ECE C143A Homework 3

Lawrence Liu

May 14, 2022

Problem 1

(a)

We have

$$\begin{aligned} L &= \Pi_{i=0} P(\mathbf{x}_i, t_i) \\ &= \Pi_{i=0} P(t_i) P(\mathbf{x}_i | t_i) \end{aligned}$$

let $P(t_i = j) = P(C_j)$, therefore we have

$$\begin{aligned} L &= \Pi_{i \in C_1} P(C_1) P(\mathbf{x}_i | C_1) \ldots \Pi_{i \in C_k} P(C_k) P(\mathbf{x}_i | C_k) \\ \log(L) &= \sum_{i=1}^{k} N_i \log(P(c_i)) + \sum_{j \in C_1} \log(P(\mathbf{x}_j | C_1)) + \cdots + \sum_{j \in C_k} \log(P(\mathbf{x}_j | C_k)) \end{aligned}$$

We have

$$P(\mathbf{x}_j | C_k) = (2\pi \det(\Sigma_k))^{-\frac{1}{2}} \exp(-\frac{1}{2}(\mathbf{x}_j - \mu_k)^T \Sigma_k^{-1} (\mathbf{x}_j - \mu_k))$$

$$\log(P(\mathbf{x}_j | C_k)) = -\frac{1}{2} \log(2\pi \det(\Sigma_k)) - \frac{1}{2}(\mathbf{x}_j - \mu_k)^T \Sigma_k^{-1} (\mathbf{x}_j - \mu_k)$$

In order to find the maximum likelihood estimator μ_k we find the values of μ_k such that $\frac{\partial \log L}{\partial \mu_k} = 0$, Therefore we have

$$\frac{\partial \log L}{\partial \mu_k} = \sum_{j \in C_k} -\frac{1}{2}\frac{\partial}{\partial \mu_k}(\mathbf{x}_j - \mu_k)^T \Sigma_k^{-1}(\mathbf{x}_j - \mu_k) \quad = -\frac{1}{2}\sum_{j \in C_k}(\Sigma_k + \Sigma_k^T)(\mathbf{x}_k - \mu_k)$$
$$= -\frac{1}{2}(\Sigma_k + \Sigma_k^T)\sum_{j \in C_k} \mathbf{x}_k - \mu_k$$

therefore we have that in order for $\frac{\partial \log L}{\partial \mu_k} = 0$,

$$\sum_{j \in C_k} \mathbf{x}_k - \mu_k = 0$$

$$\boxed{\mu_k = \frac{1}{N_k}\sum_{j \in C_k} \mathbf{x}_k}$$

Likewise, the maximum likelihood estimator of Σ_k is the values such that $\frac{\partial \log L}{\partial \Sigma_k} = 0$, this is equivalent to finding the values of Σ_k^{-1} such that $\frac{\partial \log L}{\partial \Sigma_k^{-1}} = 0$

$$\frac{\partial \log L}{\partial \Sigma_k^{-1}} = 0$$
$$\frac{1}{2}\frac{\partial N_k \log(2\pi \det(\Sigma_k)) + \sum_{i \in k}(\mathbf{x}_j - \mu_k)^T \Sigma_k^{-1}(\mathbf{x}_j - \mu_k)}{\partial \Sigma_k^{-1}} = 0$$

We have

$$\begin{aligned}
\frac{\partial \log(2\pi \det(\Sigma_k))}{\partial \Sigma_k^{-1}} &= \frac{\partial \log(\det(\Sigma_k))}{\partial \Sigma_k^{-1}} \\
&= \frac{1}{\det(\Sigma_k)}\frac{\partial \det(\Sigma_k)}{\partial \Sigma_k^{-1}} \\
&= \frac{1}{\det(\Sigma_k)}(-\det(\Sigma_k))\Sigma_k^T \\
&= -\Sigma_k^T
\end{aligned}$$

Furthermore we have

$$\frac{\partial (\mathbf{x}_j - \mu_k)^T \Sigma_k^{-1}(\mathbf{x}_j - \mu_k)}{\partial \Sigma_k^{-1}} = (\mathbf{x}_j - \mu_k)^T(\mathbf{x}_j - \mu_k)$$

Therefore we have

$$\begin{aligned}
0 = \frac{\partial \log L}{\partial \Sigma_k^{-1}} &= -N_k \Sigma_k^T + \sum_{i \in C_k} (\mathbf{x}_j - \mu_k)^T (\mathbf{x}_j - \mu_k) \\
N_k \Sigma_k^T &= \sum_{i \in C_k} (\mathbf{x}_j - \mu_k)^T (\mathbf{x}_j - \mu_k) \\
\Sigma_k &= \boxed{\frac{1}{N_k} \sum_{i \in C_k} (\mathbf{x}_j - \mu_k)(\mathbf{x}_j - \mu_k)^T}
\end{aligned}$$

And to find the maximum likelihood estimator for $P(C_i)$ we must find the value of $P(C_i)$ for any $i \in k$ such that $\sum_{j \in k} N_j \log(p(C_j))$ is maximized, since $\sum_{i \in K} p(C_i) = 1$, we have $\sum_{j \in k} N_j \log(p(C_j)) = \sum_{j \in k} N_j \log(p(C_j)) + \lambda(\sum_{i \in K} p(C_j) - 1)$ Therefore taking the derivative of each with respect to $P(C_i)$ for any $i \in k$ we get

$$\begin{aligned}
\frac{\partial}{\partial P(C_1)} \sum_{j \in k} N_j p(C_j) + \lambda(\sum_{i \in K} p(C_i) - 1) &= \frac{N_j}{C_j} + \lambda = 0 \\
&\vdots \\
\frac{\partial}{\partial P(C_k)} \sum_{j \in k} N_j p(C_j) + \lambda(\sum_{i \in K} p(C_i) - 1) &= \frac{N_k}{C_k} + \lambda = 0 \\
\frac{\partial}{\partial \lambda} \sum_{j \in k} N_j p(C_j) + \lambda(\sum_{i \in K} p(C_i) - 1) &= \sum_{i \in K} p(C_i) = 1
\end{aligned}$$

Solving these we get that $p(C_k) = \boxed{\frac{N_k}{N}}$

(b)

We have

$$\begin{aligned}
L &= \Pi_{i=0} P(\mathbf{x}_i, t_i) \\
&= \Pi_{i=0} P(t_i) P(\mathbf{x}_i | t_i)
\end{aligned}$$

let $P(t_i = j) = P(C_j)$, therefore we have

$$\begin{aligned} L &= \Pi_{i\in C_1} P(C_1)P(\mathbf{x}_i|C_1) \dots \Pi_{i\in C_k} P(C_k)P(\mathbf{x}_i|C_k) \\ \log(L) &= \sum_{i=1}^{k} N_i \log(P(c_i)) + \sum_{j\in C_1} \log(P(\mathbf{x}_j|C_1)) + \dots + \sum_{j\in C_k} \log(P(\mathbf{x}_j|C_k)) \end{aligned}$$

We have

$$P(\mathbf{x}_j|C_k) = \Pi_{i=1}^{D} P(x_i|C_k)$$

where

$$P(x_i|C_k) = \text{Poisson}(\lambda_{ik}) = \frac{\lambda_{ik}^{x_i} e^{-\lambda_{ik}}}{x_i!}$$

Therefore we have

$$\begin{aligned} log(P(\mathbf{x}_j|C_k)) &= \sum_{i=1}^{D} \log\left(\frac{\lambda_{ik}^{x_i} e^{-\lambda_{ik}}}{x_i!}\right) \\ &= \sum_{i=1}^{D} (x_i \log(\lambda_{ik}) - \lambda_{ik} - \log(x_i!)) \end{aligned}$$

the maximum likelihood estimator of λ_{ik} is the values such that $\frac{\partial \log L}{\partial \lambda_{ik}} = 0$, we have

$$\begin{aligned} \frac{\partial}{\partial \lambda_{ik}} \log(L) &= \frac{\partial}{\partial \lambda_{ik}} \sum_{i=1}^{k} N_i \log(P(c_i)) + \frac{\partial}{\partial \lambda_{ik}} \sum_{j\in C_1} \log(P(\mathbf{x}_j|C_1)) + \dots + \frac{\partial}{\partial \lambda_{ik}} \sum_{j\in C_k} \log(P(\mathbf{x}_j|C_k)) \\ &= \frac{\partial}{\partial \lambda_{ik}} \sum_{j\in C_k} \log(P(\mathbf{x}_j|C_k)) \end{aligned}$$

$$\begin{aligned} \frac{\partial}{\partial \lambda_{ik}} \log(P(\mathbf{x}_j|C_k)) &= \frac{\partial}{\partial \lambda_{ik}} (x_i \log(\lambda_{ik}) - \lambda_{ik} - \log(x_i!)) \\ &= \frac{x_i}{\lambda_{ik}} - 1 \end{aligned}$$

Therefore we have

$$\begin{aligned}
0 &= \frac{\partial}{\partial \lambda_{ik}} \log(L) \\
0 &= \sum_{j \in C_k} \frac{x_j i}{\lambda_{ik}} - 1 \\
N_k \lambda_{ik} &= \sum_{j \in C_k} x_j i \\
\lambda_{ik} &= \boxed{\frac{1}{N_k} \sum_{j \in C_k} x_j i}
\end{aligned}$$

And to find the maximum likelihood estimator for $P(C_i)$ we must find the value of $P(C_i)$ for any $i \in k$ such that $\sum_{j \in k} N_j \log(p(C_j))$ is maximized, since $\sum_{i \in K} p(C_i) = 1$, we have $\sum_{j \in k} N_j \log(p(C_j)) = \sum_{j \in k} N_j \log(p(C_j)) + \lambda(\sum_{i \in K} p(C_j) - 1)$ Therefore taking the derivative of each with respect to $P(C_i)$ for any $i \in k$ we get

$$\begin{aligned}
\frac{\partial}{\partial P(C_1)} \sum_{j \in k} N_j p(C_j) + \lambda(\sum_{i \in K} p(C_i) - 1) &= \frac{N_j}{C_j} + \lambda = 0 \\
&\vdots \\
\frac{\partial}{\partial P(C_k)} \sum_{j \in k} N_j p(C_j) + \lambda(\sum_{i \in K} p(C_i) - 1) &= \frac{N_k}{C_k} + \lambda = 0 \\
\frac{\partial}{\partial \lambda} \sum_{j \in k} N_j p(C_j) + \lambda(\sum_{i \in K} p(C_i) - 1) &= \sum_{i \in K} p(C_i) = 1
\end{aligned}$$

Solving these we get that $p(C_k) = \boxed{\frac{N_k}{N}}$

Problem 2

(a)

We have that the decision boundary is all values of $\mathbf{x}$ such that

$$P(\mathbf{x}, C_1) = P(\mathbf{x}, C_2)$$

Therefore we must have

$$P(C_1)P(\mathbf{x}|C_1) = P(C_1)P(\mathbf{x}|C_2)$$

$$log(P(C_1)) - \frac{1}{2}\log(2\pi\det(\Sigma_1)) - \frac{1}{2}(\mathbf{x}-\mu_1)^T\Sigma_1^{-1}(\mathbf{x}-\mu_1) =$$
$$log(P(C_2)) - \frac{1}{2}\log(2\pi\det(\Sigma_2)) - \frac{1}{2}(\mathbf{x}-\mu_2)^T\Sigma_2^{-1}(\mathbf{x}-\mu_2)$$

Therefore the decision boundary is

$$\boxed{\mathbf{x}^T A\mathbf{x} + B\mathbf{x} + C = 0}$$

where

$$A = \frac{1}{2}(\Sigma_1^{-1} - \Sigma_2^{-1})$$
$$B = \mu_2^T\Sigma_2^{-1} - \mu_1^T\Sigma_1^{-1}$$

$$C = \log(P(C_2)) - \log(P(C_1)) + \frac{1}{2}\log(2\pi\det(\Sigma_1)) - \frac{1}{2}\log(2\pi\det(\Sigma_2)) + \frac{1}{2}\mu_1^T\Sigma_1^{-1}\mu_1 - \frac{1}{2}\mu_2^T\Sigma_2^{-1}\mu_2$$

This is not linear, therefore the linear decision boundary is not linear

(b)

We have that the decision boundary is all values of $\mathbf{x}$ such that

$$P(\mathbf{x}, C_1) = P(\mathbf{x}, C_2)$$

Therefore we must have

$$P(C_1)P(\mathbf{x}|C_1) = P(C_1)P(\mathbf{x}|C_2)$$

$$log(P(C_1)) + \sum_{i=1}^{D}(x_i \log(\lambda_{i1}) - \lambda_{i1} - \log(x_i!)) = log(P(C_2)) + \sum_{i=1}^{D}(x_i \log(\lambda_{i2}) - \lambda_{i2} - \log(x_i!))$$

let $\boldsymbol{\lambda}_1 = \begin{bmatrix}\lambda_{11}\\ \vdots \\ \lambda_{D1}\end{bmatrix}$, and $\boldsymbol{\lambda}_2 = \begin{bmatrix}\lambda_{12}\\ \vdots \\ \lambda_{D2}\end{bmatrix}$, we have

$$\log(P(C_1)) - \sum_{i=1}^{D}\lambda_{i1} + \log(\boldsymbol{\lambda}_1)^T\mathbf{x} = \log(P(C_2)) - \sum_{i=1}^{D}\lambda_{i2} + \log(\boldsymbol{\lambda}_2)^T\mathbf{x}$$

Therefore the decision boundary is,

$$\boxed{\log(P(C_1)) - \sum_{i=1}^{D}\lambda_{i1} - \log(P(C_2)) + \sum_{i=1}^{D}\lambda_{i2} = (\log(\boldsymbol{\lambda}_2)^T - \log(\boldsymbol{\lambda}_1)^T)\mathbf{x}}$$

Therefore there will be a linear decision boundary.

hw4p3

May 14, 2022

0.1 Homework 4, Problem 3 Classification on simulated data

ECE C143A/C243A, Spring Quarter 2022, Prof. J.C. Kao, TAs T. Monsoor, W. Yu

0.2 Background

We will now apply the results of Problems 1 and 2 to simulated data. The dataset can be found on CCLE as ps4_simdata.mat.

The following describes the data format. The .mat file has a single variable named 'trial', which is a structure of dimensions (20 data points) $\times$ (3 classes). The nth data point for the kth class is denoted via:

`data['trial'][n][k][0]` where n = 0,…,19 and k = 0,1,2 are the data points and classes respectively. The `[0]` after `[n][k]` is an artifact of how the `.mat` file is imported into Python. You can get a clearer sense of this below in the plotting scripts.

To make the simulated data as realistic as possible, the data are non-negative integers, so one can think of them as spike counts. With this analogy, there are D = 2 neurons and K = 3 stimulus conditions.

Please follow steps (a)–(e) below for each of the three models. The result of this problem should be three separate plots, one for each model. These plots will be similar in spirit to Figure 4.5 in PRML.

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.special
import scipy.io as sio
import math

# Load matplotlib images inline
%matplotlib inline

# Reloading any code written in external .py files.
%load_ext autoreload
%autoreload 2

data = sio.loadmat('ps4_simdata.mat') # load the .mat file.
NumData = data['trial'].shape[0]
NumClass = data['trial'].shape[1]
```

[2]:
```
data['trial'][0,2][0]
```

[2]:
```
array([[6],
       [7]], dtype=uint8)
```

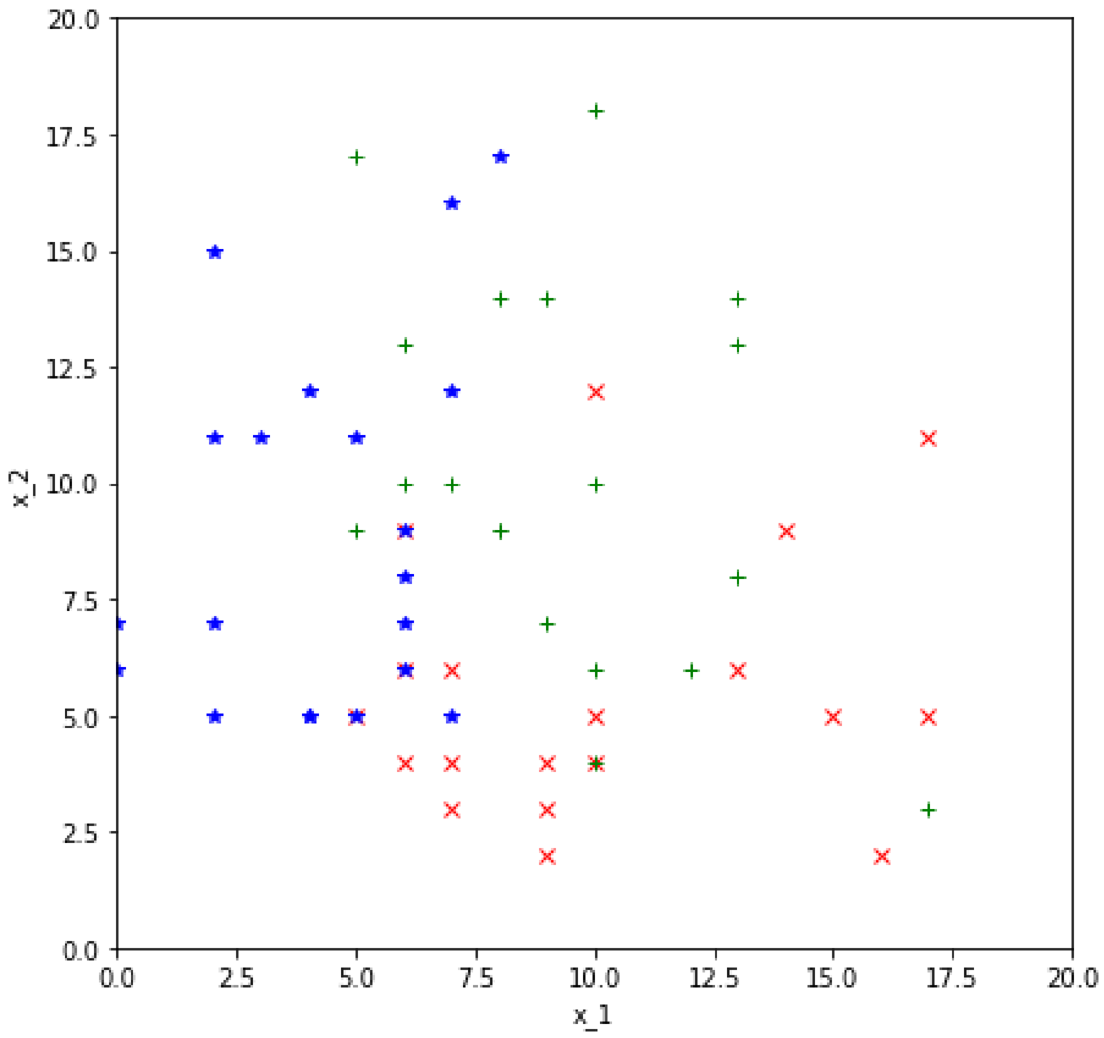
0.2.1 (a) Plot the data points

Here, to get you oriented on the dataset, we'll give you code that plots the data points. You do not have to write any new code here, but you should review this code to understand it, since we'll ask you to make plots in later parts of the notebook.

Here, we plot the data points in a two-dimensional space. For classes k = 1, 2, 3, we use a red $\times$, green $+$, and blue $*$ for each data point, respectively. The axis limits of the plot are between 0 and 20. You should use these axes bounds for the rest of the homework.

[3]:
```
# a
plt.figure(figsize=(7,7))
#=====================================================#
# PLOTTING CODE BELOW
#=====================================================#
dataArr =  np.zeros((NumClass,NumData ,2)) # dataArr contains the points
for classIX in range(NumClass):
    for dataIX in range(NumData):
        x = data['trial'][dataIX,classIX][0][0][0]
        y = data['trial'][dataIX,classIX][0][1][0]
        dataArr[classIX,dataIX,0]=x
        dataArr[classIX,dataIX,1]=y
MarkerPat=np.array(['rx','g+','b*'])

for classIX in range(NumClass):
    for dataIX in range(NumData):
        plt.
 ↪plot(dataArr[classIX,dataIX,0],dataArr[classIX,dataIX,1],MarkerPat[classIX])

#=====================================================#
# END PLOTTING CODE
#=====================================================#
plt.axis([0,20,0,20])
plt.xlabel('x_1')
plt.ylabel('x_2')
```

[3]:
```
Text(0, 0.5, 'x_2')
```



0.2.2 (b) (15 points) Find the ML model parameters

Find the ML model parameters, for each model, using results from Problem 1. Report the values of all the ML parameters for each model. (Please print the names and values of all the ML parameters in Jupyter Notebook)

[4]:
```
#=====================================================#
# YOUR CODE HERE:
#   Find the parameters for each model you derived in problem 1 using
#   the simulated data, and print out the values of each parameter.
#
#   To facilitate plotting later on, we're going to ask you to
#   format the data in the following way.
#
#   (1) Keep three dictionaries, modParam1, modParam2, and modParam3
#       which contain the model parameters for model 1 (Gaussian, shared cov),
#       model 2 (Gaussian, class specific cov), and model 3 (Poisson).
#
#       The Python dictionary is like a MATLAB struct. e.g., you can declare:
#       modParam1 = {} # declares the dictionary
#       modParam1['pi'] = np.array((0.33, 0.33, 0.34)) # sets the field 'pi' to␣
↪be
#         an np.array of size (3,) containing the class probabilities.
#
#   (2) modParam1 has the following structure
#
#     modParam1['pi'] is an np.array of size (3,) containing the class␣
↪probabilities.
#     modParam1['mean'] is an np.array of size (3,2) containing the class means.
#     modParam1['cov'] is an np.array of size (2,2) containing the shared cov.
#
#   (3) modParam2:
#
#     modParam2['pi'] is an np.array of size (3,) containing the class␣
↪probabilities.
#     modParam2['mean'] is an np.array of size (3,2) containing the class means.
#     modParam2['cov'] is an np.array of size (3,2,2) containing the cov for␣
↪each of the 3 classes.
#
#   (4) modParam3:
#     modParam2['pi'] is an np.array of size (3,) containing the class␣
↪probabilities.
#     modParam2['mean'] is an np.array of size (3,2) containing the Poisson␣
↪parameters for each class.
#
#   These should be consistent with the print statement after this code block.
#
#   HINT: the np.mean and np.cov functions ought simplify the code.
#
#=====================================================#
modParam1={'pi':np.zeros(3),'mean':np.zeros((3,2)),'cov':np.zeros((2,2))}
modParam2={'pi':np.zeros(3),'mean':np.zeros((3,2)),'cov':np.zeros((3,2,2))}
modParam3={'pi':np.zeros(3),'mean':np.zeros((3,2))}

for k in range(3):
    for n in range(20):
        modParam1['pi'][k]+=data['trial'][n,k][0].shape[1]
        modParam2['pi'][k]+=data['trial'][n,k][0].shape[1]
        modParam3['pi'][k]+=data['trial'][n,k][0].shape[1]
        modParam1['mean'][k,:]+=data['trial'][n,k][0].flatten()
        modParam2['mean'][k,:]+=data['trial'][n,k][0].flatten()
        modParam3['mean'][k,:]+=data['trial'][n,k][0].flatten()

    modParam1['mean'][k]/=modParam1['pi'][k]
    modParam2['mean'][k]/=modParam2['pi'][k]
    modParam3['mean'][k]/=modParam3['pi'][k]


    for n in range(20):
        modParam1['cov']+=1/60*np.matmul((data['trial'][n,k][0].
 ↪flatten()-modParam1['mean'][k]).reshape((2,-1)),
                                    (data['trial'][n,k][0].
 ↪flatten()-modParam1['mean'][k]).reshape((-1,2)))

        modParam2['cov'][k]+=1/20*np.matmul((data['trial'][n,k][0].
 ↪flatten()-modParam1['mean'][k]).reshape((2,-1)),
                                    (data['trial'][n,k][0].
 ↪flatten()-modParam1['mean'][k]).reshape((-1,2)))

modParam1['pi']/=np.sum(modParam1['pi'])
modParam2['pi']/=np.sum(modParam2['pi'])
modParam3['pi']/=np.sum(modParam3['pi'])
#=====================================================#
# END YOUR CODE
#=====================================================#

# Print out the model parameters
print("Model 1:")
print("Class priors:")
print( modParam1['pi'])
print("Means:")
print( modParam1['mean'])
print("Cov:")
print( modParam1['cov'])

print("model 2:")
print("Class priors:")
print( modParam2['pi'])
print("Means:")
print( modParam2['mean'])
print("Cov1:")
print( modParam2['cov'][0])
print("Cov2:")
print( modParam2['cov'][1])
print("Cov3:")
print( modParam2['cov'][2])

print("model 3:")
```

```python
print("Class priors:")
print( modParam3['pi'])
print("Lambdas:")
print( modParam3['mean'])
```

```
Model 1:
Class priors:
[0.33333333 0.33333333 0.33333333]
Means:
[[10.75  5.55]
 [ 9.6  10.1 ]
 [ 4.3   9.  ]]
Cov:
[[11.97916667 -0.02416667]
 [-0.02416667 12.5125    ]]
model 2:
Class priors:
[0.33333333 0.33333333 0.33333333]
Means:
[[10.75  5.55]
 [ 9.6  10.1 ]
 [ 4.3   9.  ]]
Cov1:
[[20.9875  2.1375]
 [ 2.1375  7.2475]]
Cov2:
[[ 9.54 -4.71]
 [-4.71 15.79]]
Cov3:
[[ 5.41  2.5 ]
 [ 2.5  14.5 ]]
model 3:
Class priors:
[0.33333333 0.33333333 0.33333333]
Lambdas:
[[10.75  5.55]
 [ 9.6  10.1 ]
 [ 4.3   9.  ]]
```

0.2.3 (c) Plot the ML mean

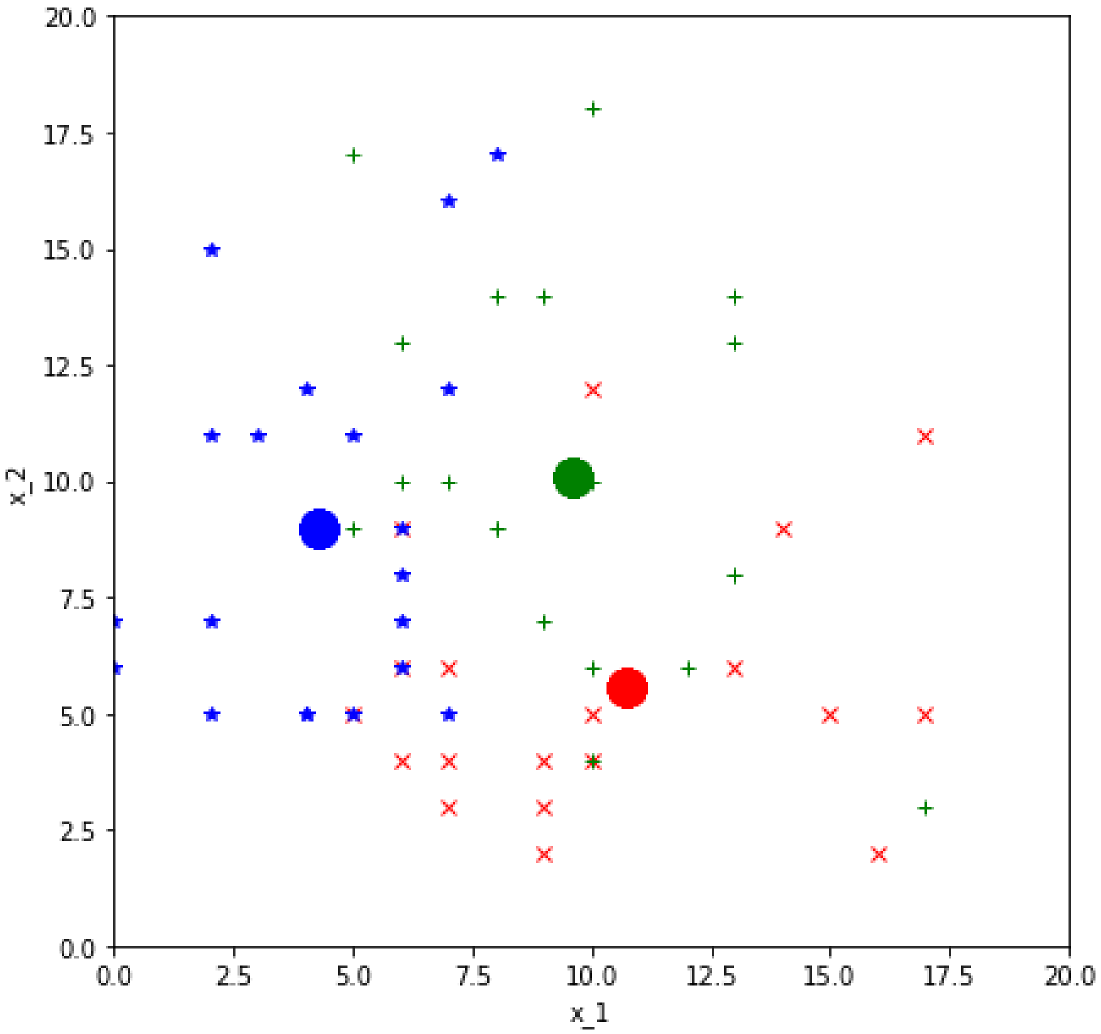
The following code plots the ML mean for each class. You should read the code to understand what is going on. If you followed our instructions on how to format the data, you should not have to modify any code here. This plot needs to be generated for us to check if you implemented the means correctly. You may also use this as a sanity check.

*** If you made modifications in the way the data is formatted, you need to change this code to visualize the ML means***

For each class, we plot the ML mean on top of the data using a solid dot of the appropriate color. We set the marker size of this dot to be much larger than the marker sizes you used in part a, so the dot is easy to see.

```
[5]: # c
plt.figure(figsize=(7,7))
#======================================================#
# ML MEAN PLOT CODE HERE.
#======================================================#
colors = ['r.','g.','b.']
for classIX in range(NumClass):
    for dataIX in range(NumData):
        plt.
 ↪plot(dataArr[classIX,dataIX,0],dataArr[classIX,dataIX,1],MarkerPat[classIX])
        plt.
 ↪plot(modParam1['mean'][classIX,0],modParam1['mean'][classIX,1],colors[classIX],markersize=3
#======================================================#
# END CODE
#======================================================#
plt.axis([0,20,0,20])
plt.xlabel('x_1')
plt.ylabel('x_2')
```

```
[5]: Text(0, 0.5, 'x_2')
```



0.2.4 (d) Plot the ML covariance ellipsoids.

The following code plots the ML covariance for each class. You should read the code to understand what is going on. If you followed our instructions on how to format the data, you should not have to modify any code here. This plot needs to be generated for us to check if you implemented the means correctly. You may also use this as a sanity check.

*** If you made modifications in the way the data is formatted, you need to change this code to visualize the ML covariance ellipsoids***

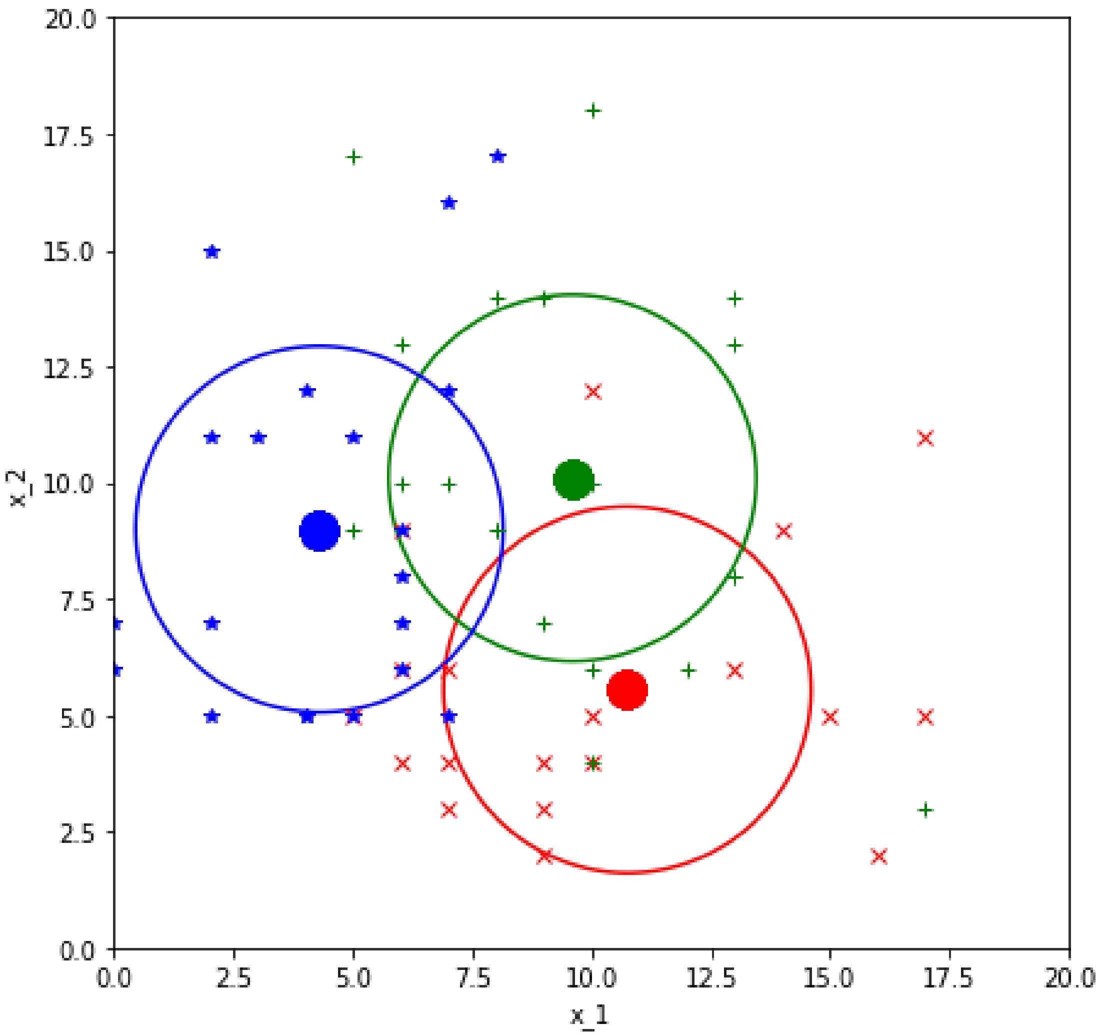
For each class, we plot the ML covariance using an ellipse of the appropriate color. We plot this on top of the data with the means. This part only encapsulates the Gaussian models i) and ii). We generate separate plots for models i) and ii).

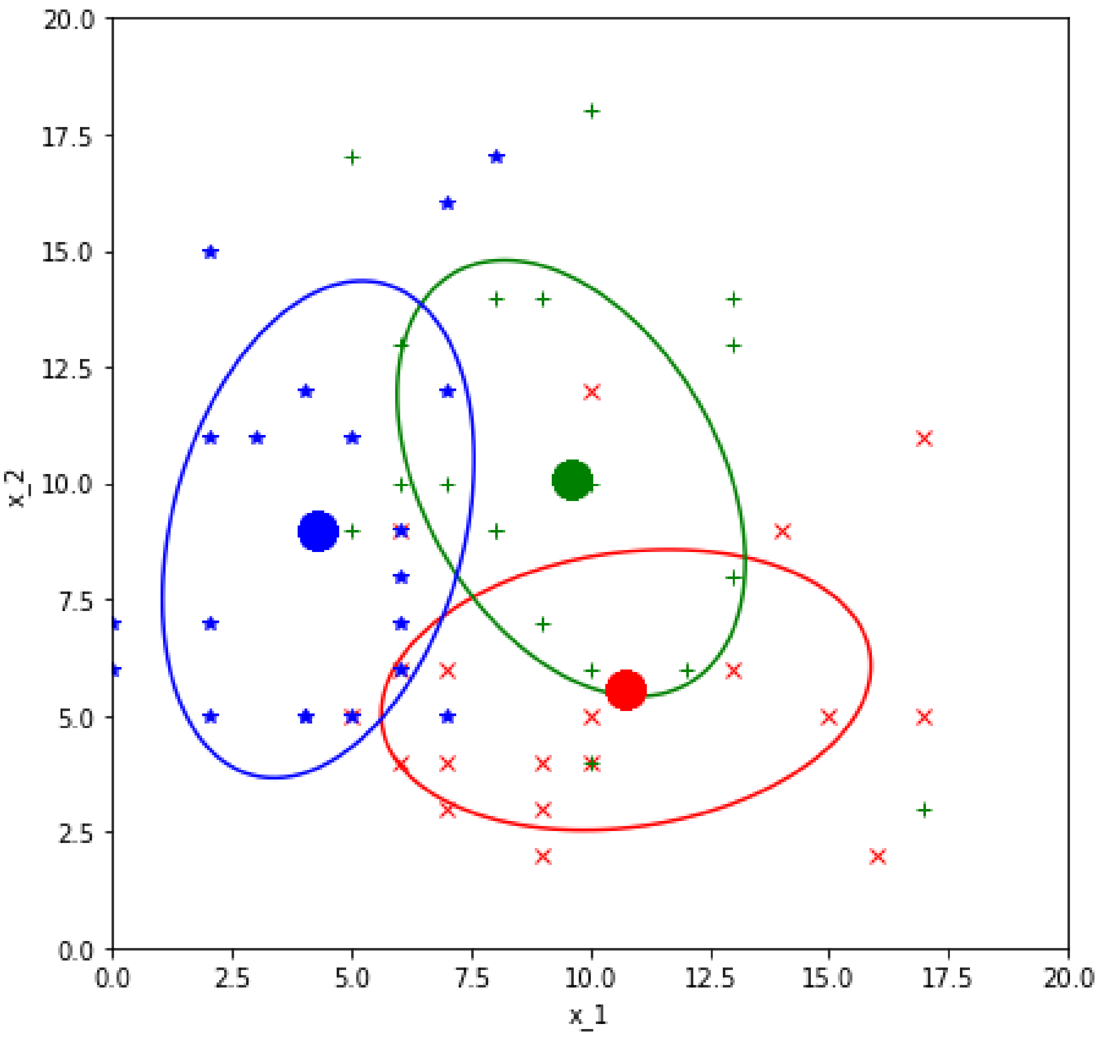
We use of `plt.contour` can be used to draw an iso-probability contour for each class. To aid interpretation, the contour should be drawn at the same probability level for each class. We call `plt.contour(X, Y, Z, levels = level, colors = color)`.

For this specific problem, we choose the contour level so you can see each ellipsoid reasonably, e.g. levels = 0.007, where X and Y are obtained via `[X,Y] = np.meshgrid(np.linspace(0, 20, N), np.linspace(0, 20, N))`, where N is the number of partitions, e.g. N = 20. Z is the function value, Please set the contour color to be the same as data points color.

Please understand this code, as it will facilitate the last part of this notebook where we ask you to generate a plot with classification boundaries. In prior years we asked the students to generate this, but have provided it here to reduce the homework load.

[14]:
```
# d
#=====================================================#
# ML COV PLOT CODE HERE.
#=====================================================#
colors2 = ['r','g','b']
modParam = [modParam1 , modParam2]
for modelIX in range(2):
    plt.figure(modelIX,figsize=(7,7))
    for classIX in range(NumClass):
        for dataIX in range(NumData):
            #plot the points and their means, just like before
            plt.
 ↪plot(dataArr[classIX,dataIX,0],dataArr[classIX,dataIX,1],MarkerPat[classIX])
            plt.
 ↪plot(modParam[modelIX]['mean'][classIX,0],modParam[modelIX]['mean'][classIX,1],colors[class
        plt.axis([0,20,0,20])
        plt.xlabel('x_1')
        plt.ylabel('x_2')
        MarkerCol=['r','g','b']

    #now begins plotting the elipse
    for classIX in range(NumClass):
        currMean=modParam[modelIX]['mean'][classIX ,:]
        if(modelIX == 0):
            currCov=modParam[modelIX]['cov']
        else:
            currCov=modParam[modelIX]['cov'][classIX]
        xl = np.linspace(0, 20, 201)
        yl = np.linspace(0, 20, 201)
        [X,Y] = np.meshgrid(xl,yl)

        Xlong = np.reshape(X-currMean[0],(np.prod(np.size(X))))
        Ylong = np.reshape(Y-currMean[1],(np.prod(np.size(X))))
        temp = np.row_stack([Xlong,Ylong])
        Zlong = []
        for i in range(np.size(Xlong)):
            Zlong.append(np.matmul(np.matmul(temp[:,i], np.linalg.
 ↪inv(currCov)), temp[:,i].T))
        Zlong = np.matrix(Zlong)
        Zlong = np.exp(-Zlong/2)/np.sqrt((2*np.pi)*(2*np.pi)*np.linalg.
↪det(currCov))
        Z = np.reshape(Zlong,X.shape)
        isoThr=0.007
        plt.contour(X,Y,Z,levels = [0,isoThr],colors = colors2[classIX])
#=====================================================#
# END CODE
#=====================================================#
```





0.2.5 (e) (15 points) Plot multi-class decision boundaries

Plot multi-class decision boundaries corresponding to the decision rule

$$\hat{k} = argmax_k \ P(C_k|x) \tag{1}$$

and label each decision region with the appropriate class k. This should be plotted on top of your means and the covariance ellipsoids. Thus, you should start by copying and pasting code from the prior Jupyter Notebook cell.

To plot the multi-class decision boundaries, we recommend that you do it by classifying a dense sampling of the two-dimensional data space.

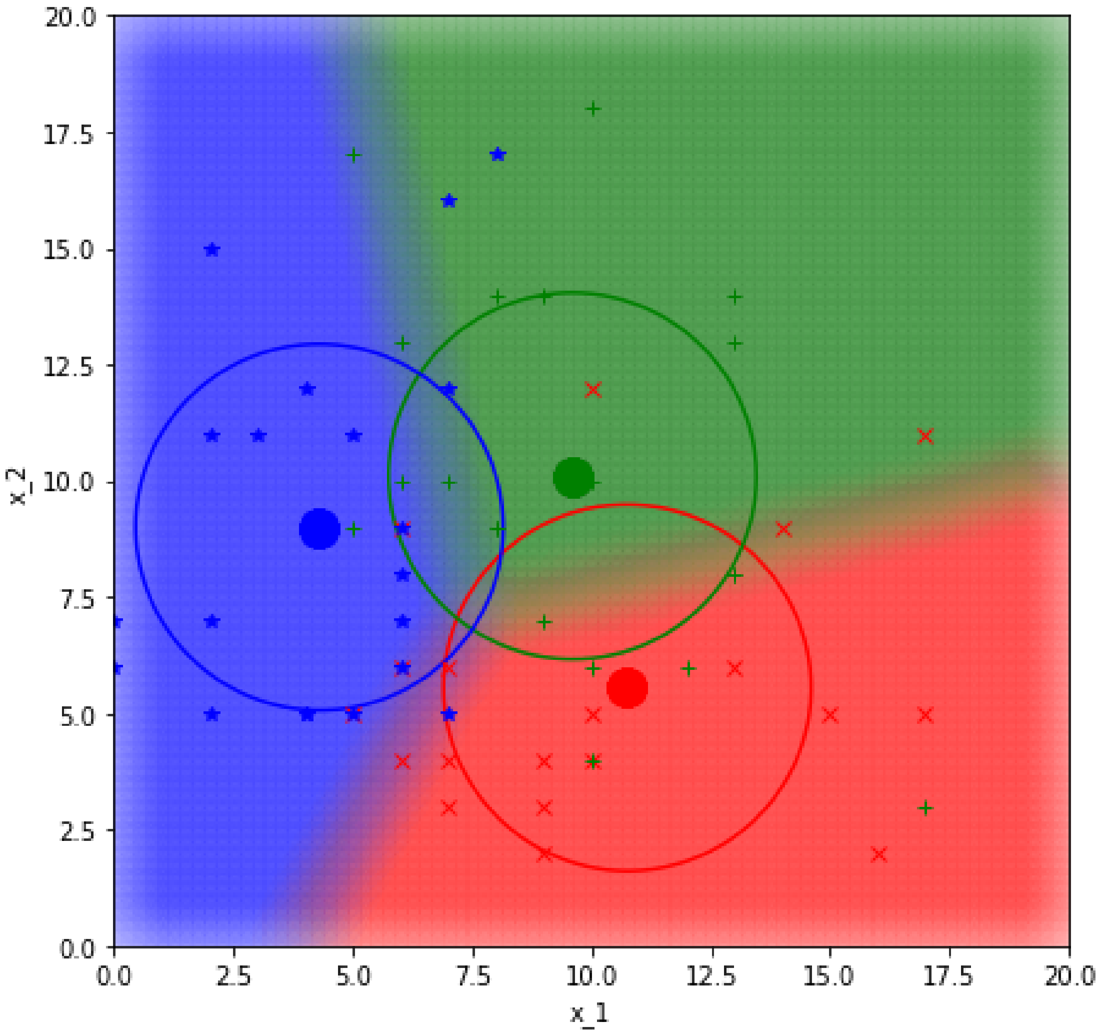
Hint 1: You can do this by calling `[X,Y] = np.meshgrid(np.linspace(0, 20, N), np.linspace(0, 20, N))` to partition the space as done in the previous section, and then classifying each of these points. N should be large; in our solution, we use N = 81. Then at each of these points, draw a dot of the color of the classified class.

Hint 2: You can check that you've done this properly by verifying that the decision boundaries pass through the intersection points of the contours drawn in part (d).

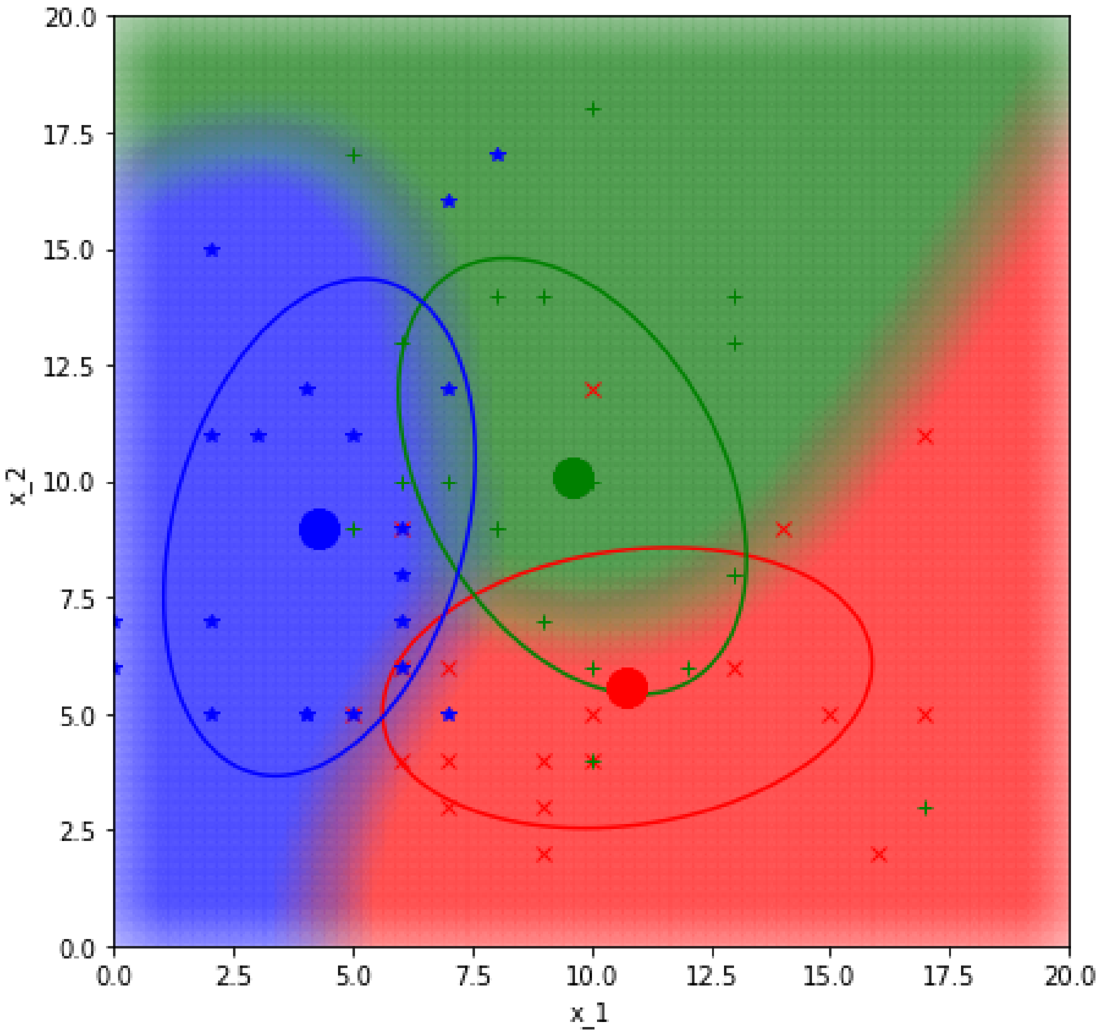
Hint 3: It's a good idea to do this one model at a time. You should get things working for model 1 for a smaller N value, so in code development, it doesn't take a long time to test your code. In the final result, your code will probably take some time to run because you're classifying each data point in a dense grid for each model.

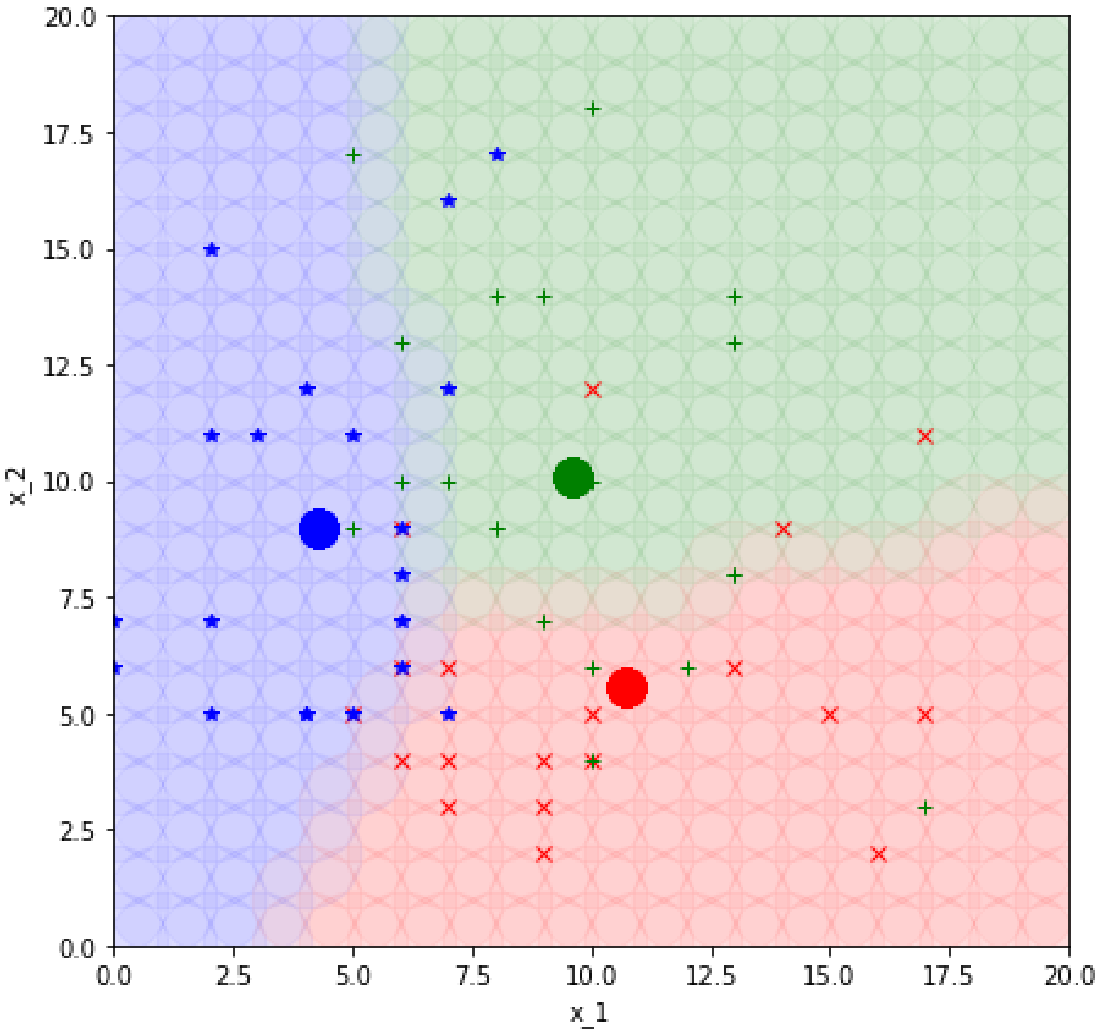
[62]:
```
#e
#=====================================================#
# YOUR CODE HERE:
#   Plot the data points, their means, covariance ellipsoids,
#     and decision boundaries for each model.
#   Note that the naive Bayes Poisson model does not have an ellipsoid.
#   As in the above description, the decision boundary should be achieved
#     by densely classifying points in a grid.
#=====================================================#
from scipy.stats import poisson
from scipy.stats import multivariate_normal

colors2 = ['r','g','b']
modParam = [modParam1 , modParam2,modParam3]
for modelIX in range(3):
    plt.figure(modelIX,figsize=(7,7))
    for classIX in range(NumClass):
        for dataIX in range(NumData):
            #plot the points and their means, just like before
            plt.
 ↪plot(dataArr[classIX,dataIX,0],dataArr[classIX,dataIX,1],MarkerPat[classIX])
            plt.
 ↪plot(modParam[modelIX]['mean'][classIX,0],modParam[modelIX]['mean'][classIX,1],colors[class
        plt.axis([0,20,0,20])
        plt.xlabel('x_1')
        plt.ylabel('x_2')
        MarkerCol=['r','g','b']
    if modelIX<2:
        #now begins plotting the elipse
        for classIX in range(NumClass):
            currMean=modParam[modelIX]['mean'][classIX ,:]
            if(modelIX == 0):
                currCov=modParam[modelIX]['cov']
            else:
                currCov=modParam[modelIX]['cov'][classIX]
            xl = np.linspace(0, 20, 201)
            yl = np.linspace(0, 20, 201)
            [X,Y] = np.meshgrid(xl,yl)

            Xlong = np.reshape(X-currMean[0],(np.prod(np.size(X))))
            Ylong = np.reshape(Y-currMean[1],(np.prod(np.size(X))))
            temp = np.row_stack([Xlong,Ylong])
            Zlong = []
            for i in range(np.size(Xlong)):
                Zlong.append(np.matmul(np.matmul(temp[:,i], np.linalg.
↪inv(currCov)), temp[:,i].T))
            Zlong = np.matrix(Zlong)
            Zlong = np.exp(-Zlong/2)/np.sqrt((2*np.pi)*(2*np.pi)*np.linalg.
↪det(currCov))
            Z = np.reshape(Zlong,X.shape)
            isoThr=0.007
            plt.contour(X,Y,Z,levels = [0,isoThr],colors = colors2[classIX])

    #now plot the decision boundaries
    [X,Y] = np.meshgrid(np.linspace(0, 20, 81), np.linspace(0, 20, 81))
    if modelIX==0:
        ps=[multivariate_normal(modParam[modelIX]['mean'][0],␣
↪modParam[modelIX]['cov']).pdf,
                multivariate_normal(modParam[modelIX]['mean'][1],␣
↪modParam[modelIX]['cov']).pdf,
                multivariate_normal(modParam[modelIX]['mean'][2],␣
↪modParam[modelIX]['cov']).pdf,]
    elif modelIX==1:
        ps=[multivariate_normal(modParam[modelIX]['mean'][0],␣
↪modParam[modelIX]['cov'][0]).pdf,
                multivariate_normal(modParam[modelIX]['mean'][1],␣
↪modParam[modelIX]['cov'][1]).pdf,
                multivariate_normal(modParam[modelIX]['mean'][2],␣
↪modParam[modelIX]['cov'][2]).pdf,]
    elif modelIX==2:
        [X,Y] = np.meshgrid(np.linspace(0, 20,21,dtype=int), np.linspace(0,␣
↪20,21,dtype=int))
        ps=[lambda x: poisson.pmf(x[0],modParam[modelIX]['mean'][0,0])*poisson.
↪pmf(x[1],modParam[modelIX]['mean'][0,1]),
            lambda x: poisson.pmf(x[0],modParam[modelIX]['mean'][1,0])*poisson.
↪pmf(x[1],modParam[modelIX]['mean'][1,1]),
            lambda x: poisson.pmf(x[0],modParam[modelIX]['mean'][2,0])*poisson.
↪pmf(x[1],modParam[modelIX]['mean'][2,1]),]

    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            probablities=[p([X[i,j],Y[i,j]]) for p in ps]
            plt.scatter(X[i,j],Y[i,j],color=MarkerCol[np.
↪argmax(probablities)],alpha=0.05*20/X.shape[0],s=2000)
#==================================================#
# END YOUR CODE
```

```
#=========================================================#
```



20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
x_2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
x_1


x_1
x_2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0



20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
x_2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
x_1

```
[ ]:
```

hw4p4

May 14, 2022

0.1 Homework 4, Problem 4 Classification on real data

ECE C143A/C243A, Spring Quarter 2022, Prof. J.C. Kao, TAs T. Monsoor, W. Yu

0.2 Background

Neural prosthetic systems can be built based on classifying neural activity related to planning. As described in class, this is analogous to mapping patterns of neural activity to keys on a keyboard. In this problem, we will apply the results of Problems 1 and 2 to real neural data. The neural data were recorded using a 100-electrode array in premotor cortex of a macaque monkey1. The dataset can be found on CCLE as `ps4_realdata.mat`.

The following describes the data format. The `.mat` file is loaded into Python as a dictionary with two keys: `train_trial` contains the training data and `test_trial` contains the test data. Each of these contains spike trains recorded simultaneously from 97 neurons while the monkey reached 91 times along each of 8 different reaching angles.

The spike train recorded from the i_{th} neuron on the n_{th} trial of the k_{th} reaching angle is accessed as

`data['train_trial'][n,k][1][i,:]`

where n = 0,...,90, k = 0,...,7, and i = 0, . . . , 96. The [1] in between [n,k] and [i,:] does not mean anything for this assignment and is simply an "artifact" of how the data is structured. A spike train is represented as a sequence of zeros and ones, where time is discretized in 1 ms steps. A zero indicates that the neuron did not spike in the 1 ms bin, whereas a one indicates that the neuron spiked once in the 1 ms bin. The structure test trial has the same format as train trial.

Each spike train is 700 ms long (and thus represented by an array of length 700). This comprises a 200ms baseline period (before the reach target turned on), a 500ms planning period (after the reach target turned on). Because it takes time for information about the reach target to arrive in premotor cortex (due to the time required for action potentials to propagate and for visual processing), we will ignore the first 150ms of the planning period. *** FOR THIS PROBLEM, we will take spike counts for each neuron within a single 200ms bin starting 150ms after the reach target turns on. ***

In other words, to calculate firing rates, you will calculate it over the 200ms window:

`data['train_trial'][n,k][1][i,350:550]`

[1]:
```
import numpy as np
import numpy.matlib as npm
import matplotlib.pyplot as plt
import scipy.special
import scipy.io as sio
import math

data = sio.loadmat('ps4_realdata.mat') # load the .mat file.
NumTrainData = data['train_trial'].shape[0]
NumClass = data['train_trial'].shape[1]
NumTestData = data['test_trial'].shape[0]

# Reloading any code written in external .py files.
%load_ext autoreload
%autoreload 2
```

0.2.1 (a) (8 points)

Fit the ML parameters of model i) to the training data (91 × 8 observations of a length 97 array of neuron firing rates).

To calculate the firing rates, use a single 200ms bin starting from 150ms after the target turns on. This corresponds to using `data['train_trial'][n,k][1][i, 350:550]` to calculate all firing rates. This corresponds to a 200ms window that turns on 150ms after the reach turns on.

Then, use these parameters to classify the test data (91 × 8 data points) according to the decision rule (1). What is the percent of test data points correctly classified?

[19]:
```
##4a

# Calculate the firing rates.

trainDataArr =  np.zeros((NumClass,NumTrainData,97)) # contains the firing␣
↪rates for all neurons on all 8 x 91 trials in the training set
testDataArr =  np.zeros((NumClass,NumTestData,97)) # for the testing set.

for classIX in range(NumClass):
    for trainDataIX in range(NumTrainData):
        trainDataArr[classIX,trainDataIX,:] = np.
↪sum(data['train_trial'][trainDataIX,classIX][1][:,350:550],1)
    for testDataIX in range(NumTestData):
        testDataArr[classIX,testDataIX,:]=np.
↪sum(data['test_trial'][testDataIX,classIX][1][:,350:550],1)
#====================================================#
# YOUR CODE HERE:
#   Fit the ML parameters of model i) to training data
#====================================================#
from scipy.stats import multivariate_normal

means=np.mean(trainDataArr,axis=1)
```

```
cov=np.zeros([97,97])
for i in range(8):
    cov+=np.cov(trainDataArr[i].T)*1/8

#=====================================================#
# END YOUR CODE
#=====================================================#

#=====================================================#
# YOUR CODE HERE:
#   Classify the test data and print the accuracy
#=====================================================#
n_correct=0
n_total=0
for i in range(8):
    predicted=np.empty((8,91))
    for j in range(means.shape[0]):
        predicted[j]=multivariate_normal(means[j],cov).pdf(testDataArr[i])

    n_correct+=np.sum(np.argmax(predicted,axis=0)==i)
    n_total+=91

#=====================================================#
# END YOUR CODE
#=====================================================#
print(f"percent correct={round(100*n_correct/n_total,4)}%")
```

```
percent correct=96.0165%
```

Question: What is the percent of test data points correctly classified?

Your answer: 96.0165%

0.2.2 (b) (6 points)

Repeat part (a) for model ii). You `should encounter a Python error` when classifying the test data. What is this error? Why did the Python error occur? What would we need to do to correct this error?

To be concrete, the output of this cell should be a `Python error` and that's all fine. But we want you to understand what the error is so we can fix it later.

[26]:
```
##4b

#=====================================================#
# YOUR CODE HERE:
# Fit the ML parameters of model ii) to training data
#=====================================================#
```

```
from scipy.stats import multivariate_normal

means=np.mean(trainDataArr,axis=1)
cov=np.zeros([8,97,97])
for i in range(8):
    cov[i]=np.cov(trainDataArr[i].T)*1/8

n_correct=0
n_total=0
for i in range(8):
    predicted=np.empty((8,91))
    for j in range(means.shape[0]):
        #print(cov[j])
        predicted[j]=multivariate_normal(means[j],cov[j]).pdf(testDataArr[i])

    n_correct+=np.sum(np.argmax(predicted,axis=0)==i)
    n_total+=91

print(f"percent correct={round(100*n_correct/n_total,4)}%")
#=====================================================#
# END YOUR CODE
#=====================================================#
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
/tmp/ipykernel_10194/312127973.py in <module>
     18     for j in range(means.shape[0]):
     19         #print(cov[j])
---> 20         predicted[j]=multivariate_normal(means[j],cov[j]).
 ↪pdf(testDataArr[i])
     21 
     22     n_correct+=np.sum(np.argmax(predicted,axis=0)==i)

~/anaconda3/lib/python3.9/site-packages/scipy/stats/_multivariate.py in␣
 ↪__call__(self, mean, cov, allow_singular, seed)
    358         See `multivariate_normal_frozen` for more information.
    359         """
--> 360         return multivariate_normal_frozen(mean, cov,

    361                                           allow_singular=allow_singular
    362                                           seed=seed)

~/anaconda3/lib/python3.9/site-packages/scipy/stats/_multivariate.py in␣
 ↪__init__(self, mean, cov, allow_singular, seed, maxpts, abseps, releps)
    728         self.dim, self.mean, self.cov = self._dist._process_parameters(
    729                                                             None, mean,␣
 ↪cov)
```

```
--> 730         self.cov_info = _PSD(self.cov, allow_singular=allow_singular)
    731         if not maxpts:
    732             maxpts = 1000000 * self.dim

~/anaconda3/lib/python3.9/site-packages/scipy/stats/_multivariate.py in␣
 ↪__init__(self, M, cond, rcond, lower, check_finite, allow_singular)
    163         d = s[s > eps]
    164         if len(d) < len(s) and not allow_singular:
--> 165             raise np.linalg.LinAlgError('singular matrix')
    166         s_pinv = _pinv_1d(s, eps)
    167         U = np.multiply(u, np.sqrt(s_pinv))

LinAlgError: singular matrix
```

Question: Why did the python error occur? What would we need to do to correct this error?

Your answer: This error occurs because the covariance matrix is singular, which happens because certain neurons do not fire at all, which we can confirm running the command `plt.imshow(cov[j]==0)`

0.2.3 (c) (8 points)

Correct the problem from part (b) by detecting and then removing offending neurons that cause the error. Now, what is the percent of test data points correctly classified? Is it higher or lower than your answer to part (a)? Why might this be?

[69]:
```
##4c
neuronsToRemove = []
#====================================================#
# YOUR CODE HERE:
#   Detect and then remove the offending neurons, so that
#   you no longer run into the bug in part (b).
#====================================================#
neuronsToKeep=list(range(97))
for i in range(8):
    for neuron in range(97):
        if all(trainDataArr[i,:,neuron]==0) and neuron not in neuronsToRemove:
            neuronsToRemove.append(neuron)
            neuronsToKeep.remove(neuron)
#====================================================#
# END YOUR CODE
#====================================================#
##
#====================================================#
# YOUR CODE HERE:
# Fit the ML parameters,classify the test data and print the accuracy
#====================================================#
```

```
from scipy.stats import multivariate_normal

trainDataKept=trainDataArr[:,:,neuronsToKeep]
testDataKept=testDataArr[:,:,neuronsToKeep]

means=np.mean(trainDataKept,axis=1)
cov=np.zeros([8,len(neuronsToKeep),len(neuronsToKeep)])
for i in range(8):
    cov[i]=np.cov(trainDataKept[i].T,bias=True)

n_correct=0
n_total=0
for i in range(8):
    predicted=np.empty((8,91))
    for j in range(means.shape[0]):
        #print(cov[j])
        predicted[j]=multivariate_normal(means[j],cov[j]).
 ↪logpdf(testDataKept[i])

    n_correct+=np.sum(np.argmax(predicted,axis=0)==i)
    n_total+=predicted.shape[1]

print(f"percent correct={round(100*n_correct/n_total,4)}%")
#=====================================================#
# END YOUR CODE
#=====================================================#
```

```
percent correct=44.0934%
```

Question: What is the percent of test data points correctly classified? Is it higher or lower than your answer to part (a)? Why might this be?

Your answer: 44.0934%, which is lower than for part (a), this is could because the neurons removed still contains some information, such as that will not fire for certain orientations

0.2.4 (d) (8 points)

Now we classify using a naive Bayes model. Repeat part (a) for model iii). Keep the convention in part (c), where offending neurons were removed from the anal- ysis.

[86]:
```
##4d
#=====================================================#
# YOUR CODE HERE:
# Fit the ML parameters,classify the test data and print the accuracy
#=====================================================#
from scipy.stats import poisson
from scipy.special import factorial
means=np.mean(trainDataKept,axis=1)
```

```
n_correct=0
n_total=0
for i in range(8):
    predicted=np.ones((8,91))
    for j in range(means.shape[0]):
        predicted[j]=np.sum(poisson.logpmf(testDataKept[i],means[j]),axis=1)
        #predicted[j]=np.sum(testDataKept[i]*np.log(means[j])-np.
 ↪log(factorial(testDataKept[i]))-means[j],axis=1)
    n_correct+=np.sum(np.argmax(predicted,axis=0)==i)
    n_total+=predicted.shape[1]

print(f"percent correct={round(100*n_correct/n_total,4)}%")
#=====================================================#
# END YOUR CODE
#=====================================================#
```

```
percent correct=92.033%
```

Question: what is the percent of test data points correctly classified?

Your answer: 92.033%