1.A)

$$\mathcal{L} = \prod_{i=1}^{N} P(t_i)\, P(y_i|t_i)$$

$$y|C_i = N(M_i, \Sigma_i) \qquad i = 1-k$$

Let $P(t_i = j) = \pi_j$. $\quad j = 1-K$

$$\log \mathcal{L} = \log\Big[\prod_{i \in C_1} \{\pi_1 P(y_i|C_1)\}\Big] + \cdots \log\Big[\prod_{i \in C_k} \{\pi_k P(y_i|C_k)\}\Big]$$

$$\log \mathcal{L} = \sum_{i=1}^{k} N_i \log(\pi_i) + \sum_{i \in C_1} \log N(y_i|M_1, \Sigma_1) + \sum_{i \in C_k} \log N(y_i|M_k, \Sigma_k)$$

Compute M_k

$$f(M_k) = \sum_{i \in C_k} \log N(y_i|M_k, \Sigma_k) = \sum_{i \in C_k} \left\{-\frac{1}{2}(y_i - u_k)^T \Sigma_k^{-1} (y_i - M_k)\right\}$$

$$M_k^* = \arg\max_{M_k} f(u_k) = \boxed{\frac{1}{N_k} \sum_{n \in C_k} x_n = M_k}$$

$$\max_{\pi_i} \sum_{i=1}^{k} N_i \log(\pi_i) \quad s.t. \quad \sum_{i=1}^{k} \pi_i = 1$$

$$\max_{\pi_i, \lambda} \sum_{i=1}^{k} N_i \log(\pi_i) + \lambda\left[\sum_{i=1}^{k} \pi_i - 1\right]$$

$$\frac{df(\pi_1, \ldots, \pi_k, \lambda)}{d\pi_1} = \frac{N_1}{\pi_1} + \lambda = 0 \qquad \pi_1 \ldots \pi_k = 1$$

$$\frac{df(\pi_1, \ldots, \pi_k, \lambda)}{d\pi_k} = \frac{N_k}{\pi_k} + \lambda = 0$$

Solve for equation

$$\pi_i = \frac{N_i}{N} \qquad \lambda = -N$$

$$\frac{df(\pi_1, \ldots, \pi_k, \lambda)}{d\lambda} = \sum_{i=1}^{k} \pi_i - 1 = 0$$

$$\boxed{P(C_k) = \frac{N_k}{N}}$$

1a) continued Solve Σ_k

$$f(\Sigma_k) = \sum_{i \in C_k} \left\{ -\frac{1}{2}(y_i - \mu_k)^T \Sigma_k^{-1} (y_i - \mu_k) \right\} + \frac{N_k}{2} \log |\Sigma_k|^{-1}$$

$$\Sigma_k = \arg\max_{\Sigma_k} f(\Sigma_k)$$

$$\frac{d}{d\Sigma_k} f(\Sigma_k) = \sum_{i \in C_k} -\frac{1}{2} \left(y_i^T \Sigma_k^{-1} y_i - y_i^T \Sigma^{-1} \mu_k - \mu_k^T \Sigma_k^{-1} y_i + \mu_k^T \Sigma_k^{-1} \mu_k \right) + \frac{N_k}{2} \Sigma_k$$

$$\frac{d \log L}{d\Sigma} = \frac{1}{2} \sum_{i \in C_k} (y_i - \mu_k)(y_i - \mu_k)^T - \frac{1}{2} N_k \Sigma_k = 0$$

$$\boxed{\sum_{i \in C_k} \frac{N_k}{N} (y_i - \mu_k)(y_i - \mu_k)^T = \Sigma_k}$$

1B) $\mathcal{L}(\theta, x) = P(x|\theta)$

$x = \{(y_1, t_1) \ldots (y_n, t_n)\}$

Assuming Independent observations

$$\prod_{n=1}^{n} P(t_n|\theta) \cdot P(y_n|t_n, \theta)$$

Take log...

$$= \sum_{i=1}^{k} N_i \log(\pi_i) + \sum_{n \in \{t_n=1\}} \log P(y_n|t_n=1) \ldots \sum_{n \in \{t_n=k\}} \log P(y_n|t_n=k)$$

From Poisson model

$$\log P(y_n|t_n=j) = \sum_{i=1}^{D} [y_{ni} \cdot \log(\lambda_{ji}) - \log(\lambda_i!) - \lambda_{ji}]$$

take $f(\lambda_{ji})$ being the terms only dependent on λ_{ji} then

$$f(\lambda_{ji}) = \sum_{n \in (t_n=j)} [y_{ni} \log(\lambda_{ji}) - \lambda_{ji}]$$

$$\frac{df(\lambda_{ji})}{d\lambda_{ji}} = \sum_{n \in (t_n=j)} \left[\frac{y_{ni}}{\lambda_{ji}} - 1\right]$$

set $=0$ and solve

$$\lambda_{ji} = \frac{\sum_{n \in (t_n=j)} y_{ni}}{N_j}$$

where $j = 1 \ldots k$

$i = 1 \ldots D$

Intuitively $P(C_k)$ is the same so...

$$P(C_k) = \frac{N_k}{N}$$

Model 2 Multi Variance

2a) $\log(N(y|M,\Sigma)) = -\frac{1}{2}(y-\mu)^T\Sigma^{-1}(y-M) - \frac{1}{2}\log|\Sigma| - \frac{D}{2}\log(k\pi)$

$-\frac{1}{2}(y-M_1)^T\Sigma_1^{-1}(y-y_1) - \frac{1}{2}\log|\Sigma_1| - \frac{D}{2}\log(2\pi) = -\frac{1}{2}(y-M_2)\Sigma_2^{-1}(y-M_2) - \frac{1}{2}\log|\Sigma_2|$

$-\frac{D}{2}\log(2\pi)$

$-\frac{1}{2}(y-M_1)^T\Sigma_1^{-1}(y-M_1) + b = -\frac{1}{2}(y-M_2)^T\Sigma_2^{-1}(y-M_2) + b$

$(y^T - M_1^T)\Sigma_1^{-1}(y-M_1) = \dots$

$y^T\Sigma_1^{-1} - M_1^T\Sigma_1^{-1}(y-M_1) = \dots$

$y^T\Sigma_1^{-1}y + y^T\Sigma_1^{-1}M_1 - M_1^T\Sigma_1^{-1}y + M_1^T\Sigma_1^{-1}M_1 = \dots$

Subtract right side

$D(y) = y^T(\Sigma_K^{-1} - \Sigma_J^{-1})y + y^T[\Sigma_K^{-1}M_K + \Sigma_J^{-1}M_J] + y[-M_K^T\Sigma_K^{-1} + M_J^T\Sigma_J^{-1}] + \dots$

$\dots + M_K^T\Sigma_K M_K - M_J^T\Sigma_J M_J$

$[y_1\ y_2]\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$

Decision boundary

~ Quadratic, not linear

$Dy > 0$ Class K

$Dy < 0$ Class J

2b) $\log(P) = \log\left(\frac{(\lambda)^n e^{-\lambda}}{n!}\right) = n\log(\lambda) - \lambda\log(e) - \log(n!)$

$n\log(\lambda) + -\lambda - \log(n!)$

For D neurons

LHS $\sum_{i=1}^{D}\left(y_i \log(\lambda_{ki}) - \log(y_i!) - \lambda_{ki}\right) + \log P(C_k)$

RHS $= \sum_{i=1}^{D}\left(y_i \log(\lambda_{ji}) - \log(y_i!) - \lambda_{ji}\right) + \log P(C_j)$

$$D(y) = \sum_{i=1}^{D} y_i(\log(\lambda_{ki}) - \log(\lambda_{ji})) + \lambda_{ji} - \lambda_{ki}] + \log P(C_k) - \log P(C_j))$$

Linear Boundary for Class (C_k, C_j)

if $D_y > 0$, Class C_k

if $D_y < 0$ Class C_j

Homework 4, Problem 3 Classification on simulated data

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

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) × (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.

In [1]:
```
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]
```

(a) Plot the data points

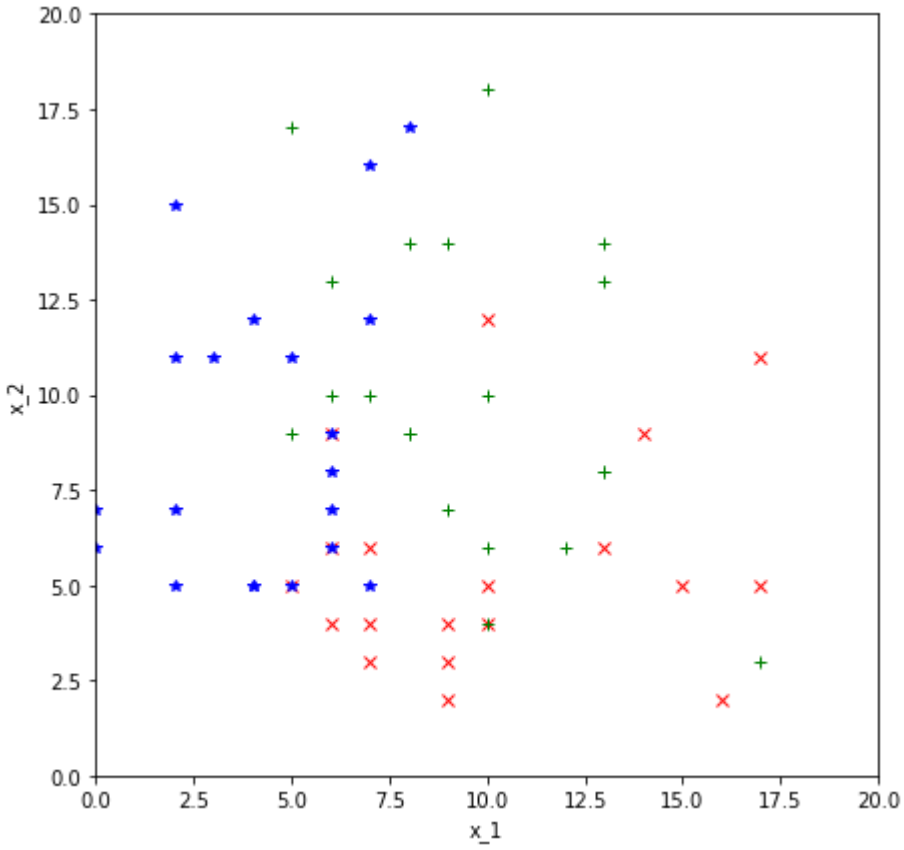
Here, to get you oriented on the dataset, we'll give you code that plots the data points. You do not have to write any new code here, but you should review this code to understand it, since

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 ×, 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.

In [2]:
```
# 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')
```

Out[2]:
```
Text(0, 0.5, 'x_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)

In [3]:
```
#=====================================================#
# 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 t
#
#   (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
#
#   These should be consistent with the print statement after this code block.
#
#   HINT: the np.mean and np.cov functions ought simplify the code.
#
#=====================================================#

X_rate_m1 = []
Y_rate_m1 = []
for classIX in range(NumClass):
    for dataIX in range(NumData):
        X_rate_m1.append(dataArr[classIX,dataIX,0])
        Y_rate_m1.append(dataArr[classIX,dataIX,1])

######Model 1######
modParam1 = {}
x_class_0_rate = X_rate_m1[0:20]
y_class_0_rate = Y_rate_m1[0:20]
x_class_1_rate = X_rate_m1[20:40]
y_class_1_rate = Y_rate_m1[20:40]
x_class_2_rate = X_rate_m1[40:60]
y_class_2_rate = Y_rate_m1[40:60]

class_0_cov = np.cov(x_class_0_rate, y_class_0_rate)
class_1_cov = np.cov(x_class_1_rate, y_class_1_rate)
class_2_cov = np.cov(x_class_2_rate, y_class_2_rate)

avg_0 = [np.mean(X_rate_m1[0:20]),np.mean(Y_rate_m1[0:20])]
avg_1 = [np.mean(X_rate_m1[20:40]),np.mean(Y_rate_m1[20:40])]
avg_2 = [np.mean(X_rate_m1[40:60]),np.mean(Y_rate_m1[40:60])]

modParam1['pi'] = np.array((1/3, 1/3, 1/3))
modParam1['mean'] = np.array([avg_0 , avg_1, avg_2])
modParam1['cov'] = (class_0_cov + class_1_cov + class_2_cov)/3
#modParam1['cov'] = np.cov(X_rate_m1[0:60], Y_rate_m1[0:60])
####################


#######Model 2 #######
modParam2 = {}
x_class_0_rate = X_rate_m1[0:20]
y_class_0_rate = Y_rate_m1[0:20]
x_class_1_rate = X_rate_m1[20:40]
y_class_1_rate = Y_rate_m1[20:40]
x_class_2_rate = X_rate_m1[40:60]
y_class_2_rate = Y_rate_m1[40:60]

class_0_cov = np.cov(x_class_0_rate, y_class_0_rate)
class_1_cov = np.cov(x_class_1_rate, y_class_1_rate)
class_2_cov = np.cov(x_class_2_rate, y_class_2_rate)

modParam2['pi'] = np.array((1/3, 1/3, 1/3))
```

```
modParam2['mean'] = np.array([avg_0 , avg_1, avg_2])
modParam2['cov'] = np.stack((class_0_cov, class_1_cov, class_2_cov))
##############



########Model 3#####
modParam3 = {}
modParam3['pi'] = np.array((1/3, 1/3, 1/3))
modParam3['mean'] = np.array([avg_0 , avg_1, avg_2])
####################


#=====================================================#
# 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:")
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:
[[12.60964912 -0.0254386 ]
 [-0.0254386  13.17105263]]
model 2:
Class priors:
[0.33333333 0.33333333 0.33333333]
Means:
[[10.75  5.55]
 [ 9.6  10.1 ]
 [ 4.3   9.  ]]
Cov1:
[[22.09210526  2.25      ]
 [ 2.25        7.62894737]]
Cov2:
[[10.04210526 -4.95789474]
 [-4.95789474 16.62105263]]
Cov3:
[[ 5.69473684  2.63157895]
 [ 2.63157895 15.26315789]]
model 3:
Class priors:
[0.33333333 0.33333333 0.33333333]
Lambdas:
[[10.75  5.55]
 [ 9.6  10.1 ]
 [ 4.3   9.  ]]
```

(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.

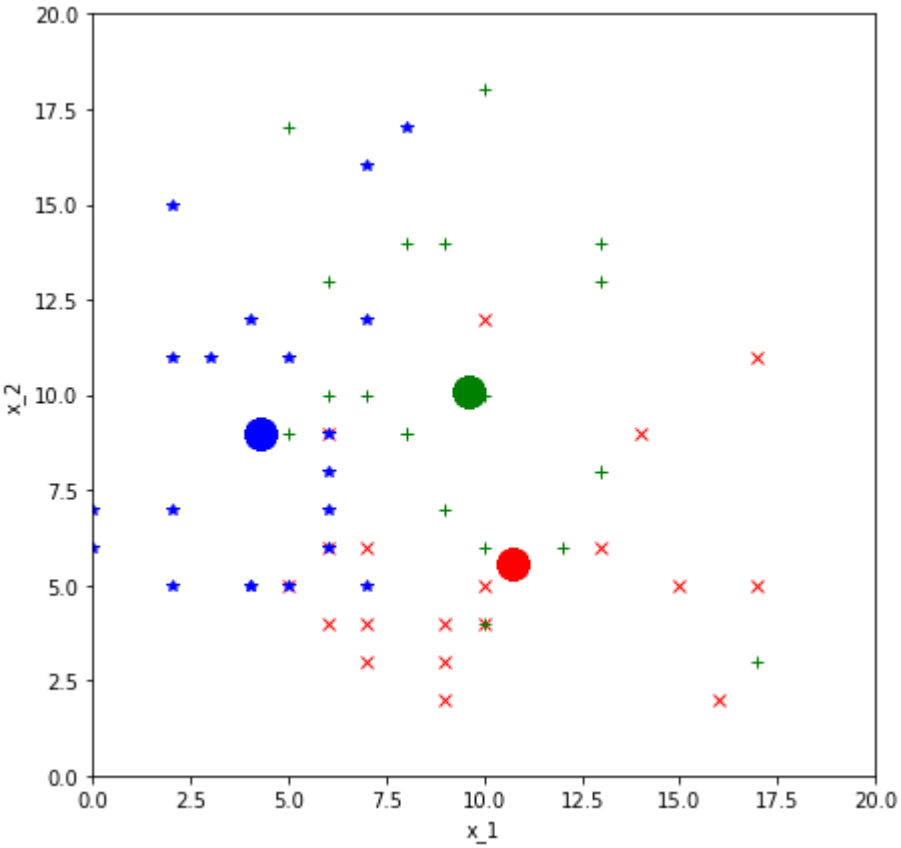
If you made modifications in the way the data is formatted, you need to change this code to visualize the ML means

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.

In [4]:
```
# 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[clas
#=====================================================#
# END CODE
#=====================================================#
plt.axis([0,20,0,20])
plt.xlabel('x_1')
plt.ylabel('x_2')
```

Out[4]: Text(0, 0.5, 'x_2')



(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).

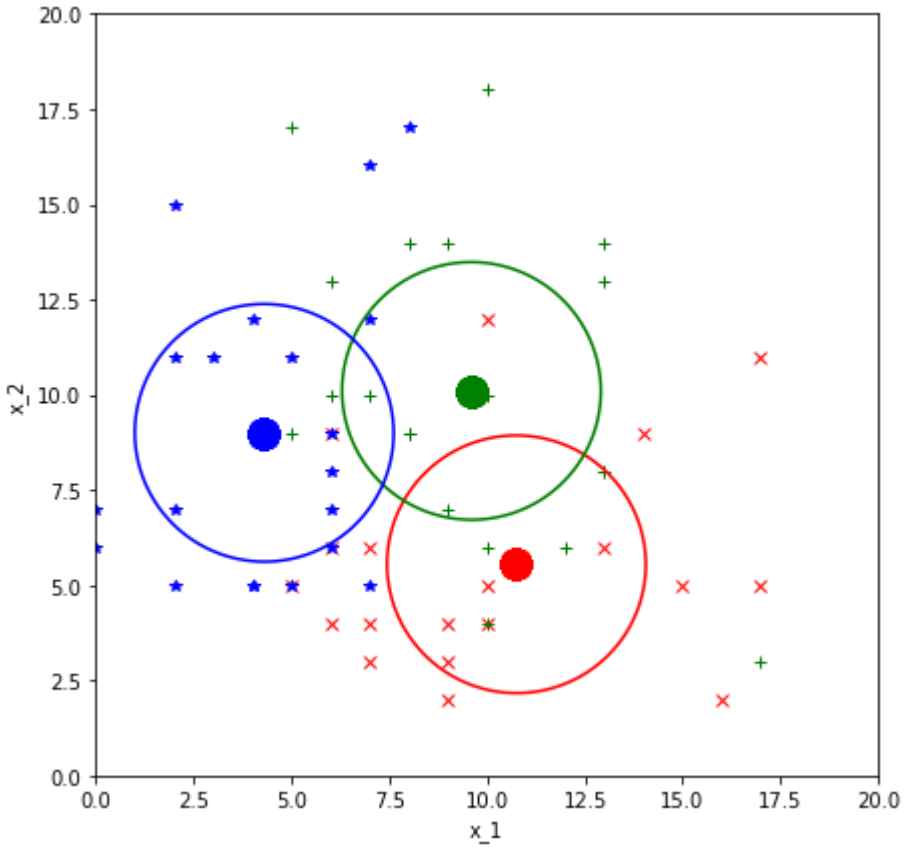
We use of `plt.contour` can be used to draw an iso-probability contour for each class. To aid interpretation, the contour should be drawn at the same probability level for each class. We call

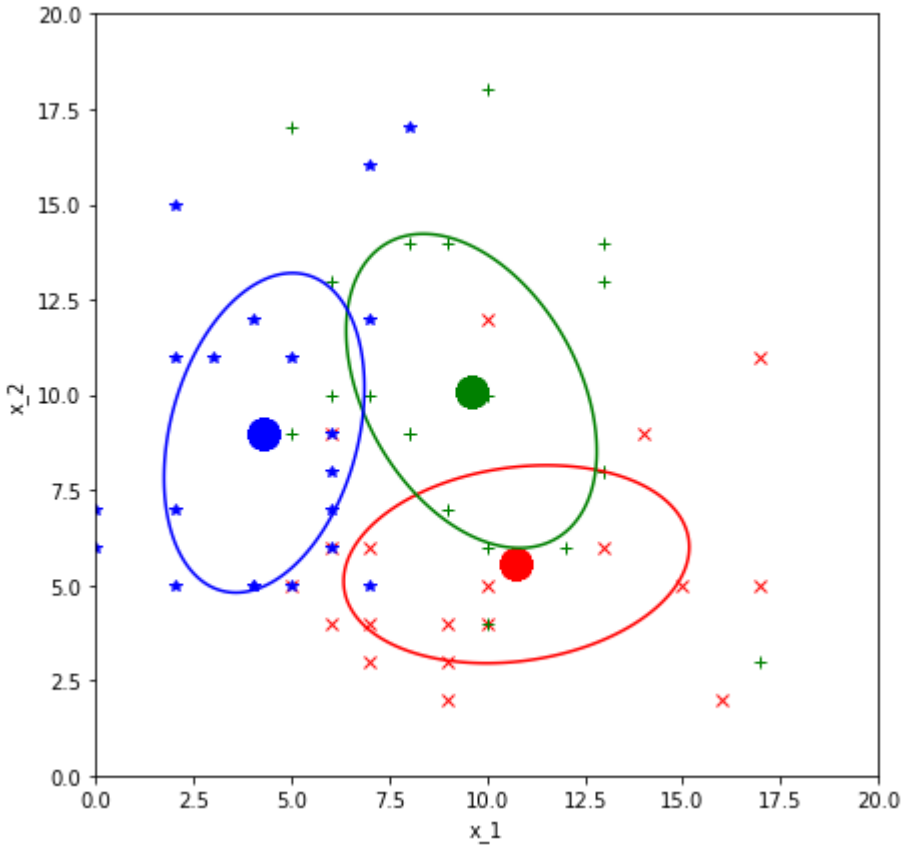
`plt.contour(X, Y, Z, levels = level, colors = color)`.

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.

In [5]:
```
# 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[cla
            plt.plot(modParam[modelIX]['mean'][classIX,0],modParam[modelIX]['mean'][cl
        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[
        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,1,colors = colors2[classIX])
#=====================================================#
# END CODE
#=====================================================#
```





(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).

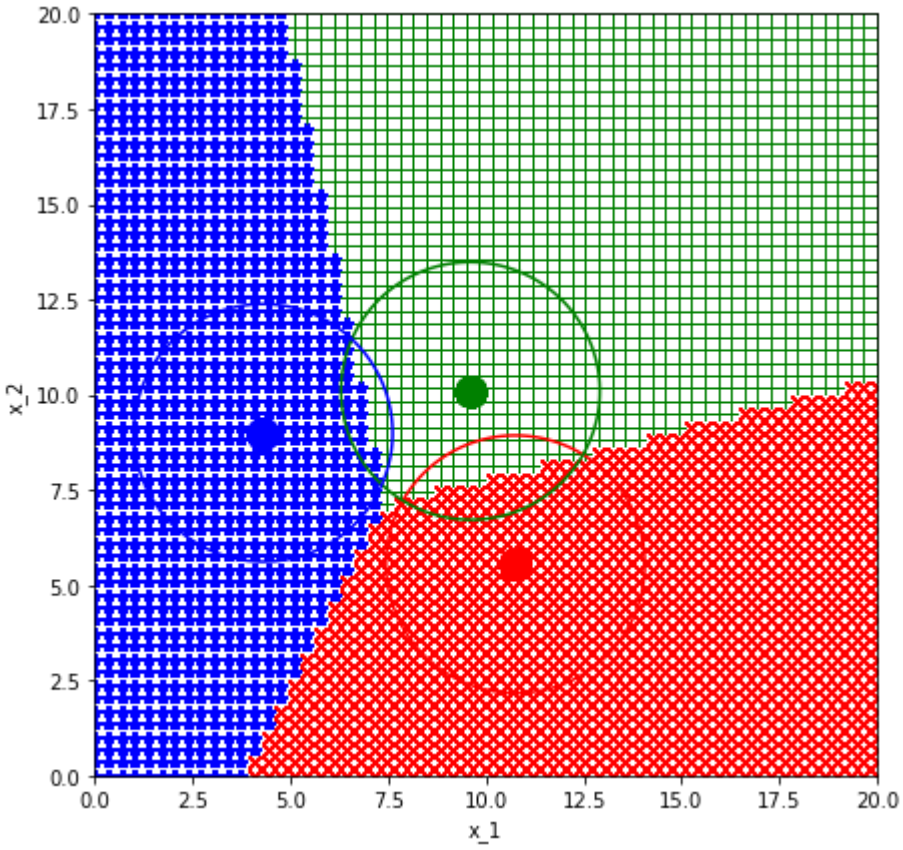
Hint 3: It's a good idea to do this one model at a time. You should get things working for model 1 for a smaller `N` value, so in code development, it doesn't take a long time to test your code. In the final result, your code will probably take some time to run because you're classifying each data point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

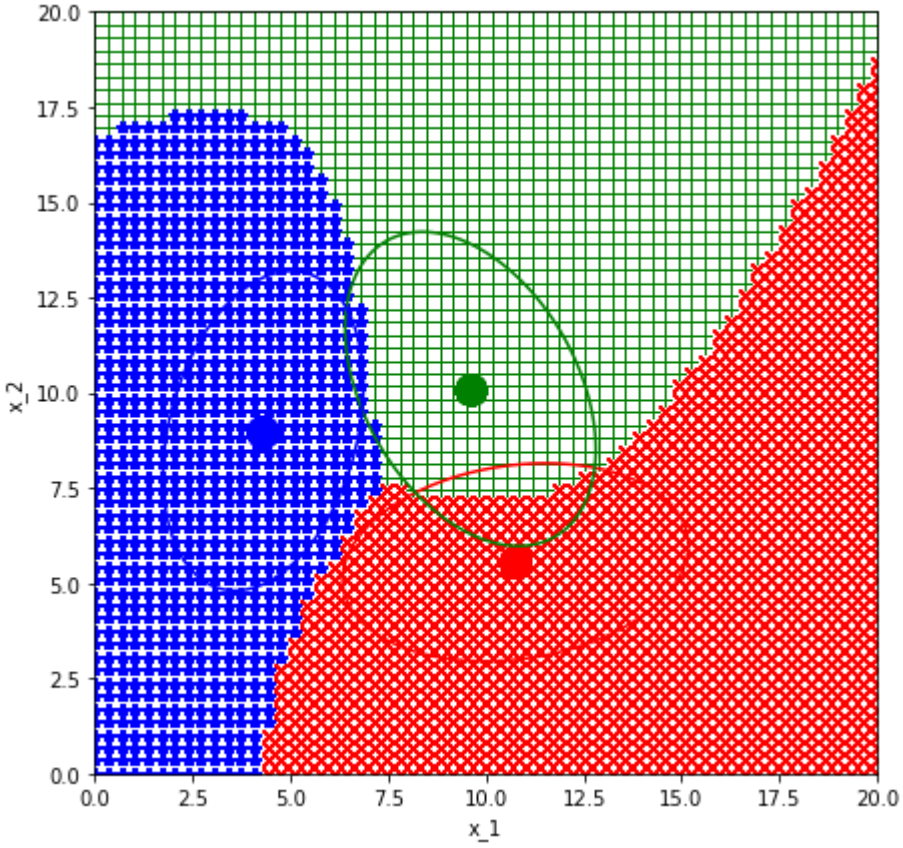
In [22]:
```
#e
#=====================================================#
# YOUR CODE HERE:
#   Plot the data points, their means, covariance ellipsoids,
#     and decision boundaries for each model.
#   Note that the naive Bayes Poisson model does not have an ellipsoid.
#   As in the above description, the decision boundary should be achieved
#     by densely classifying points in a grid.
#=====================================================#
N = 60
X = np.linspace(0, 20, N)
Y = np.linspace(0, 20, N)
print("starting")
###############Model 1#################
vals = []
for x in X:
    for y in Y:
        Z = np.array([x, y])
        inv_cov = np.linalg.inv(modParam1['cov'])

        #Class 0 prob
        diff_mean_0 = Z - modParam1['mean'][0]
        diff_mean_0_t = np.transpose(diff_mean_0)
        prob_class_0 =  -.5 *np.matmul(np.matmul(diff_mean_0_t, inv_cov), diff_mean_0)
        #Class 1 prob
        diff_mean_1 = Z - modParam1['mean'][1]
        diff_mean_1_t = np.transpose(diff_mean_1)
        prob_class_1 = -.5* np.matmul(np.matmul(diff_mean_1_t,inv_cov) , diff_mean_1)
        #Class 2 prob
        diff_mean_2 = Z - modParam1['mean'][2]
```

```
        diff_mean_2_t = np.transpose(diff_mean_2)
        prob_class_2 =  -.5*np.matmul(np.matmul(diff_mean_2_t , inv_cov), diff_mean_2)

        probs = [prob_class_0, prob_class_1, prob_class_2]
        max_probs = np.max(probs)
        if max_probs == prob_class_0:
            vals.append([x, y, 0])
        elif max_probs == prob_class_1:
            vals.append([x,y, 1])
        elif max_probs == prob_class_2:
            vals.append([x,y, 2])

#for val in vals:
    #plt.plot(val[0], val[1], MarkerPat[val[2]])
#plt.axis([0,20,0,20])
#plt.xlabel('x_1')
#plt.ylabel('x_2')

##################Model 2################
vals_2 = []
for x in X:
    for y in Y:
        Z = np.array([x, y])

        #Class 0 prob
        inv_cov_0 = np.linalg.inv(modParam2['cov'][0])
        diff_mean_0 = Z - modParam2['mean'][0]
        diff_mean_0_t = np.transpose(diff_mean_0)
        log_det_cov_0 = np.log(np.linalg.det(modParam2['cov'][0]))
        prob_class_0 = -.5 * np.matmul(np.matmul(diff_mean_0_t, inv_cov_0), diff_mean_
        #Class 1 prob
        inv_cov_1 = np.linalg.inv(modParam2['cov'][1])
        diff_mean_1 = Z - modParam2['mean'][1]
        diff_mean_1_t = np.transpose(diff_mean_1)
        log_det_cov_1 = np.log(np.linalg.det(modParam2['cov'][1]))
        prob_class_1 = -.5* np.matmul(np.matmul(diff_mean_1_t,inv_cov_1) , diff_mean_1
        #Class 2 prob
        inv_cov_2 = np.linalg.inv(modParam2['cov'][2])
        diff_mean_2 = Z - modParam2['mean'][2]
        diff_mean_2_t = np.transpose(diff_mean_2)
        log_det_cov_2 = np.log(np.linalg.det(modParam2['cov'][2]))
        prob_class_2 =  -.5*np.matmul(np.matmul(diff_mean_2_t , inv_cov_2), diff_mean_

        probs = [prob_class_0, prob_class_1, prob_class_2]
        max_probs = np.max(probs)
        if max_probs == prob_class_0:
            vals_2.append([x, y, 0])
        elif max_probs == prob_class_1:
            vals_2.append([x,y, 1])
        elif max_probs == prob_class_2:
            vals_2.append([x,y, 2])

#for val in vals_2:
    #plt.plot(val[0], val[1], MarkerPat[val[2]])
#plt.axis([0,20,0,20])
#plt.xlabel('x_1')
#plt.ylabel('x_2')

print('starting')
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[cl
            if (modelIX == 0):
                for val in vals:
                    plt.plot(val[0], val[1], MarkerPat[val[2]])
            if (modelIX == 1):
                for val in vals_2:
                    plt.plot(val[0], val[1], MarkerPat[val[2]])

            plt.plot(modParam[modelIX]['mean'][classIX,0],modParam[modelIX]['mean'][cl
        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[
        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,1,colors = colors2[classIX])


#Model 3
vals_3 = []
for x in X:
    for y in Y:
        #Z = np.array([x, y])

        #Class0 prob
        lamda_x0 = modParam3['mean'][0][0]
        lamda_y0 = modParam3['mean'][0][1]
        sum_x0 = x * np.log(lamda_x0) - lamda_x0
        sum_y0 = y * np.log(lamda_y0) - lamda_y0
        prob0 = sum_x0 + sum_y0

        #Class1 prob
        lamda_x1 = modParam3['mean'][1][0]
        lamda_y1 = modParam3['mean'][1][1]
```

```
        sum_x1 = x * np.log(lamda_x1) - lamda_x1
        sum_y1 = y * np.log(lamda_y1) - lamda_y1
        prob1 = sum_x1 + sum_y1

        #Class2 prob
        lamda_x2 = modParam3['mean'][2][0]
        lamda_y2 = modParam3['mean'][2][1]
        sum_x2 = x * np.log(lamda_x2) - lamda_x2
        sum_y2 = y * np.log(lamda_y2) - lamda_y2
        prob2 = sum_x2 + sum_y2

        probs = [prob0, prob1, prob2]
        max_probs = np.max(probs)
        classification = probs.index(max_probs)
        vals_3.append([x,y, classification])
for val in vals_3:
    plt.plot(val[0], val[1], MarkerPat[val[2]])
plt.axis([0,20,0,20])
plt.xlabel('x_1')
plt.ylabel('x_2')

print("done")
```



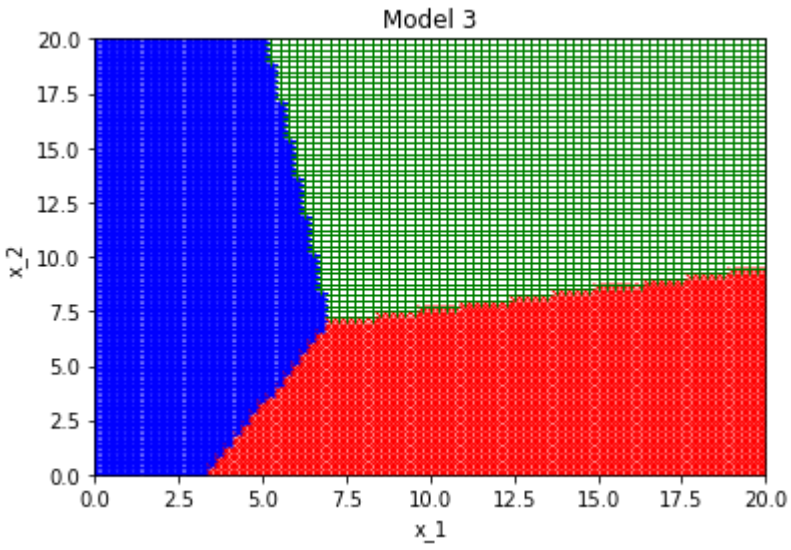


In [20]:
```
#Model 3
vals_3 = []
N = 81
X = np.linspace(0, 20, N)
Y = np.linspace(0, 20, N)
```

```
for x in X:
    for y in Y:

        #Class0 prob
        lamda_x0 = modParam3['mean'][0][0]
        lamda_y0 = modParam3['mean'][0][1]
        sum_x0 = x * np.log(lamda_x0) - lamda_x0
        sum_y0 = y * np.log(lamda_y0) - lamda_y0
        prob0 = sum_x0 + sum_y0

        #Class1 prob
        lamda_x1 = modParam3['mean'][1][0]
        lamda_y1 = modParam3['mean'][1][1]
        sum_x1 = x * np.log(lamda_x1) - lamda_x1
        sum_y1 = y * np.log(lamda_y1) - lamda_y1
        prob1 = sum_x1 + sum_y1

        #Class2 prob
        lamda_x2 = modParam3['mean'][2][0]
        lamda_y2 = modParam3['mean'][2][1]
        sum_x2 = x * np.log(lamda_x2) - lamda_x2
        sum_y2 = y * np.log(lamda_y2) - lamda_y2
        prob2 = sum_x2 + sum_y2

        probs = [prob0, prob1, prob2]

        max_probs = np.max(probs)
        classification = probs.index(max_probs)
        vals_3.append([x,y, classification])
for val in vals_3:
    plt.plot(val[0], val[1], MarkerPat[val[2]])
plt.axis([0,20,0,20])
plt.xlabel('x_1')
plt.ylabel('x_2')
plt.title('Model 3')

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

Out[20]:

```
Text(0.5, 1.0, 'Model 3')
```



In [ ]:

Homework 4, Problem 4 Classification on real data

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

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]`

In [85]:
```
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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

(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?

In [90]:
```
##4a

# Calculate the firing rates.

trainDataArr =  np.zeros((NumClass,NumTrainData,97)) # contains the firing rates for c
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,c
    for testDataIX in range(NumTestData):
        testDataArr[classIX,testDataIX,:]=np.sum(data['test_trial'][testDataIX,classIX
#=====================================================#
# YOUR CODE HERE:
#   Fit the ML parameters of model i) to training data
#=====================================================#
n_rates = []
for classk in range(8):
    mov_arr = []
    for neuron in range(97):
        neuron_arr = []
        for datas in range(91):
            neuron_arr.append(trainDataArr[classk, datas, neuron] * 5) #multiply by 5
        mov_arr.append(neuron_arr)
    n_rates.append(mov_arr)

means = []
for mov in n_rates:
    mov_arr = []
    for neuron in mov:
        avg = np.mean(neuron)
        mov_arr.append(avg)
```

```
    means.append(mov_arr)
np_mean_arr = np.empty([8,97])
for mov in range(8):
    for neuron in range(97):
        np_mean_arr[mov][neuron] = means[mov][neuron]
conv = ([vars for vars in n_rates[0]])
class_0_cov = np.cov(conv[0:])
conv = ([vars for vars in n_rates[1]])
class_1_cov = np.cov(conv[0:])
conv = ([vars for vars in n_rates[2]])
class_2_cov = np.cov(conv[0:])
conv = ([vars for vars in n_rates[3]])
class_3_cov = np.cov(conv[0:])
conv = ([vars for vars in n_rates[4]])
class_4_cov = np.cov(conv[0:])
conv = ([vars for vars in n_rates[5]])
class_5_cov = np.cov(conv[0:])
conv = ([vars for vars in n_rates[6]])
class_6_cov = np.cov(conv[0:])
conv = ([vars for vars in n_rates[7]])
class_7_cov = np.cov(conv[0:])

sum_covs = (class_0_cov + class_1_cov + class_2_cov + class_3_cov + class_4_cov
            + class_5_cov + class_6_cov + class_7_cov)

######MODEL 1######
modParam1 = {}
modParam1['pi'] = np.array((1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8,))
modParam1['mean'] = np_mean_arr
modParam1['cov'] = sum_covs/8


#######Test Model 1##########
right = 0
guesses = 0

for classk in range(8):
    for datas in range(91):
        Z = (trainDataArr[classk, datas, ] ) * 5 #multiply by 5 to get firing rate
        prob_arr = []
        inv_cov = np.linalg.inv(modParam1['cov'])

        for probs in range(8):
            diff_mean = Z - modParam1['mean'][probs]
            diff_mean_t = np.transpose(diff_mean)
            prob_class =  -.5 *np.matmul(np.matmul(diff_mean_t, inv_cov), diff_mean)
            prob_arr.append(prob_class)
        max_probs = np.max(prob_arr)
        guess = prob_arr.index(max_probs)
        if (guess == classk):
            right+=1
        guesses+=1
print(f'Accuracy is {right/guesses}')
#=================================================#
# END YOUR CODE
#=================================================#


#=================================================#
# YOUR CODE HERE:
```

```
#   Classify the test data and print the accuracy
#=====================================================#

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

```
Accuracy is 0.9807692307692307
```

Question:

What is the percent of test data points correctly classified?

Your answer:

98% were correctly classified

(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.

In [87]:
```
##4b

#=====================================================#
# YOUR CODE HERE:
# Fit the ML parameters of model ii) to training data
#=====================================================#
######MODEL 2######
modParam2 = {}
modParam2['pi'] = np.array((1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8,))
modParam2['mean'] = np_mean_arr
modParam2['cov'] = [class_0_cov, class_1_cov, class_2_cov, class_3_cov, class_4_cov, c

#######Test Model 2###########
right = 0
guesses = 0
#print(modParam2['cov'])
for classk in range(8):
    for datas in range(91):
        Z = (trainDataArr[classk, datas, ] ) * 5 #multiply by 5 to get firing rate
        prob_arr = []

        for probs in range(8):
            inv_cov = np.linalg.inv(modParam2['cov'][probs])
            diff_mean = Z - modParam2['mean'][probs]
            diff_mean_t = np.transpose(diff_mean)
            log_det_cov = np.log(np.linalg.det(modParam2['cov'][probs]))
            prob_class = -.5 * np.matmul(np.matmul(diff_mean_t, inv_cov), diff_mean) -
            prob_arr.append(prob_class)
        max_probs = np.max(prob_arr)
        guess = prob_arr.index(max_probs)
```

```
        if (guess == classk):
            right+=1
        guesses+=1
print(f'Accuracy is {right/guesses}')


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

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
Input In [87], in <cell line: 17>()
     20 prob_arr = []
     22 for probs in range(8):
---> 23     inv_cov = np.linalg.inv(modParam2['cov'][probs])
     24     diff_mean = Z - modParam2['mean'][probs]
     25     diff_mean_t = np.transpose(diff_mean)

File <__array_function__ internals>:180, in inv(*args, **kwargs)

File ~\AppData\Local\Programs\Python\Python310\lib\site-packages\numpy\linalg\linalg.
py:545, in inv(a)
    543 signature = 'D->D' if isComplexType(t) else 'd->d'
    544 extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
--> 545 ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
    546 return wrap(ainv.astype(result_t, copy=False))

File ~\AppData\Local\Programs\Python\Python310\lib\site-packages\numpy\linalg\linalg.
py:88, in _raise_linalgerror_singular(err, flag)
     87 def _raise_linalgerror_singular(err, flag):
---> 88     raise LinAlgError("Singular matrix")

LinAlgError: Singular matrix
```

Question:

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

Your answer:

Part of the Covariance matrix is not the correct size due to neurons being silent over all samples for a class.

(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?

In [88]:
```
##4c
neuronsToRemove = []
#=====================================================#
# YOUR CODE HERE:
#   Detect and then remove the offending neurons, so that
```

```
#   you no longer run into the bug in part (b).
#=========================================================#
for mov in means:
    index = 0
    for neuron in mov:
        if (np.mean(neuron)== 0):
            neuronsToRemove.append(index)
        index+=1
#=========================================================#
# END YOUR CODE
#=========================================================#
##
#=========================================================#
# YOUR CODE HERE:
# Fit the ML parameters,classify the test data and print the accuracy
#=========================================================#

def remove_mul_ele(input_list,list_of_indexes):
    list_back = input_list
    removed = 0
    for index in list_of_indexes:
        list_back.pop(index-removed)
        removed+=1
    return list_back
neuronsToRemove = sorted(list(dict.fromkeys(neuronsToRemove)))
num_removed = len(neuronsToRemove)
tot_neurons = 97 - num_removed

new_rates = []
for mov in n_rates:
    new_mov = remove_mul_ele(mov, neuronsToRemove)
    new_rates.append(new_mov)


means = []
for mov in new_rates:
    mov_arr = []
    for neuron in mov:
        avg = np.mean(neuron)
        mov_arr.append(avg)
    means.append(mov_arr)
np_mean_arr = np.empty([8,97-num_removed])
for mov in range(8):
    for neuron in range(97-num_removed):
        np_mean_arr[mov][neuron] = means[mov][neuron]
conv = ([vars for vars in new_rates[0]])
class_0_cov = np.cov(conv[0:])
conv = ([vars for vars in new_rates[1]])
class_1_cov = np.cov(conv[0:])
conv = ([vars for vars in new_rates[2]])
class_2_cov = np.cov(conv[0:])
conv = ([vars for vars in new_rates[3]])
class_3_cov = np.cov(conv[0:])
conv = ([vars for vars in new_rates[4]])
class_4_cov = np.cov(conv[0:])
conv = ([vars for vars in new_rates[5]])
class_5_cov = np.cov(conv[0:])
conv = ([vars for vars in new_rates[6]])
class_6_cov = np.cov(conv[0:])
conv = ([vars for vars in new_rates[7]])
```

```
class_7_cov = np.cov(conv[0:])

modParam2 = {}
modParam2['pi'] = np.array((1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8,))
modParam2['mean'] = np_mean_arr
modParam2['cov'] = [class_0_cov, class_1_cov, class_2_cov, class_3_cov, class_4_cov, c


#######Test Model 1##########
right = 0
guesses = 0
for classk in range(8):
    for datas in range(91):
        Z = (trainDataArr[classk, datas, ] ) * 5 #multiply by 5 to get firing rate
        inds_to_delete = sorted(neuronsToRemove, reverse=True) # [5,3,1]
        for inds in inds_to_delete:
            Z = np.delete(Z, inds)
        prob_arr= []
        for probs in range(8):
            inv_cov = np.linalg.inv(modParam2['cov'][probs])
            diff_mean = Z - modParam2['mean'][probs]
            diff_mean_t = np.transpose(diff_mean)
            log_det_cov = np.log(np.linalg.det(modParam2['cov'][probs]))
            prob_class = -.5 * np.matmul(np.matmul(diff_mean_t, inv_cov), diff_mean) -
            prob_arr.append(prob_class)
        #print(prob_arr)
        #print(classk)
        max_probs = np.max(prob_arr)
        guess = prob_arr.index(max_probs)
        if (guess == classk):
            right+=1
        guesses+=1
print(f'Accuracy is {right/guesses}')




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

```
Accuracy is 1.0
```

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:

It is higher at 100% becuase it uses individual covariance matrices which is more accurate than averaging over all the classes for one cov matrix.

(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.

In [84]:
```
##4d
#=====================================================#
# YOUR CODE HERE:
# Fit the ML parameters,classify the test data and print the accuracy
#=====================================================#
modParam3 = {}
modParam3['pi'] = np.array((1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8,))
modParam3['mean'] = np_mean_arr


right = 0
guesses = 0

##Test
for classk in range(8):
    for datas in range(91):
        Z = (trainDataArr[classk, datas, ]) * 5 #multiply by 5 to get firing rate
        inds_to_delete = sorted(neuronsToRemove, reverse=True) # [5,3,1]
        for inds in inds_to_delete:
            Z = np.delete(Z, inds)

        prob_arr= []
        for probs in range(8):
            prob = 0
            for neuron in range(97 - num_removed):
                lamda = modParam3['mean'][probs][neuron]
                summed = Z[neuron] * np.log(lamda) - lamda
                prob += summed
            prob_arr.append(prob)

        max_probs = np.max(prob_arr)
        guess = prob_arr.index(max_probs)
        if (guess == classk):
            right+=1
        guesses+=1
print(f'Accuracy is {right/guesses}')

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

```
Accuracy is 0.9313186813186813
```

Question:

what is the percent of test data points correctly classified?

Your answer:

93%