Assignment 3 - Mahdiar Edraki

Monday, November 16, 2020 8:04 AM

Question 1 (60%)

In this exercise, you will train many multilayer perceptrons (MLP) to approximate the class label posteriors, using maximum likelihood parameter estimation (equivalently, with minimum average cross-entropy loss) to train the MLP. Then, you will use the trained models to approximate a MAP classification rule in an attempt to achieve minimum probability of error (i.e. to minimize expected loss with 0-1 loss assignments to correct-incorrect decisions).

Data Distribution: For $C = 4$ classes with uniform priors, specify Gaussian class-conditional pdfs for a 3-dimensional real-valued random vector $\mathbf{x}$ (pick your own mean vectors and covariance matrices for each class). Try to adjust the parameters of the data distribution so that the MAP classifier that uses the true data pdf achieves between $10\% - 20\%$ probability of error.

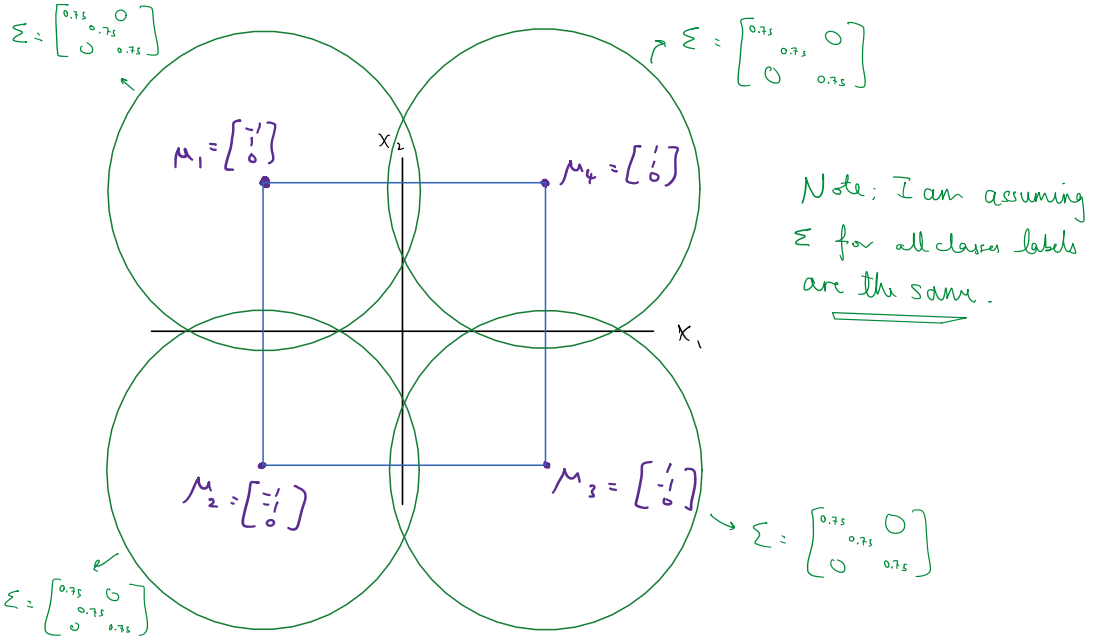
For the data distribution, I will use a similar distribution as Take-home exam I:

Given:

Random vector $X \in \mathbb{R}^3$

4 class labels, $L \in \{1, 2, 3, 4\}$

Let's visualize the Gaussian class conditional pdfs:

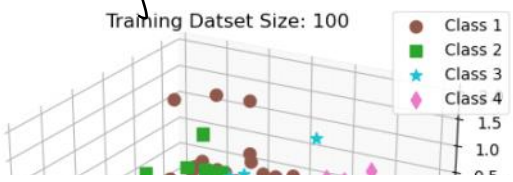


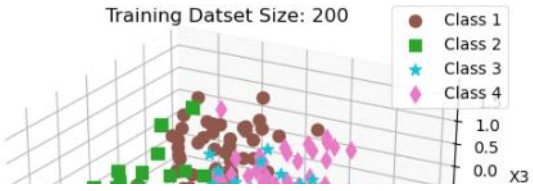
Note: I am assuming Σ for all classes labels are the same.

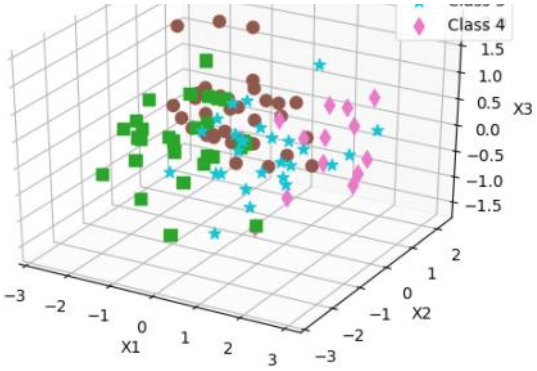
↳ From this distribution, I will generate the training and test datasets.

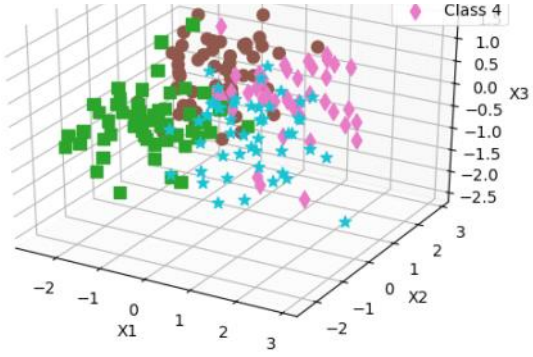
Generate Data: Using your specified data distribution, generate multiple datasets: Training datasets with $100, 200, 500, 1000, 2000, 5000$ samples and a test dataset with 100000 samples. You will use the test dataset only for performance evaluation.

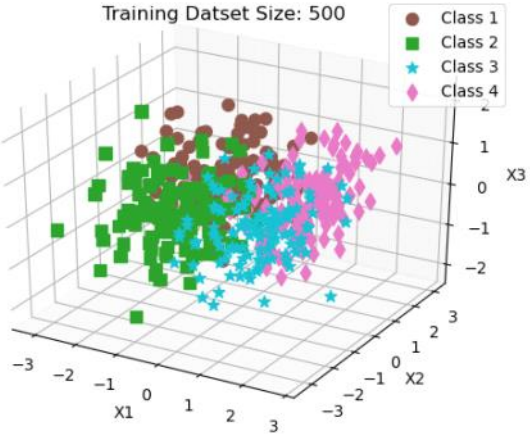
Training Datasets:

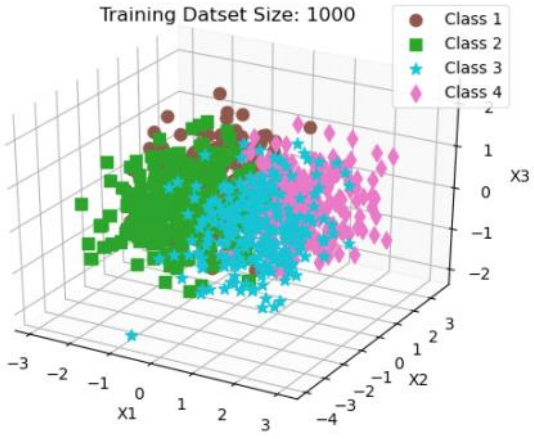


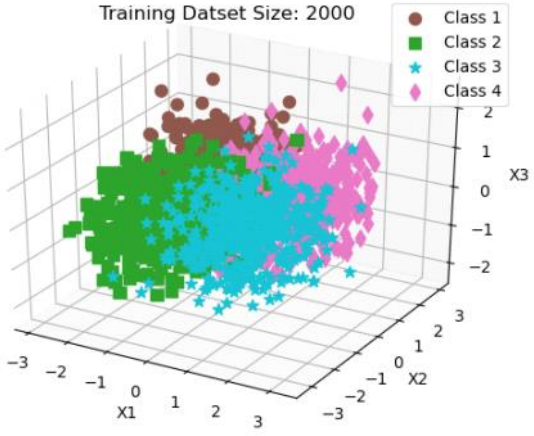


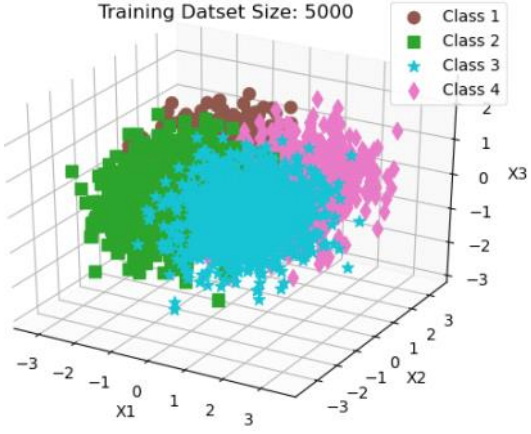




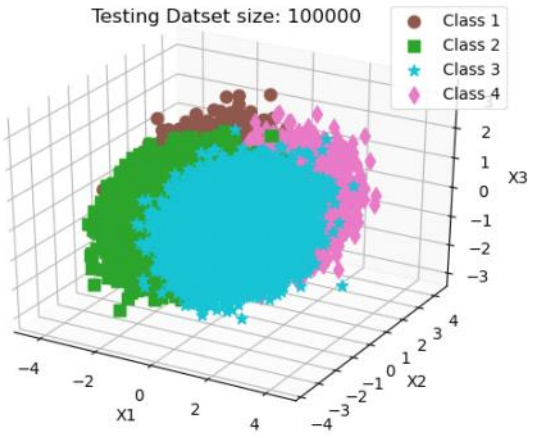








Testing Dataset :



Theoretically Optimal Classifier: Using the knowledge of your true data pdf, construct the minimum-probability-of-error classification rule, apply it on the test dataset, and empirically estimate the probability of error for this theoretically optimal classifier. This provides the aspirational performance level for the MLP classfier.

Our goal is to find the following:

$$\begin{bmatrix} R(D=1|x) \\ \vdots \\ R(D=C|x) \end{bmatrix} = \Lambda \begin{bmatrix} P(L=1|x) \\ \vdots \\ P(L=C|x) \end{bmatrix}$$

Expected Risk

$R(D=1|x)$ ... $R(D=C|x)$ → using our decisions = labels

Λ → loss matrix

$P(L=1|x)$ ... $P(L=C|x)$ → class posteriors

Once we find the Expected Risks, we pick the one with the lowest risk.

- First we need to find class posteriors:

Using Bayes Rule: $P(L=i|x) = \dfrac{p(x|L=i)\,p(L=i)}{p(x)}$

- $p(x) = \sum_{i=1}^{C} p(x|L=j)\,p(L=j)$ → equal scaling factor across all risks. Can be ignored for our purposes.
- $p(L=i)$ = class prior = given ✓
- $p(x|L=i)$ = gaussian pdf with μ_i and Σ ✓

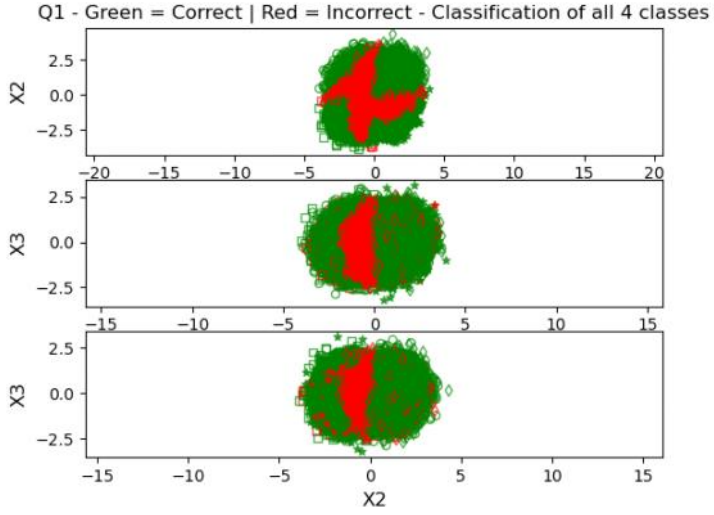
↓ Rewritten as

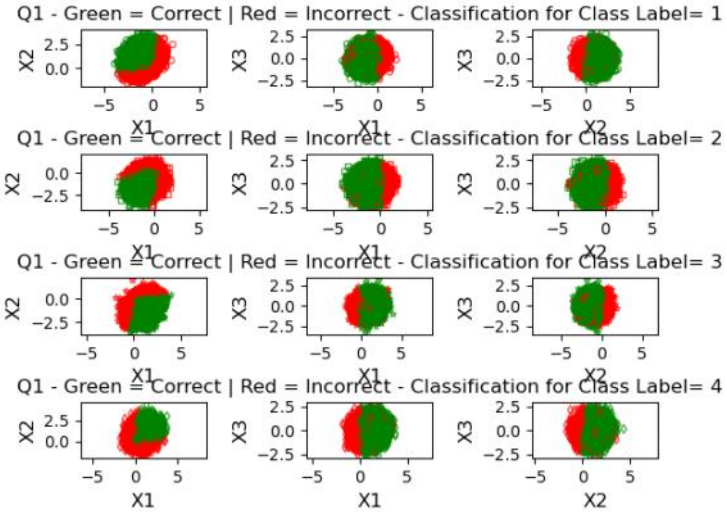
$$\begin{bmatrix} R(D=1|x) \\ \vdots \\ R(D=C|x) \end{bmatrix} = \Lambda \text{ (ignoring } \tfrac{1}{p(x)}\text{)} \begin{bmatrix} p(x|L=1)\,p(L=1) \\ \vdots \\ p(x|L=C)\,p(L=C) \end{bmatrix}$$

- Assume Λ is a 0-1 loss matrix.

$\Rightarrow \Lambda = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$

By implementing this into python, we obtain the following classification:






```
Minimum Probability of error using the theoretically optimal classifier is: 0.1429
```

The minimum probability of error using the theoretically optimal classifier is: 0.1429

MLP Structure: Use a 2-layer MLP (one hidden layer of perceptrons) that has P perceptrons in the first (hidden) layer with smooth-ramp style activation functions (e.g., ISRU, Smooth-ReLU, ELU, etc). At the second/output layer use a softmax function to ensure all outputs are positive and add up to 1. The best number of perceptrons for your custom problem will be selected using cross-validation.

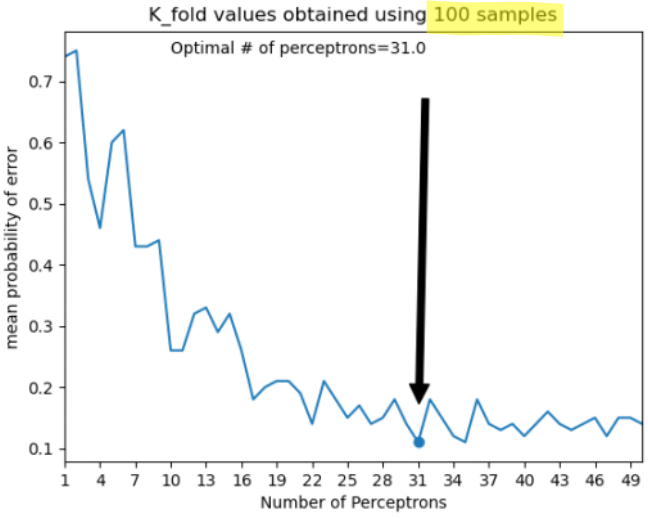
I am using Keras to design and train the 2-layer MLP:

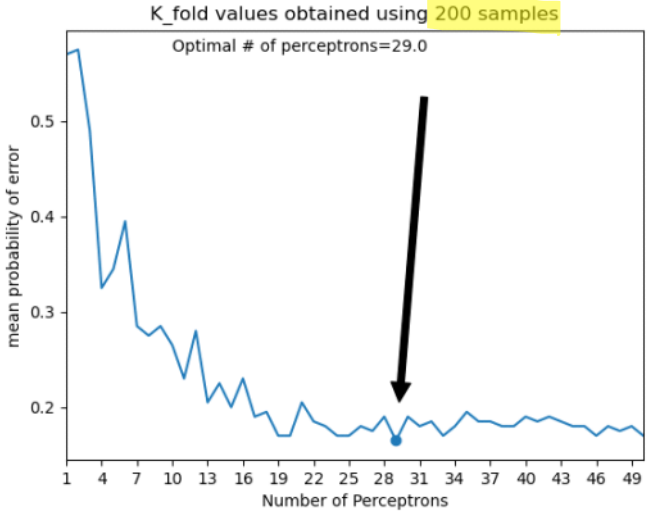
```python
# define the keras model
def get_Keras_model(NODES_0):
    model = Sequential()
    model.add(Dense(NODES_0, input_dim=n, activation='elu'))
    model.add(Dense(C, activation='softmax'))
    model.compile(optimizer='adam',loss=losses.CategoricalCrossentropy(),metrics=['accuracy'])

    return model
```

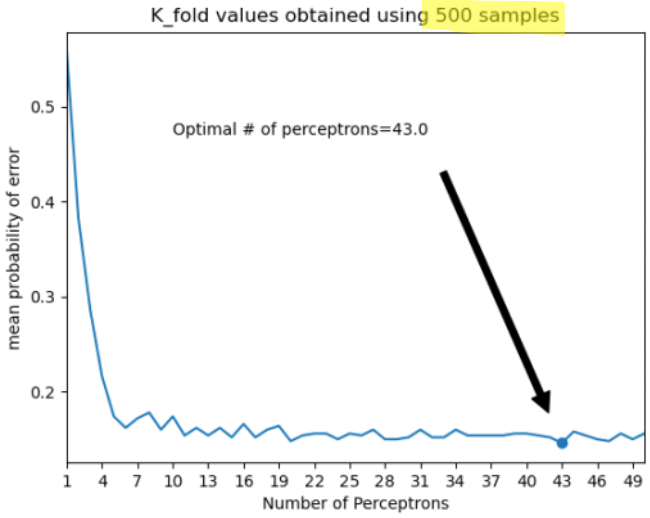
The model above contains 2-Dense layers that are characteristic of MLPs. The variable NODES_0 is the number of perceptrons of the MLP. This variable is looped through values of 1 to 50 in my code to find the optimum number.

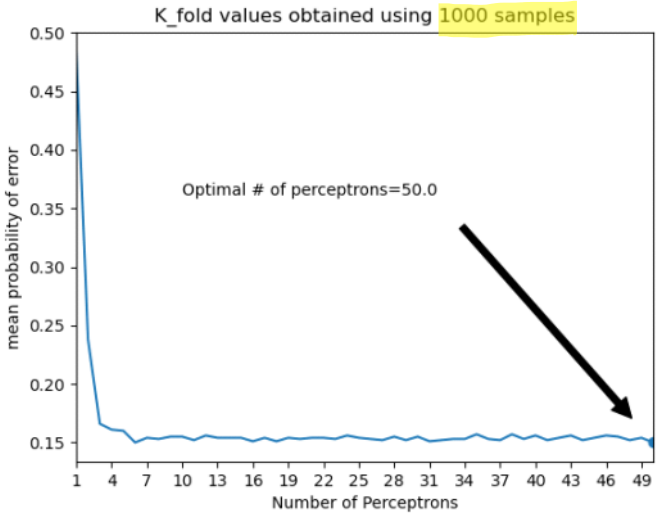
The activation function of the first layer is "elu" and the second layer is a "softmax" function that will provide an output that we derive as class likelihood. We classify each data point as the class with the highest likelihood on the output.

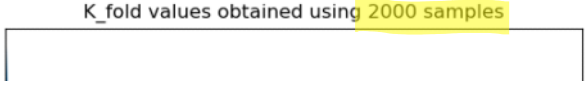
Below I show the mean probability of error obtained by varying the number of perceptrons and training the MLP with different sized datasets as shown:

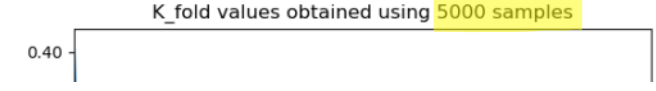


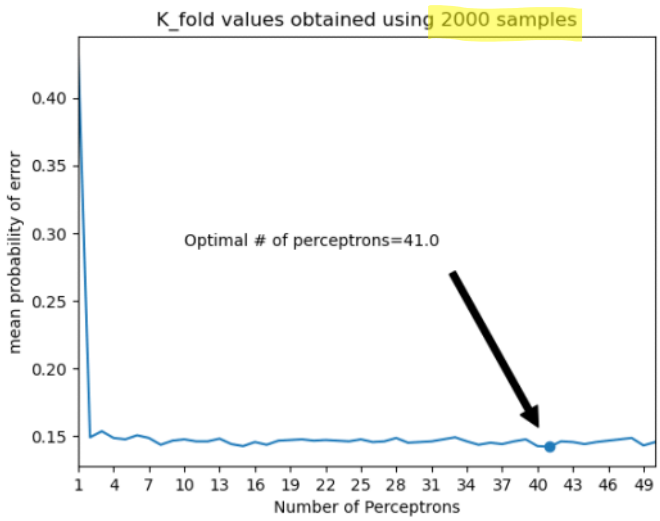


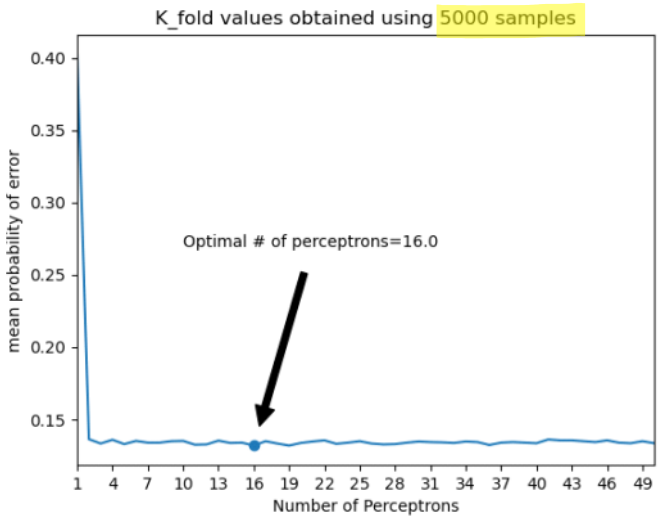






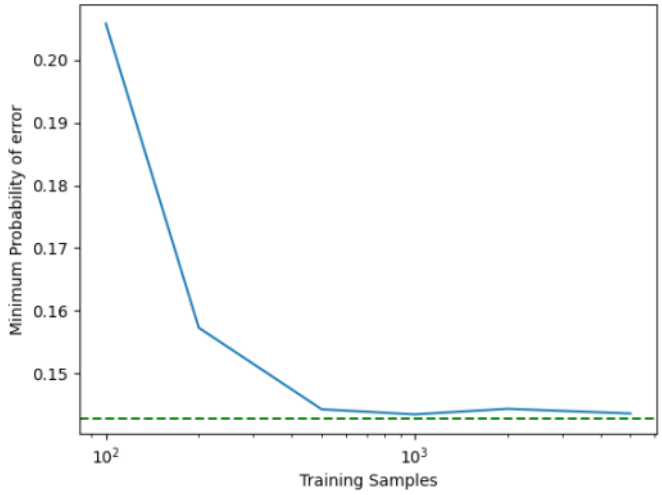






The results above show that for all the sample datasets, we require fewer than 50 perceptrons to achieve an acceptable mean probability of error. This is expected as the dataset we feed into the MLP is not complex and is easily separated into its respective classes.

I will now take the respective optimum number of perceptrons and train an MLP for each dataset and evaluate the trained MLP using the test dataset.
The plot below shows the minimum probability of error achieved by training the MLP with different dataset sample sizes.



The green dashed line is the minimum probability of error using the theoretical optimizer. As the plot shows, The more training samples used to train the MLP, the better it performs on the test data. Interestingly, even with the larger dataset, the MLP is not able to achieve exactly the minimum achievable probability of error.

Question 2 (40%)

We will derive the solution for ridge regression in the case of a linear model with additive white Gaussian noise corrupting both input and output data. Then we will implement it and select the hyper parameter (prior parameter) with cross-validation. Use $n = 7, N_{train} = 100$, and $N_{test} = 10000$ in this question.

Generate Data: Select an arbitrary non-zero n-dimensional vector $\boldsymbol{a}$. Pick an arbitrary Gaussian with nonzero-mean $\boldsymbol{\mu}$ and non-diagonal covariance matrix $\boldsymbol{\Sigma}$ for a n-dimensional random vector $\boldsymbol{x}$. Draw N_{train} iid samples of n-dimensional samples of $\boldsymbol{x}$ from this Gaussian pdf. Draw N_{train} iid samples of a n-dimensional random variable $\mathbf{z}$ from a $\mathbf{0}$-mean $\alpha\boldsymbol{I}$-covariance-matrix Gaussian pdf. Draw N_{train} iid samples of a scalar random variable v from a 0-mean unit-variance Gaussian pdf. Calculate N_{train} scalar values of a new random variable as follows $y = \boldsymbol{a}^T(\boldsymbol{x}+\boldsymbol{z})+v$ using the samples of $\boldsymbol{x}$ and v. This is your training dataset that consists of $(\boldsymbol{x}, y)$ pairs. Similarly, generate a separate test dataset that consists of N_{test} $(\boldsymbol{x}, y)$ pairs.

We need to pick: a ($n\times 1$), μ ($n\times 1$), Σ ($n\times n$), and α (scalar)

and draw to data sets from:

$$x \sim N(\mu, \Sigma)$$
$$z \sim N(0, \alpha I)$$
$$v \sim N(0, 1) \quad \text{(scalar)}$$

we can produce y as follows:

$$y = a^T(x+z) + v$$

And so our dataset is: $D = \{(x^1, y^1) \dots (x^N, y^N)\}$

for our model we will assume $y = w_x^T x + w_0 + v$ → so we know $v \sim N(0,1)$

but we are "missing" the z vector that is included in the true model

to make it easier to code, we will rearrange:

$$y = \underbrace{[w_0 \; w_x^T]}_{w^T} \begin{bmatrix} 1 \\ x \end{bmatrix} + v \qquad w = \begin{bmatrix} w_0 \\ w_x \end{bmatrix}$$

$$\underbrace{w \sim N(0, \beta I)}_{\text{prior}}$$

Now we want to do MAP parameter estimation:

Objective: $\hat{w}_{MAP} = \underset{w}{\text{argmax}} \; \ln p(w \mid D)$

Bayes Rule ⇒ $\hat{w}_{MAP} = \underset{w}{\text{argmax}} \; \underbrace{\ln p(D \mid w)}_{\text{log likelihood}} + \overbrace{\ln p(w)}^{\text{prior}}$

assuming iid samples ⇒ $\underset{w}{\text{argmax}} \; \ln p(w) + \sum_{i=1}^{N} p(x_i, y_i \mid w)$

⇒ $\underset{w}{\text{argmax}} \; \ln p(w) + \sum_{i=1}^{N} \ln p(y_i \mid x_i, w) + \sum_{i=1}^{N} \ln p(x_i \mid w)$ → 0

$$\Rightarrow \underset{w}{\text{argmax}} \quad \ln p(w) + \sum_{i=1}^{N} \ln p(y_i \mid x_i, w) + \sum_{i=1}^{N} \ln p(x_i \mid w) \to 0$$

if we correctly assume that $x \perp\!\!\!\perp w$ (i.e., x is independent of w) then this term goes to 0.

Objective: $\hat{w}_{MAP} = \underset{w}{\text{argmax}} \ln p(w) + \sum_{i=1}^{N} \ln p(y_i \mid x_i, w)$

based on the assumed model, this will be a gaussian

$\ln p(w)$: gaussian distribution

$\ln p(y_i \mid x_i, w)$: gaussian distribution

- $p(w) = (2\pi)^{\frac{n+1}{2}} |\beta I|^{-\frac{1}{2}} e^{-\frac{w^T w}{2\beta}}$ (determinant)

- $p(y_i \mid x_i, w)$: let $m_i = w^T \begin{bmatrix} 1 \\ x_i \end{bmatrix}$, $y \sim \mathcal{N}(m, 1)$

  $p(y_i \mid x_i, w) = (2\pi)^{-\frac{1}{2}} e^{-\frac{(y-m)^2}{2}}$

- $\ln p(y_i \mid x_i, w) = \ln (2\pi)^{-\frac{1}{2}} - \frac{1}{2} (y_i - m_i)^2$

- $\ln p(w) = \ln \left[(2\pi)^{-\frac{n+1}{2}} |\beta I|^{-\frac{1}{2}} \right] - \frac{1}{2\beta} w^T w$

Let's rewrite the important equations:

1. $m_i = w^T \begin{bmatrix} 1 \\ x_i \end{bmatrix}$
2. $y_i = m_i + v$
3. $\ln p(w) = \ln \left((2\pi)^{-\frac{n+1}{2}} |\beta I|^{-\frac{1}{2}} \right) - \frac{1}{2\beta} w^T w$
4. $\ln p(y_i \mid x_i, w) = \ln \left((2\pi)^{-\frac{1}{2}} \right) - \frac{1}{2} (y_i - m_i)^2$
5. Objective: $\hat{w}_{MAP} = \underset{w}{\text{argmax}} \ln p(w) + \sum_{i=1}^{N} \ln p(y_i \mid x_i, w)$

<u>Hyper-parameter optimization:</u>

We know the prior, $\ln p(w)$, is a function of β.
Therefore Eq^n 5, is also a function of β.

Let's find the analytical solution of $\hat{w}_{MAP}$:

$$\hat{w}_{MAP} = \frac{\partial (\text{Eqn } 5)}{\partial w} = 0$$

$$\hat{w}_{MAP} = -\frac{w}{\beta} - \frac{1}{2}\sum_{i=1}^{N} \frac{\partial (y_i - w^T x)(y_i - w^T x)}{\partial w}$$

$$(y_i - w^T x)(y_i - w^T x) = y_i^2 - 2 y_i w^T x + w^T x x^T w$$

$$\left(\frac{\partial}{\partial w}\right) \Rightarrow -2 y_i x + 2 x x^T w$$

$$\hat{w}_{MAP}: \quad -\frac{\hat{w}}{\beta} - \frac{1}{2}\sum_{i=1}^{N} (-2 y_i x + 2 x x^T \hat{w}) = 0$$

$$-\frac{\hat{w}}{\beta} + \sum_{i=1}^{N} y_i x - \sum_{i=1}^{N} x x^T \hat{w} = 0$$

$$\left(\sum_{i=1}^{N} y_i x\right) = \left(\frac{1}{\beta} + \sum_{i=1}^{N} x x^T\right) \hat{w}$$

- $$\boxed{\hat{w} = \left(\frac{1}{\beta} + \sum_{i=1}^{N} x_i x_i^T\right)^{-1} \left(\sum_{i=1}^{N} y_i x_i\right)} \rightarrow \text{Eqn } 6$$

We want to use max-log-likelihood to determine the optimum β value:

log-likelihood = Eqn 4: $\ln p(y_i | x_i, w) = \ln\left((2\pi)^{-\frac{1}{2}}\right) - \frac{1}{2}(y_i - m_i)^2$

$\ln\left((2\pi)^{-\frac{1}{2}}\right) = 0$

we will drop this term since it is not a function of w or β

- we want to find the maximum of $-\frac{1}{2}(y_i - m_i)^2$

$$= -\frac{1}{2}\left(y_i - \hat{w}^T \begin{bmatrix} 1 \\ x_i \end{bmatrix}\right)^2$$

y_i: given; $\hat{w}^T$: calculate using Eqn 6; $\begin{bmatrix} 1 \\ x_i \end{bmatrix}$: given

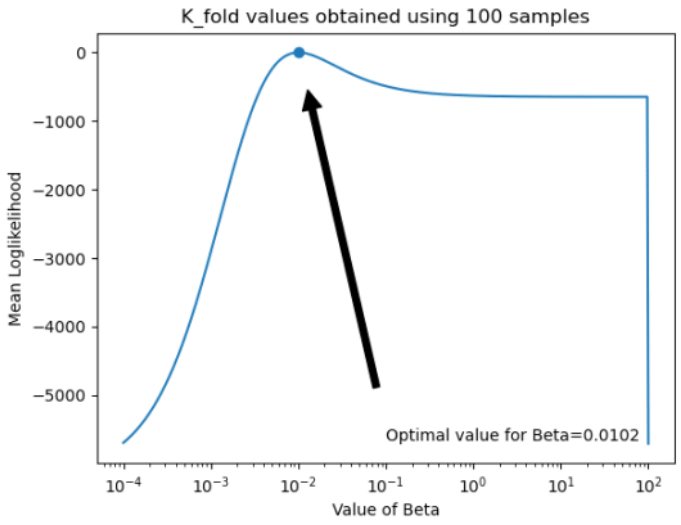
So now we need to type up the code and run.

```
86          #max-Log-Likelihood calculation
87  ŵ_MAP  ←w_hat=1/(1/Beta[M]+np.sum(X_Train_kfold@np.transpose(X_Train_kfold),axis=0))*np.sum(y_Train_kfold*X_Train_kfold,axis=1)
88          LogLikelihood=-1/2*(np.sum(y_Train_kfold)-np.sum(np.transpose(w_hat)@X_Train_kfold) )**2
89          Sum_LogLikelihood[M-1]= Sum_LogLikelihood[M-1]+np.sum(LogLikelihood)
90
91  Mean_Loglikelihood=Sum_LogLikelihood/K_fold
```

```
91  Mean_Loglikelihood=Sum_LogLikelihood/K_fold
```

↳ Here I calculate the average loglikelihood across the 10 folds.

↳ I vary β from 10^{-4} to 10^{2} with 500 data points in between to find the optimum as shown below:



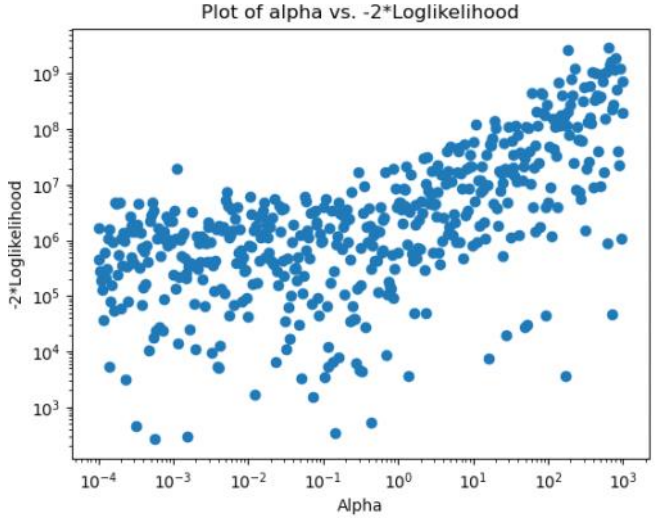
Having run the code, we now have an optimum β value for this dataset.

Model Optimization:

Now I will run the MAP with the optimum β on the full training dataset and estimate the optimum $\hat{w}_{MAP}$.

With this $\hat{w}_{MAP}$, I will evaluate the -2 times log likelihood of the **test data**.

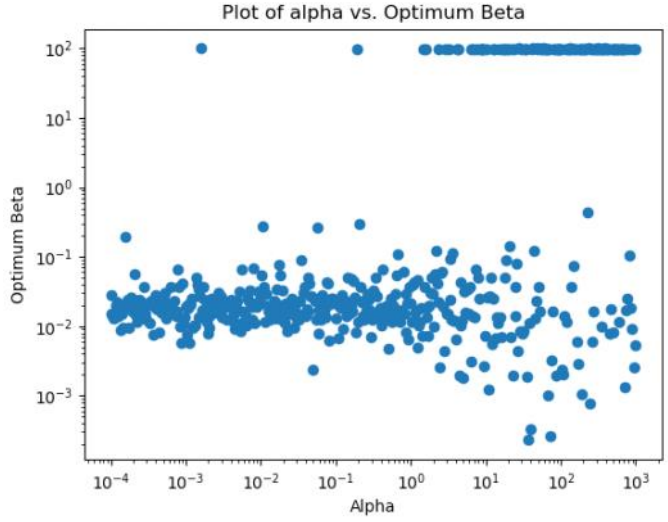
I will vary α from 10^{-4} to 10^{3} with 500 data points in between, and plot α vs -2 loglikelihood:



→ note that x and y are set to logarithmic scale

The figure above shows that as alpha increases, the -2 log likelihood of the data increases.
Since alpha affects the covariance of the additive input noise, this result shows that a noisier data makes it harder to accurately estimate the true underlying model of the dataset. Thus, the log likelihood of the test data decreases as input noise increases.

Similarly, we can look at the relation between α and β:



→ note that x and y are set to logarithmic scale

This plot shows that as alpha increases, the optimum β decreases.
β is an inverse weight multiplied by prior. Therefore, it is no suprise that as the input gets noisier, the more the algorithm has to rely on the prior to perform better. This is achieved by decreasing β which in turn increases the weight of the prior.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Nov 16 08:11:26 2020

@author: Mahdi
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from sklearn.preprocessing import LabelBinarizer

#####################################################################
#                           Question 1                              #
#####################################################################

#------ Generate Data------#
C=4 # Number of Classes
p=[0.25, 0.25, 0.25, 0.25] #Uniform Class Priors
n=3  #Number of dimensions
N_Training=[100, 200, 500, 1000, 2000, 5000] # Training Datset sample sizes
N_Testing=100000 # Testing Dataset sample size

## Class means and covariance ##
Sigma=np.array([[0.5,       0.12,       0.1],
                [  0.1,     0.5,       0],
                [  0,       0.05,     0.5]]);
Mu=np.zeros((C,n));
Mu[0,:]=np.array([-1,  1, 0]);
Mu[1,:]=np.array([-1, -1, 0]);
Mu[2,:]=np.array([ 1, -1, 0]);
Mu[3,:]=np.array([ 1,  1, 0]);
##----------------------------##
Best_M_Kfold=np.zeros(len(N_Training))
P_Error_Optimum=np.zeros(len(N_Training))
Fig_Num=0;
for Loop in range(len(N_Training)):
    ## Generate training samples based on class conditional pdfs ##
    N_Train=N_Training[Loop]
    N_random_numbers=np.random.random(N_Train);
    label=np.zeros(N_Train);
    for i in range(0,N_Train):
        if N_random_numbers[i]<=p[0]:
            label[i]=1;
        elif N_random_numbers[i]<=(p[0]+p[1]):
                label[i]=2;
        elif N_random_numbers[i]<=(p[0]+p[1]+p[2]):
                label[i]=3;
        else:
            label[i]=4;
    Sum_CL=np.zeros(C)
    for i in range(0,C):
        Sum_CL[i]=sum(label==(i+1)); #Number of data points with the given class label

    X = np.zeros((n, N_Train))
    L_Counter=np.zeros(C);
```

```
Data_Set_L=np.zeros((C,N_Train,n))
for i in range(0,C):
    temp_Mu=np.zeros(n)
    temp_Mu[:]=Mu[i,:]
    Data_Set_L[i,0:int(Sum_CL[i]),:]= np.random.multivariate_normal(temp_Mu,Sigma,int(Sum_CL[i]

for i in range(0,N_Train):
    for CLASS in range(0,C):
        if label[i]==CLASS+1:
            X[:,i]= np.transpose(Data_Set_L[CLASS,int(L_Counter[CLASS]),:]);
            L_Counter[CLASS]=L_Counter[CLASS]+1;


# Fig_Num=Fig_Num+1
# fig=plt.figure(Fig_Num)
# ax = fig.add_subplot(111, projection='3d')

# m=['o','s','*','d']

# ax.scatter(Data_Set_L[0,:,0], Data_Set_L[0,:,1], Data_Set_L[0,:,2], c='#8c564b', marker=m[0],
# ax.scatter(Data_Set_L[1,:,0], Data_Set_L[1,:,1], Data_Set_L[1,:,2], c='#2ca02c', marker=m[1],
# ax.scatter(Data_Set_L[2,:,0], Data_Set_L[2,:,1], Data_Set_L[2,:,2], c='#17becf', marker=m[2],
# ax.scatter(Data_Set_L[3,:,0], Data_Set_L[3,:,1], Data_Set_L[3,:,2], c='#e377c2', marker=m[3],
# ax.legend(['Class 1', 'Class 2', 'Class 3', 'Class 4'])
# ax.set_xlabel("X1")
# ax.set_ylabel("X2")
# ax.set_zlabel("X3")
# ax.set_title("Training Dataset Size: "+str(N_Train))


N_random_numbers=np.random.random(N_Testing);
label_Testing=np.zeros(N_Testing);
for i in range(0,N_Testing):
    if N_random_numbers[i]<=p[0]:
        label_Testing[i]=1;
    elif N_random_numbers[i]<=(p[0]+p[1]):
            label_Testing[i]=2;
    elif N_random_numbers[i]<=(p[0]+p[1]+p[2]):
            label_Testing[i]=3;
    else:
        label_Testing[i]=4;
Sum_CL_Testing=np.zeros(C)
for i in range(0,C):
    Sum_CL_Testing[i]=sum(label_Testing==(i+1)); #Number of data points with the given class la

X_Testing = np.zeros((n, N_Testing))
L_Counter_Testing=np.zeros(C);

Data_Set_L_Testing=np.zeros((C,N_Testing,n))
for i in range(0,C):
    temp_Mu=np.zeros(n)
    temp_Mu[:]=Mu[i,:]
    Data_Set_L_Testing[i,0:int(Sum_CL_Testing[i]),:]= np.random.multivariate_normal(temp_Mu,Sig

for i in range(0,N_Testing):
    for CLASS in range(0,C):
        if label_Testing[i]==CLASS+1:
```

```
            X_Testing[:,i]= np.transpose(Data_Set_L_Testing[CLASS,int(L_Counter_Testing[CLASS])
            L_Counter_Testing[CLASS]=L_Counter_Testing[CLASS]+1;

# Fig_Num=Fig_Num+1

# fig=plt.figure(Fig_Num)
# ax = fig.add_subplot(111, projection='3d')

# m=['o','s','*','d']

# ax.scatter(Data_Set_L_Testing[0,:,0], Data_Set_L_Testing[0,:,1], Data_Set_L_Testing[0,:,2], c
# ax.scatter(Data_Set_L_Testing[1,:,0], Data_Set_L_Testing[1,:,1], Data_Set_L_Testing[1,:,2], c
# ax.scatter(Data_Set_L_Testing[2,:,0], Data_Set_L_Testing[2,:,1], Data_Set_L_Testing[2,:,2], c
# ax.scatter(Data_Set_L_Testing[3,:,0], Data_Set_L_Testing[3,:,1], Data_Set_L_Testing[3,:,2], c
# ax.legend(['Class 1', 'Class 2', 'Class 3', 'Class 4'])
# ax.set_xlabel("X1")
# ax.set_ylabel("X2")
# ax.set_zlabel("X3")
# ax.set_title("Testing Dataset size: "+str(N_Testing))
#--- END Generate Data-----#

#--- Theoretically Optimal Classifier ------#

# #Define Loss Matrix
# lossMatrix= np.ones(C)-np.identity(C);

# #---- Calculate Expected Risk and decide on the minimum one ----#
# def evalGaussian(x, Mu, Sigma):
#     g=multivariate_normal.pdf(x,Mu,Sigma);
#     return g

# pxgivenL=np.zeros((C,N_Testing))
# Class_Posterior=np.zeros((C,N_Testing))
# Expected_Risk=np.zeros((C,N_Testing))
# Decision=np.zeros(N_Testing)
# for i in range(0,C):
#     for j in range(0,N_Testing):
#         pxgivenL[i,j] = evalGaussian(X_Testing[:,j],Mu[i,:],Sigma[:,:]); #Evaluate p(x|L=i)
#     Class_Posterior[i,:]=pxgivenL[i,:]*p[i]; #This is not really class posterior, we must div

# for j in range(0,N_Testing):
#     Expected_Risk[:,j]=lossMatrix@Class_Posterior[:,j];
#     Decision[j]= np.argmin(Expected_Risk[:,j])+1;
# #- END Calculate Expected Risk --#

# ##Calculate Confusion Matrix
# Confusion_Matrix=np.zeros((C,C));
# Sum_DL=np.zeros((C,C));
# for D in range(0,C):
#         for L in range(0,C):
#             Sum_DL[D,L]=sum(np.logical_and(Decision==D+1, label_Testing==L+1)); #Number of da
#             Confusion_Matrix[D,L]=Sum_DL[D,L]/Sum_CL_Testing[L]; #L-1 since python indexing s

# print("Q1 - The Confusion Matrix is: "+str(Confusion_Matrix))

# Color=[]
# COLOR_L1=[]
```

```
# COLOR_L2=[]
# COLOR_L3=[]
# COLOR_L4=[]

# Indeces_L1=np.zeros(int(Sum_CL_Testing[0]),dtype=int); #Contains the index numbers of Dataset
# Indeces_L2=np.zeros(int(Sum_CL_Testing[1]),dtype=int);
# Indeces_L3=np.zeros(int(Sum_CL_Testing[2]),dtype=int);
# Indeces_L4=np.zeros(int(Sum_CL_Testing[3]),dtype=int);
# L_Counter=np.zeros(C,dtype=int);
# for i in range(0,N_Testing):
#     if Decision[i]==label_Testing[i]:
#         Color.append('green');
#     else:
#         Color.append('red');
#     if label_Testing[i]==1:
#         Indeces_L1[L_Counter[0]]=i;
#         L_Counter[0]=L_Counter[0]+1;
#         COLOR_L1.append(Color[i]);
#     elif label_Testing[i]==2:
#         Indeces_L2[L_Counter[1]]=i;
#         L_Counter[1]=L_Counter[1]+1;
#         COLOR_L2.append(Color[i]);
#     elif label_Testing[i]==3:
#         Indeces_L3[L_Counter[2]]=i;
#         L_Counter[2]=L_Counter[2]+1;
#         COLOR_L3.append(Color[i]);
#     else:
#         Indeces_L4[L_Counter[3]]=i;
#         L_Counter[3]=L_Counter[3]+1;
#         COLOR_L4.append(Color[i]);

# Fig_Num=Fig_Num+1;
# plt.figure(Fig_Num)
# plt.subplot(311)
# plt.scatter(X_Testing[0,Indeces_L1],X_Testing[1,Indeces_L1],facecolors='none', edgecolors=COL
# plt.scatter(X_Testing[0,Indeces_L2],X_Testing[1,Indeces_L2],facecolors='none', edgecolors=COL
# plt.scatter(X_Testing[0,Indeces_L3],X_Testing[1,Indeces_L3],c=COLOR_L3,marker=m[2], s=25, alp
# plt.scatter(X_Testing[0,Indeces_L4],X_Testing[1,Indeces_L4],facecolors='none', edgecolors=COL
# plt.title('Q1 - Green = Correct | Red = Incorrect - Classification of all 4 classes')
# plt.xlabel('X1',fontsize=12)
# plt.ylabel('X2',fontsize=12)
# plt.axis('equal')

# plt.subplot(312)
# plt.scatter(X_Testing[0,Indeces_L1],X_Testing[2,Indeces_L1],facecolors='none', edgecolors=COL
# plt.scatter(X_Testing[0,Indeces_L2],X_Testing[2,Indeces_L2],facecolors='none', edgecolors=COL
# plt.scatter(X_Testing[0,Indeces_L3],X_Testing[2,Indeces_L3],c=COLOR_L3,marker=m[2], s=25, alp
# plt.scatter(X_Testing[0,Indeces_L4],X_Testing[2,Indeces_L4],facecolors='none', edgecolors=COL
# plt.xlabel('X1',fontsize=12)
# plt.ylabel('X3',fontsize=12)
# plt.axis('equal')

# plt.subplot(313)
# plt.scatter(X_Testing[1,Indeces_L1],X_Testing[2,Indeces_L1],facecolors='none', edgecolors=COL
# plt.scatter(X_Testing[1,Indeces_L2],X_Testing[2,Indeces_L2],facecolors='none', edgecolors=COL
# plt.scatter(X_Testing[1,Indeces_L3],X_Testing[2,Indeces_L3],c=COLOR_L3,marker=m[2], s=25, alp
# plt.scatter(X_Testing[1,Indeces_L4],X_Testing[2,Indeces_L4],facecolors='none', edgecolors=COL
```

```
# plt.xlabel('X2',fontsize=12)
# plt.ylabel('X3',fontsize=12)
# plt.axis('equal')

# Fig_Num=Fig_Num+1;
# plt.figure(Fig_Num)
# plt.subplots_adjust(hspace=1.2, wspace=0.8)
# for i in range(0,C):
#     if i==0:
#         SUBPLOT=1;
#         Scat_Color=COLOR_L1
#         Scat_Indeces=Indeces_L1
#         Marker=m[0]
#     elif i==1:
#         SUBPLOT=4;
#         Scat_Color=COLOR_L2
#         Scat_Indeces=Indeces_L2
#         Marker=m[1]
#     elif i==2:
#         SUBPLOT=7;
#         Scat_Color=COLOR_L3
#         Scat_Indeces=Indeces_L3
#         Marker=m[2]
#     else:
#         SUBPLOT=10;
#         Scat_Color=COLOR_L4
#         Scat_Indeces=Indeces_L4
#         Marker=m[3]

#     plt.subplot(4,3,SUBPLOT)
#     plt.scatter(X_Testing[0,Scat_Indeces],X_Testing[1,Scat_Indeces],facecolors='none', edgecc

#     plt.xlabel('X1',fontsize=12)
#     plt.ylabel('X2',fontsize=12)
#     plt.axis('equal')

#     plt.subplot(4,3,SUBPLOT+1)
#     plt.title("Q1 - Green = Correct | Red = Incorrect - Classification for Class Label= "+str
#     plt.scatter(X_Testing[0,Scat_Indeces],X_Testing[2,Scat_Indeces],facecolors='none', edgecc
#     plt.xlabel('X1',fontsize=12)
#     plt.ylabel('X3',fontsize=12)
#     plt.axis('equal')

#     plt.subplot(4,3,SUBPLOT+2)
#     plt.scatter(X_Testing[1,Scat_Indeces],X_Testing[2,Scat_Indeces],facecolors='none', edgecc
#     plt.xlabel('X2',fontsize=12)
#     plt.ylabel('X3',fontsize=12)
#     plt.axis('equal')

# ##Probability of Error
# P_Error = (len([i for i, v in enumerate(Color) if v == 'red']))/N_Testing;
# print("Minimum Probability of error using the theoretically optimal classifier is: "+str(P_Er
#--- END Theoretically Optimal Classifier -----#
```

```
#----- Use TensorFlow to Train 2-layer MLP ------#
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import losses

## Changing labels to one-hot encoded vector
lb = LabelBinarizer()
y_train = lb.fit_transform(label)
y_test = lb.transform(label_Testing)
print('Train labels dimension:');print(y_train.shape)
print('Test labels dimension:');print(y_test.shape)

X_train=np.transpose(X)
X_test=np.transpose(X_Testing)

print('Train dimension:');print(X_train.shape)
print('Test dimension:');print(X_test.shape)

# define the keras model
def get_Keras_model(NODES_0):
    model = Sequential()
    model.add(Dense(NODES_0, input_dim=n, activation='elu'))
    model.add(Dense(C, activation='softmax'))
    model.compile(optimizer='adam',loss=losses.CategoricalCrossentropy(),metrics=['accuracy'])

    return model

#--- K-fold CV of MLP ----#
K_fold=10
N_Perceptrons=50
## Divide dataset into K_fold sub-datasets
N_perDataset=int(N_Train/K_fold)
Begin_Index=0
End_Index=N_perDataset
X_train_partitioned=np.zeros((K_fold,N_perDataset,n))
y_train_partitioned=np.zeros((K_fold,N_perDataset,C))
for ITER in range(0,K_fold):
    X_train_partitioned[ITER,:,:]=X_train[Begin_Index:End_Index,:]
    y_train_partitioned[ITER,:,:]=y_train[Begin_Index:End_Index,:]
    Begin_Index=End_Index
    End_Index=End_Index+N_perDataset

P_Error=np.zeros(N_Perceptrons)
for M in range(1,N_Perceptrons+1):
    for K in range(0,K_fold):
        y_test_kfold=np.zeros((N_perDataset,C))
        y_train_kfold=np.zeros((N_perDataset*(K_fold-1),C))
        X_test_kfold=np.zeros((N_perDataset,n))
        X_train_kfold=np.zeros((N_perDataset*(K_fold-1),n))

        X_test_kfold[:,:]=X_train_partitioned[K,:,:]
        y_test_kfold[:,:]=y_train_partitioned[K,:,:]
        K_Train=(np.linspace(0,K_fold-1,K_fold))
        K_Train=np.delete(K_Train,K)
        Begin_Index=0
        End_Index=N_perDataset
```

```
            for K_temp in K_Train:
                K_temp=int(K_temp)
                X_train_kfold[Begin_Index:End_Index,:]=X_train_partitioned[K_temp,:,:]
                y_train_kfold[Begin_Index:End_Index,:]=y_train_partitioned[K_temp,:,:]
                Begin_Index=End_Index
                End_Index=End_Index+N_perDataset

            # fit the keras model on the dataset
            model=get_Keras_model(M)
            model.fit(X_train_kfold, y_train_kfold, epochs=15, batch_size=10, verbose=0)

            # evaluate the keras model
            # make class predictions with the model
            # predictions = model.predict_classes(X_test_kfold)
            # for i in range (0,N_perDataset):
            #     if
            # P_Error[M-1]=P_Error[M-1]+(abs(np.sum(y_test_kfold[:,0])-np.sum(predictions==1))+abs(
            _, acc = model.evaluate(X_test_kfold, y_test_kfold)
            P_Error[M-1]=P_Error[M-1]+(1-acc)
        print("Done with Loop: "+str(Loop+1)+" Perceptrons: "+str(M))

    Mean_P_Error=P_Error/K_fold
    Best_M_Kfold[Loop]=np.argmin(Mean_P_Error)+1
    #---Plot the Kfold value for 1 Experiment with M from 1 to 20
    Range_M=np.linspace(1,M,M)
    Fig_Num=Fig_Num+1
    plt.figure(Fig_Num)
    plt.plot(Range_M,Mean_P_Error)
    plt.scatter(Best_M_Kfold[Loop],Mean_P_Error[int(Best_M_Kfold[Loop])-1])
    TEXT= "Optimal # of perceptrons="+str(Best_M_Kfold[Loop])
    plt.annotate(TEXT, xy=(Best_M_Kfold[Loop], Mean_P_Error[int(Best_M_Kfold[Loop])-1]), xytext=(10
    plt.xlim((1,M))
    plt.xlabel("Number of Perceptrons")
    plt.xticks(np.arange(1, M+1, step=3))
    plt.ylabel("mean probability of error")
    plt.title("K_fold values obtained using "+str(N_Train)+" samples")
    #----END K-fold For GMM----#

    #-----Performance Assessment of MLP ----#
    # fit the keras model on the test dataset
    print("Running Test Data for Loop: "+str(Loop+1))
    model=get_Keras_model(Best_M_Kfold[Loop])
    model.fit(X_train, y_train, epochs=15, batch_size=10, verbose=0)
    # evaluate the keras model
    _, acc = model.evaluate(X_test, y_test)
    P_Error_Optimum[Loop]=1-acc

    print("Done with Loop: "+str(Loop+1))

Fig_Num=Fig_Num+1
plt.figure(Fig_Num)
plt.plot(N_Training,P_Error_Optimum)
plt.axhline(0.1429,color='green', linestyle='dashed')
plt.xlabel("Training Samples")
plt.ylabel("Minimum Probability of error")
plt.xscale("log")
#---- END Performance Assessment of MLP ---#
```

```
# # make class predictions with the model
# predictions = model.predict_classes(X_test)
# # summarize the first 5 cases
# for i in range(50):
#     print("NN Output: "+str(predictions[i]+1)+" | Correct Label: "+str(label_Testing[i]))
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Nov 17 20:27:48 2020

@author: mahdiar
"""
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

#------ Generate Data------#
n=7  #Number of dimensions
N_Train=100 # Training Datset sample sizes
N_Test=10000 # Testing Dataset sample size

## Model Parameters ##
Arbit_vals=(np.random.random(n*n+7))/2
Sigma=np.zeros((n,n))
ROW=0
for i in range(0,n*n,7):
    Sigma[ROW,0:7]=Arbit_vals[i:i+7]
    ROW =ROW+1
Mu=np.array([Arbit_vals[49],  Arbit_vals[50], Arbit_vals[51], Arbit_vals[52], Arbit_vals[53], Arbit
N_Alpha=500
alpha=np.logspace(-4,3,N_Alpha)
Best_Beta=np.zeros(N_Alpha)
Neg2LogLikelihood=np.zeros(N_Alpha)

for Alpha_Index in range(0,N_Alpha):
    a= np.random.random(n)

    v_Train=np.random.normal(0,1,N_Train)
    z_Train=np.transpose(np.random.multivariate_normal(np.zeros(n),np.identity(n)*alpha[Alpha_Index
    v_Test=np.random.normal(0,1,N_Test)
    z_Test=np.transpose(np.random.multivariate_normal(np.zeros(n),np.identity(n)*alpha[Alpha_Index]

    ## Generate training samples ##
    x_Train= np.transpose(np.random.multivariate_normal(Mu,Sigma,N_Train))
    x_Test= np.transpose(np.random.multivariate_normal(Mu,Sigma,N_Test))
    X_Train=np.vstack((np.ones(N_Train),x_Train))
    X_Test=np.vstack((np.ones(N_Test),x_Test))
    ## Calculate y based on generated samples ##
    y_Train=np.transpose(a)@(x_Train+z_Train)+v_Train
    y_Test=np.transpose(a)@(x_Test+z_Test)+v_Test
    #------ END Generate Data----#


    #---K-fold ---#
    K_fold=10

    ## Divide dataset into K_fold sub-datasets
    N_perDataset=int(N_Train/K_fold)
    Begin_Index=0
    End_Index=N_perDataset
    X_partitioned=np.zeros((K_fold,n+1,N_perDataset))
    y_partitioned=np.zeros((K_fold,N_perDataset))
```

```
    for ITER in range(0,K_fold):
        X_partitioned[ITER,:,:]=X_Train[:,Begin_Index:End_Index]
        y_partitioned[ITER,:]=y_Train[Begin_Index:End_Index]
        Begin_Index=End_Index
        End_Index=End_Index+N_perDataset

    N_Params=500 #Number of Beta values to test
    Beta=np.logspace(-4,2,N_Params)
    Sum_LogLikelihood=np.zeros(N_Params)
    for M in range(0,N_Params):
        for K in range(0,K_fold):
            X_Test_kfold=np.zeros((n+1,N_perDataset))
            X_Train_kfold=np.zeros((n+1,N_perDataset*(K_fold-1)))
            y_Test_kfold=np.zeros((N_perDataset))
            y_Train_kfold=np.zeros((N_perDataset*(K_fold-1)))
            X_Test_kfold[:,:]=X_partitioned[K,:,:]
            y_Test_kfold[:]=y_partitioned[K,:]
            K_Train=(np.linspace(0,K_fold-1,K_fold))
            K_Train=np.delete(K_Train,K)
            Begin_Index=0
            End_Index=N_perDataset
            for K_temp in K_Train:
                K_temp=int(K_temp)
                X_Train_kfold[:,Begin_Index:End_Index]=X_partitioned[K_temp,:,:]
                y_Train_kfold[Begin_Index:End_Index]=y_partitioned[K_temp,:]
                Begin_Index=End_Index
                End_Index=End_Index+N_perDataset

            #max-log-likelihood calculation
            w_hat=1/(1/Beta[M]+np.sum(X_Train_kfold@np.transpose(X_Train_kfold),axis=0))*np.sum(y_T
            LogLikelihood=-1/2*(np.sum(y_Train_kfold)-np.sum(np.transpose(w_hat)@X_Train_kfold) )**
            Sum_LogLikelihood[M-1]= Sum_LogLikelihood[M-1]+np.sum(LogLikelihood)

    Mean_Loglikelihood=Sum_LogLikelihood/K_fold
    Best_M_Kfold=np.argmax(Mean_Loglikelihood)
    Best_Beta[Alpha_Index]=Beta[Best_M_Kfold]
    Fig_Num=1
    plt.figure(Fig_Num)
    plt.plot(Beta,Mean_Loglikelihood)
    plt.scatter(Best_Beta[Alpha_Index],Mean_Loglikelihood[int(Best_M_Kfold)])
    TEXT= "Optimal value for Beta="+str(round(Best_Beta[Alpha_Index],4))
    plt.annotate(TEXT, xy=(Best_Beta[Alpha_Index], Mean_Loglikelihood[int(Best_M_Kfold)]), xytext=(
    # plt.xlim((1,M))
    plt.xlabel("Value of Beta")
    # plt.xticks(np.arange(1, M+1, step=1))
    plt.ylabel("Mean Loglikelihood")
    plt.title("K_fold values obtained using "+str(N_Train)+" samples")
    plt.xscale("log")
    #---END K-fold ---#

    #------ Model Optimization -----#
    w_hat_Optimum=1/(1/Best_Beta[Alpha_Index]+np.sum(X_Train@np.transpose(X_Train),axis=0))*np.sum(
    Neg2LogLikelihood[Alpha_Index]=(np.sum(y_Test)-np.sum(np.transpose(w_hat_Optimum)@X_Test))**2

Fig_Num=Fig_Num+1
plt.figure(Fig_Num)
plt.scatter(alpha,Neg2LogLikelihood)
```

```
plt.title("Plot of alpha vs. -2*Loglikelihood")
plt.ylabel("-2*Loglikelihood")
plt.xlabel("Alpha")
plt.xscale("log")
plt.yscale("log")

Fig_Num=Fig_Num+1
plt.figure(Fig_Num)
plt.scatter(alpha,Best_Beta)
plt.title("Plot of alpha vs. Optimum Beta")
plt.ylabel("Optimum Beta")
plt.xlabel("Alpha")
plt.xscale("log")
plt.yscale("log")
```