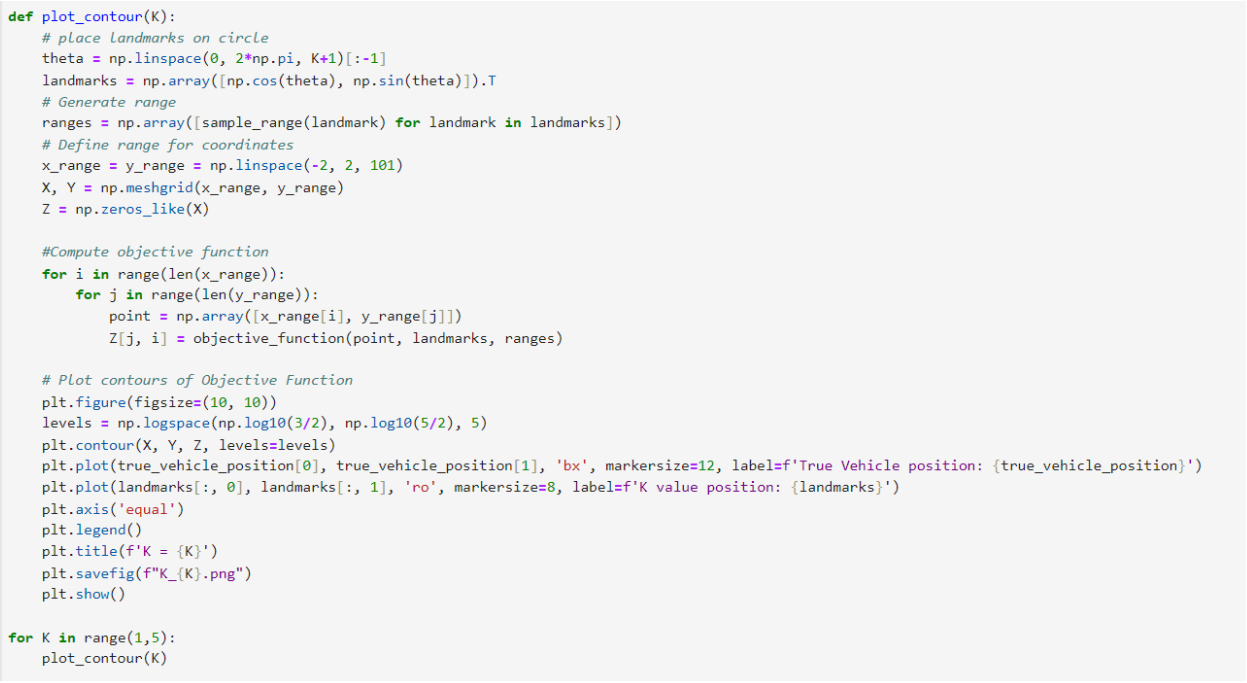
Provide plots of the MAP objective function contours for each value of K. When preparing your final contour plots for different K values, make sure to plot contours at the same function value across each of the different contour plots for easy visual comparison of the MAP objective landscapes.

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.

Ans:



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

# True position of Vehicle
true_vehicle_position = np.array([0.4, 0.8])

# Noise measurement range
noiserange = 0.3

# range measurement model
def range_model(point):
    value_1 = np.sqrt(np.sum((point - true_vehicle_position)**2)) + np.random.normal(0, noiserange)
    return value_1

# objective function for MAP estimator
def objective_function(point, landmark, ranges):
    prior = np.sum((point - np.array([0, 0]))**2) / 2
    error = np.sum((ranges - np.sqrt(np.sum((landmark - point)**2, axis=1)))**2)
    val = prior + error
    return val

# sample rejection measurement range
def sample_range(landmark):
    while True:
        sample = range_model(landmark)
        if sample >= 0:
            return sample
```



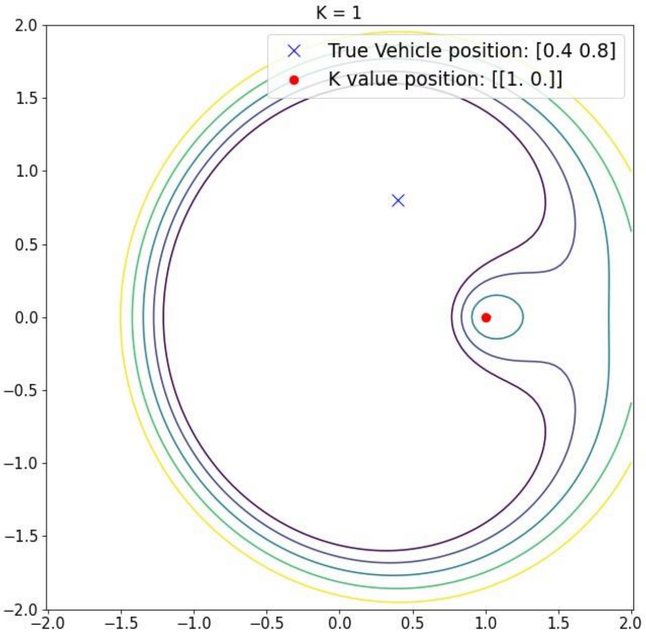
```python
def plot_contour(K):
    # place landmarks on circle
    theta = np.linspace(0, 2*np.pi, K+1)[:-1]
    landmarks = np.array([np.cos(theta), np.sin(theta)]).T
    # Generate range
    ranges = np.array([sample_range(landmark) for landmark in landmarks])
    # Define range for coordinates
    x_range = y_range = np.linspace(-2, 2, 101)
    X, Y = np.meshgrid(x_range, y_range)
    Z = np.zeros_like(X)

    #Compute objective function
    for i in range(len(x_range)):
        for j in range(len(y_range)):
            point = np.array([x_range[i], y_range[j]])
            Z[j, i] = objective_function(point, landmarks, ranges)

    # Plot contours of Objective Function
    plt.figure(figsize=(10, 10))
    levels = np.logspace(np.log10(3/2), np.log10(5/2), 5)
    plt.contour(X, Y, Z, levels=levels)
    plt.plot(true_vehicle_position[0], true_vehicle_position[1], 'bx', markersize=12, label=f'True Vehicle position: {true_vehicle_position}')
    plt.plot(landmarks[:, 0], landmarks[:, 1], 'ro', markersize=8, label=f'K value position: {landmarks}')
    plt.axis('equal')
    plt.legend()
    plt.title(f'K = {K}')
    plt.savefig(f"K_{K}.png")
    plt.show()

for K in range(1,5):
    plot_contour(K)
```

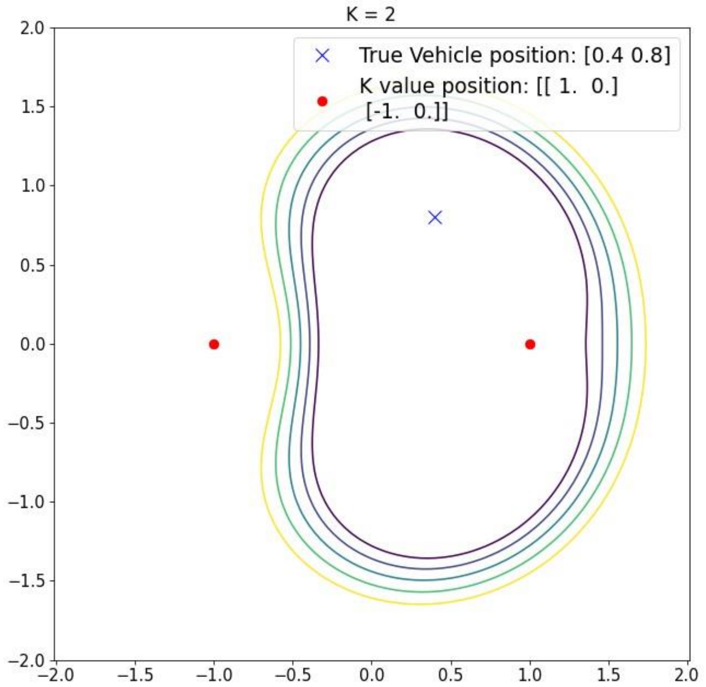
The Python code is implementing a simulation of a range-based localization problem. It defines a scenario where a vehicle is located at a true position, and several landmarks are placed on a circle around the vehicle. The objective is to estimate the position of the vehicle based on range measurements from the landmarks, where the range measurement is affected by Gaussian noise. The code is structured as follows:

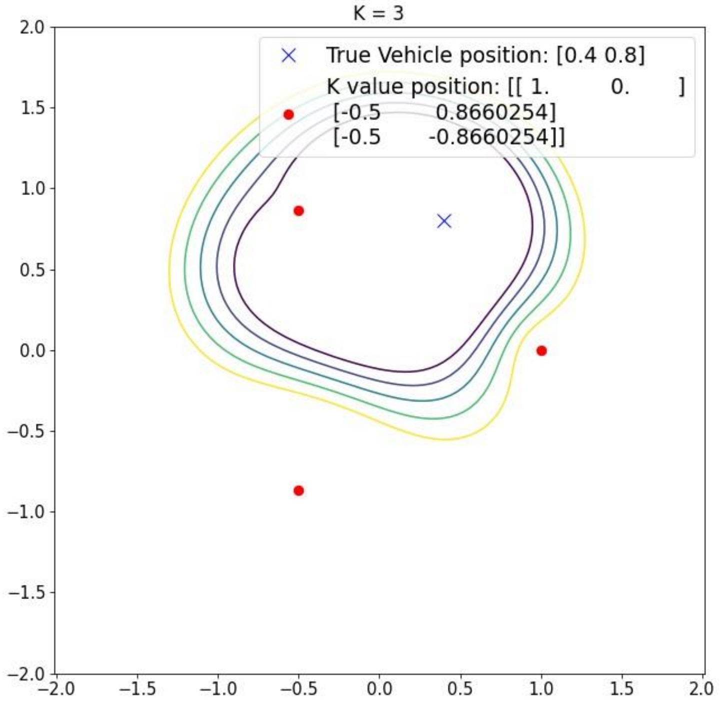
First, the required libraries, NumPy and Matplotlib, are imported. The true position of the vehicle is defined as a 2D NumPy array. The noise range variable is defined, which represents the standard deviation of the Gaussian noise affecting the range measurements. The range model function is defined, which takes a point as an input and returns a range measurement value with Gaussian noise added to it. This function uses the true vehicle position and noise range variables to generate the range measurement.

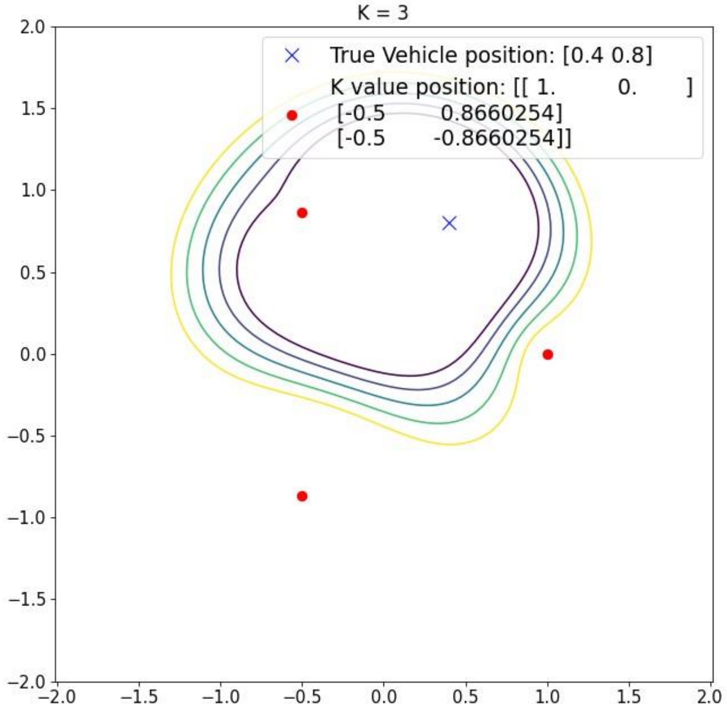
The objective function is defined, which takes a point, an array of landmarks, and an array of ranges as inputs, and returns the value of the objective function for these inputs. The objective function is a sum of two terms: a prior term that favors points close to the origin (0,0), and an error term that measures the difference between the measured ranges and the ranges calculated from the distances between the landmarks and the point. The function returns the sum of these two terms. The sample range function is defined, which takes a landmark as input and generates a range measurement for that landmark using the range model function.

This function uses a rejection sampling technique to ensure that the range measurement is non-negative. The plot contour function is defined, which takes a value of K as input and generates a plot of the objective function contours for a scenario with K landmarks. The function first places the landmarks on a circle using the np.linspace function. It then generates range measurements for each landmark using the sample range function. The function defines a range for the x and y coordinates and creates a meshgrid of points using np.meshgrid. For each point in the meshgrid, it calculates the objective function using the objective function function and stores the value in a 2D NumPy array. Finally, the function generates a contour plot of the objective function using plt.contour, where the levels of the contours are defined using np.logspace. It also plots the true position of the vehicle and the positions of the landmarks using plt.plot, and saves the plot as a PNG file. The for loop at the end calls the plot contour function for values of K from 1 to 4, generating four different plots of the objective function contours with different numbers of landmarks.

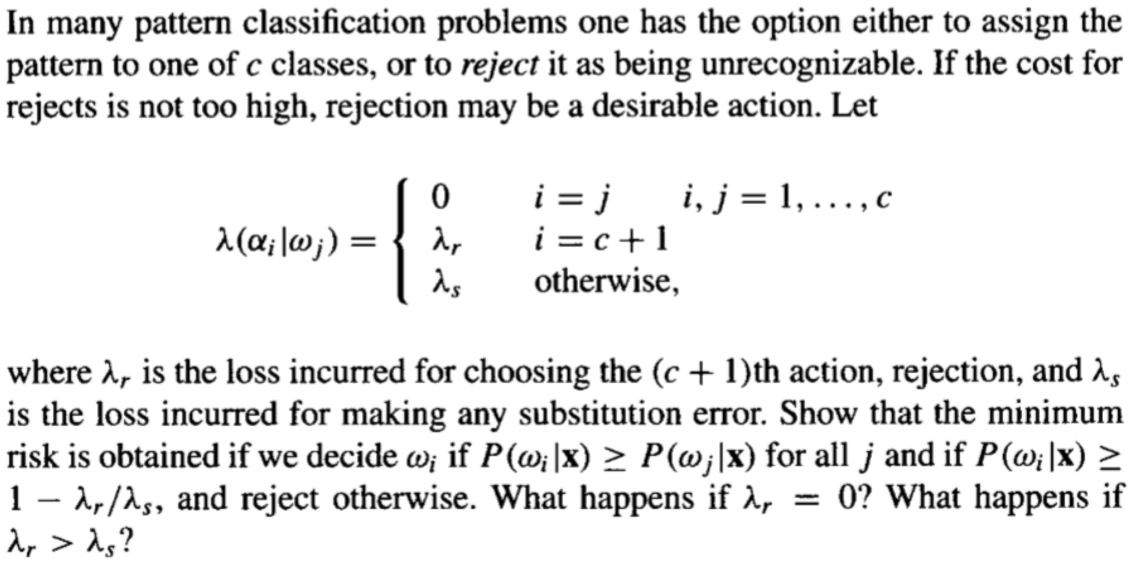



K = 2
True Vehicle position: [0.4 0.8]
K value position: [[1. 0.]
 [-1. 0.]]
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


K = 3
True Vehicle position: [0.4 0.8]
K value position: [[1.   0.   ]
 [-0.5   0.8660254]
 [-0.5   -0.8660254]]
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0



Question 4

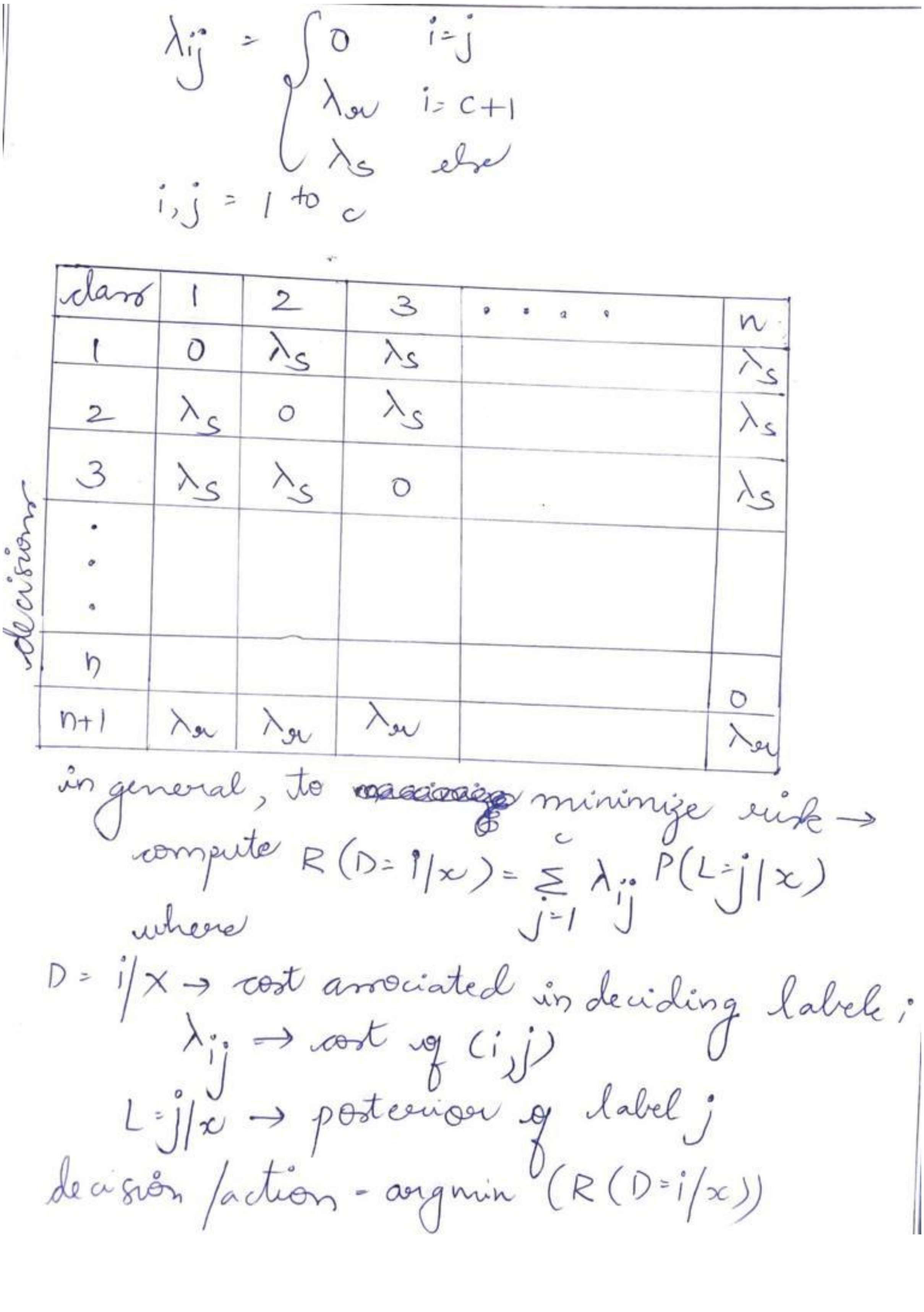


In many pattern classification problems one has the option either to assign the pattern to one of c classes, or to *reject* it as being unrecognizable. If the cost for rejects is not too high, rejection may be a desirable action. Let

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

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 $\lambda_r > \lambda_s$?

Ans:



$$\lambda_{ij} = \begin{cases} 0 & i=j \\ \lambda_r & i=c+1 \\ \lambda_s & \text{else} \end{cases}$$

$i, j = 1$ to c

decisions

class	1	2	3	. . . .	n
1	0	λ_s	λ_s		λ_s
2	λ_s	0	λ_s		λ_s
3	λ_s	λ_s	0		λ_s
⋮					
n					0
n+1	λ_r	λ_r	λ_r		λ_r

in general, to minimize risk →

compute $R(D=i|x) = \sum_{j=1}^{c} \lambda_{ij} P(L=j|x)$

where

$D = i/X$ → cost associated in deciding label i

λ_{ij} → cost of (i,j)

$L=j/x$ → posterior of label j

decision/action = argmin $(R(D=i/x))$

For our problem, considering classes i and k, risk associated in deciding class i,

$$R(D=i|x) = \sum_{j=1}^{c} \lambda_{ij} P(L=j|x).$$

$$= \lambda_s \left[\sum_{\substack{j \in \text{class} \\ \text{labels} \\ j \neq i}} P(L=j|x)\right] — \textcircled{1}$$

where $\lambda_{ij} = 0$ when $j = 1$ hence that term has to be ignored.

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

$$\therefore \sum_{\substack{j \in \text{class} \\ \text{labels} \\ j \neq i}} P(L=j|x) = 1 - P[L=i|x]$$

∴ equation ① becomes,

$$R(D=i|x) = \lambda_s \left[1 - P[L=i|x]\right] — \textcircled{2}$$

Risk associated with deciding class K →

$$R(D=K|x) = \lambda_s \left[1 - P[L=K|x]\right] - \textcircled{3}$$

To decide between class i and class K →

Decide class i if only if →

$$R(D=i|x) < R(D=K|x)$$

Rephrasing → $R(D=i|K) \underset{D=i}{\overset{D=K}{\gtrless}} R(D=K|x)$

substitute values of risk from risk equation ② and ③,

$$\lambda_s[1-p[L=i|x]] \underset{D=i}{\overset{D=K}{\gtrless}} \lambda_s[1-p(L=K|x)]$$

$$P[L=i|x] \underset{D=K}{\overset{D=i}{\gtrless}} P[L=K|x]$$

decide class i if $P[L=i/x] \geq P[L=K/x]$ for all class of K, $K=[1 \cdots c]; K \neq i$

We still need to decide between class i and class (c+1),

$$\boxed{\text{reject class} \rightarrow \text{class } (c+1)}$$

Risk associated with reject class →

$$R[D=c+1|X] = \sum_{i=1}^{c} \lambda_{ij} \cdot P(L=j|x)$$

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

$$= \lambda_r \rightarrow ④$$

$\rightarrow p(L=i|x) \geq (1-\lambda_R/\lambda_s)$ — ⑤

if $p(L=i|x) < (1-\lambda_R/\lambda_S)$;
reject class $(c+1)$

① $\lambda_r = 0$ on eq ⑤

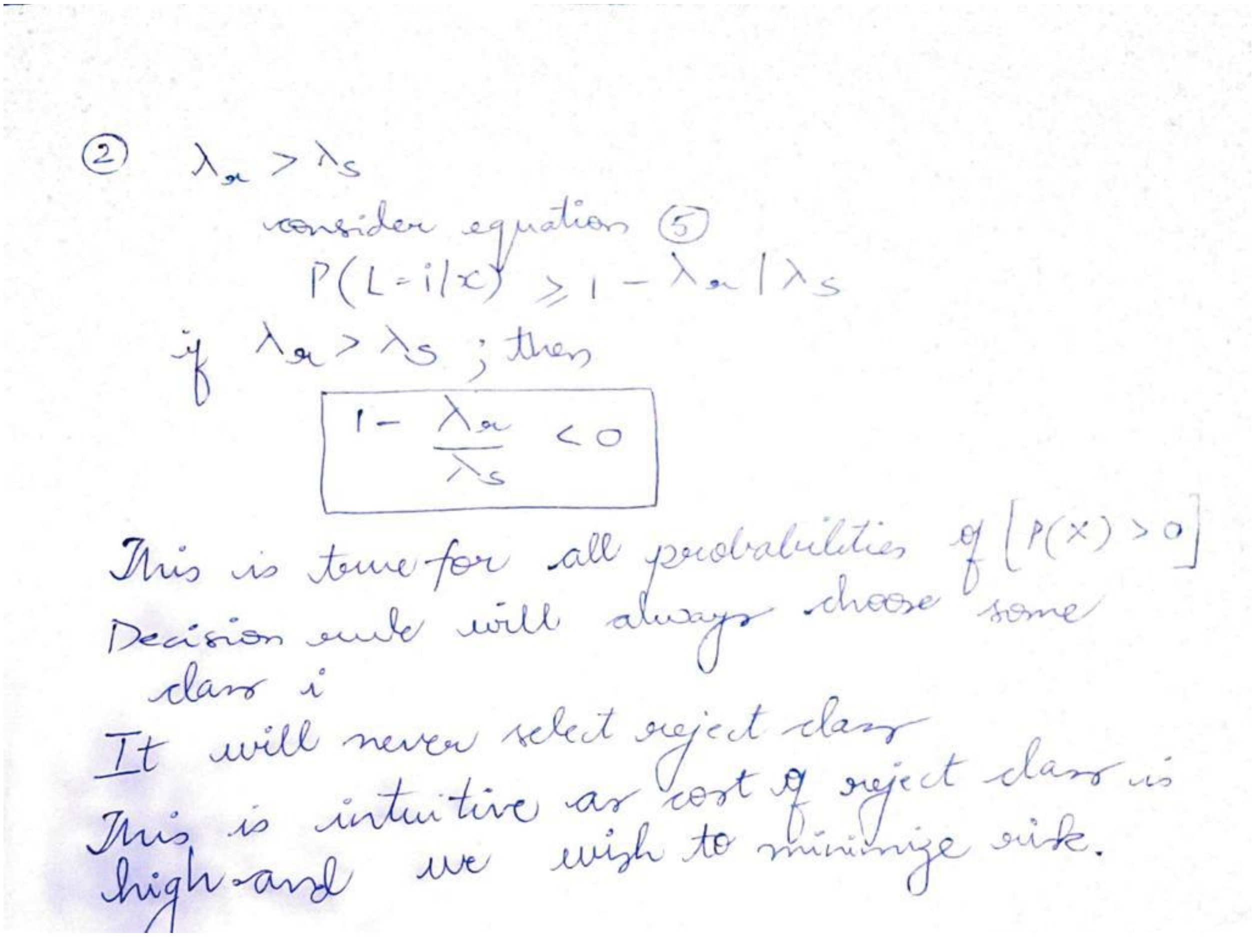
$$p(L=i|x) \geq 1 - 0/\lambda_S$$

$$p(L=i|x) \geq 1$$

it is very rare to see that the decide class is i only if posterior ≥ 1

Since $\lambda_r = 0$; cost associated with reject class is 0.

Hence, decision rule will always decide the reject class.



② $\lambda_r > \lambda_s$

consider equation ⑤

$P(L=i|x) \geq 1 - \lambda_r / \lambda_s$

if $\lambda_r > \lambda_s$; then

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

This is true for all probabilities of $[P(x) > 0]$

Decision rule will always choose some class i

It will never select reject class

This is intuitive as cost of reject class is high and we wish to minimize risk.

Question 5

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

Given $D\{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$ with iid samples $\mathbf{z}_n \sim Cat(\Theta)$ for $n \in \{1, \ldots, N\}$:

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

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

$$p(\Theta|\boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1} \text{ where the normalization constant is } B(\boldsymbol{\alpha}) = \frac{\prod_{k=1}^{K} \Gamma(\alpha_k)}{\Gamma(\sum_{k=1}^{K} \alpha_k)}$$

The ML estimator for θ is as follows:

$$p(Z_n = K|\theta) = \theta_K \, , \prod_{K=1}^{K} \theta_K = 1$$

The MAP estimator, assuming that the prior $p(\theta)$ used for the parameters is a Dirichlet distribution with hyperparameter α, is as follows:

$$p(\theta|z,\alpha) \propto p(z|\theta)\,p(\theta|\alpha) =$$

$$\prod_{n=1}^{N} \prod_{K=1}^{K} \theta_K^{z_{nK}} \frac{1}{B(\alpha)} \prod_{K=1}^{K} \theta_K^{\alpha_K - 1} \propto$$

$$\prod_{K=1}^{K} \theta_K^{\sum_{n=1}^{N} z_{n1} + \cdots + \alpha_{K-1}}$$

This is the kernel of a Dirichlet distribution with parameter vector $\alpha' = (\alpha_1 + \sum_{n=1}^{N} z_{n1}, \dots \alpha_K + \sum_{n=1}^{N} z_{nK})$:

$$p(\theta|z,\alpha) = \frac{1}{B(\alpha')} \prod_{K=1}^{K} \theta_K^{\alpha_K + \sum_{n=1}^{N} z_{nK} - 1}$$

where $B(\alpha) = \frac{\prod_{K=1}^{K} \Gamma(\alpha_K)}{\Gamma(\sum_{K=1}^{K} \alpha_K)}$ is the normalization constant of the Dirichlet distribution.