Shilpa Rao
905578954

ECE C143A/C243A, Spring 2023 **Homework #4**
Department of Electrical and Computer Engineering Prof. J.C. Kao
University of California, Los Angeles TAs T. Monsoor, R. Gore, D. Singla

Due Friday, 19 May 2023, uploaded to Gradescope.
Covers material up to Discrete Classification II.
100 points total.

Please **box your final answers for each question.**

A probabilistic generative model for classification comprises class-conditional densities $P(\mathbf{y} \mid \mathcal{C}_k)$ and class priors $P(\mathcal{C}_k)$, where $\mathbf{y} \in \mathbb{R}^D$ and $k = 1, \ldots, K$. We will consider three different generative models in this problem set:

i) Gaussian, shared covariance

$$\mathbf{y} \mid \mathcal{C}_k \sim \mathcal{N}(\boldsymbol{\mu}_k, \Sigma)$$

in class $k=2$
(plan L/R)

ii) Gaussian, class-specific covariance

$$\mathbf{y} \mid \mathcal{C}_k \sim \mathcal{N}(\boldsymbol{\mu}_k, \Sigma_k)$$

iii) Poisson

$$y_i \mid \mathcal{C}_k \sim \text{Poisson}(\lambda_{ki})$$

In iii), y_i is the ith element of the vector $\mathbf{y}$, where $i = 1, \ldots, D$. This is called a *naive Bayes* model, since the y_i are independent conditioned on $\mathcal{C}_k$.

1. (20 points) Maximum likelihood (ML) parameter estimation
In class, we derived the ML parameters for model i):

$$P(\mathcal{C}_k) = \frac{N_k}{N} \qquad \boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n \in \mathcal{C}_k} \mathbf{x}_n \qquad \Sigma = \sum_{k=1}^{K} \frac{N_k}{N} \cdot S_k,$$

where

$$S_k = \frac{1}{N_k} \sum_{n \in \mathcal{C}_k} (\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}},$$

For class specific, Σ just ⊙ this.

N_k is the number of data points in class $\mathcal{C}_k$, and N is the total number of data points in the data set.

   (a) (10 points) Find the ML parameters for model ii), i.e., find: $P(\mathcal{C}_k)$, $\boldsymbol{\mu}_k$, Σ_k. (**Hint**: You should incorporate a Lagrange multiplier that $\sum_k P(\mathcal{C}_k) = 1$.)

   (b) (10 points) Find the ML parameters for model iii), i.e., find: $P(\mathcal{C}_k)$, λ_{ki}

→ assume indp. bc can notion of multivar. Poisson dist.

2. (20 points) Decision boundaries
In class, we derived the decision boundary between class $\mathcal{C}_k$ and class $\mathcal{C}_j$ for model i):

(2) you can multiply probs ... (??

$$(\mathbf{w}_k - \mathbf{w}_j)^{\mathrm{T}} \mathbf{x} + (w_{k0} - w_{j0}) = 0,$$

where

$$\mathbf{w}_k = \Sigma^{-1} \boldsymbol{\mu}_k \qquad w_{k0} = -\frac{1}{2} \boldsymbol{\mu}_k^{\mathrm{T}} \Sigma^{-1} \boldsymbol{\mu}_k + \log P(\mathcal{C}_k).$$

For each of the models ii) and iii), we'll want to derive the decision boundary between class $\mathcal{C}_k$ and class $\mathcal{C}_j$ and say whether it is linear in $\mathbf{y}$.

   (a) (10 points) What is the decision boundary for model (ii)? Is it linear?

   (b) (10 points) What is the decision boundary for model (iii)? Is it linear?

3. (30 points) Simulated data
Please complete `hw4p3.ipynb`, which could be downloaded from BruinLearn, the data is `ps4_simdata.mat`
Please print the Jupyter Notebook and attach to your HW.

4. (30 points) Real neural data
Please complete `hw4p4.ipynb`, which could be downloaded from BruinLearn, the data is `ps4_realdata.mat`
Please print the Jupyter Notebook and attach to your HW.

   The neural data have been generously provided by the laboratory of Professor Krishna Shenoy at Stanford University. The data are to be used exclusively for educational purposes in this course.

1. (20 points) Maximum likelihood (ML) parameter estimation
In class, we derived the ML parameters for model i):

$$P(\mathcal{C}_k) = \frac{N_k}{N} \qquad \boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n \in \mathcal{C}_k} \mathbf{x}_n \qquad \Sigma = \sum_{k=1}^{K} \frac{N_k}{N} \cdot S_k,$$

where

$$S_k = \frac{1}{N_k} \sum_{n \in \mathcal{C}_k} (\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}},$$

N_k is the number of data points in class $\mathcal{C}_k$, and N is the total number of data points in the data set.

(a) (10 points) Find the ML parameters for model ii), i.e., find: $P(\mathcal{C}_k)$, $\boldsymbol{\mu}_k$, Σ_k. (**Hint**: You should incorporate a Lagrange multiplier that $\sum_k P(\mathcal{C}_k) = 1$.)

(b) (10 points) Find the ML parameters for model iii), i.e., find: $P(\mathcal{C}_k)$, λ_{ki}

(a) Find $P(C_k)$, μ_k, Σ_k for multivariate Gaussian normal [class dep. covariance]

Assume $p(y|C_k) \sim N(\mu_k, \Sigma_k)$. Similar to lecture, but: class specific covariance.
↳ k denotes class. How does D-dimensional neurons come into play?

$y \in \mathbb{R}^D$ given k

$t_{i \ldots n} \in \{C_k\}$ a trial t_i belonging to class C_k

$\mathcal{L}$ = Prob. of seeing our data given parameters.

$\mathcal{L} = \text{LIKELIHOOD}$ (N trials)

$\mathcal{L} = p\{(y_1, t_1), (y_2, t_2) \ldots (y_n, t_n) \mid \Theta\}$

(ignore Θ) $\Theta = \{\pi, \mu_k, \Sigma_k\}$ where $\pi = p(C_k)$

(why can we throw the priors out?)

$\mathcal{L} = p(y_1, t_1, y_2, t_2 \ldots) = \prod_{i=1}^{N} p(y_i, t_i) = \prod_{i=1}^{N} p(t_i) \cdot p(y_i | t_i)$ chain rule!

prob. of seeing a class $\Rightarrow p(C_k)$

$\sim N(\mu_k, \Sigma_k)$

we know:
$y | C_i \sim N(\mu_i, \Sigma_i)$, $i = 1, \ldots, k$
$p(t_i = j) = \pi_j$, $j = 1, \ldots, k$

<i in N trials>

i don't understand this jump (Δ in summation) i to N to 1 to C_k → it's because $\sum_{i \in C_1} y_i$ is # trials in class C_1.

$$= \prod_{i \in C_1} p(C_1) \cdot p(y_i | C_1) \prod_{i \in C_2} p(C_2) \cdot p(y_i | C_2) \cdots \prod_{i \in C_k} p(C_k) \cdot p(y_i | C_k)$$

↳ k classes, C

Write log of likelihood to separate the product terms into sums

WRITE $\log(\mathcal{L})$

$$\log \mathcal{L} = \sum_{i \in C_1} \log(p(C_1)\, p(y_i | C_1)) + \sum_{i \in C_2} \log(p(C_2) \cdot p(y_i | C_2)) + \ldots + \sum_{i \in C_N} \log(p(C_N) \cdot p(y_i | C_N))$$

$$\log \mathcal{L} = \sum_{i \in C_1} \log(\pi_1) + \sum_{i \in C_1} \log(p(y_i | C_1)) + \vdots + \sum_{i \in C_k} \log(\pi_k) + \sum_{i \in C_k} \log(p(y_i | C_k))$$

($\log(\pi_1)$: log of gaussian; $\log(\pi_k)$: $\pi_k = p(C_k)$)

y_i = i^{th} trial neural data

$$\sum_{i \in C_1} 1 \cdot \log(\pi_1) = \log(\pi_1) \cdot \sum_{i \in C_1} 1$$

(no dep. on i, so take out of $\sum_i$; this is the # of trials we have for class N_1.)

$$\sum_{i \in C_1} \log\left(p(y_i \mid C_1)\right) = \sum_{i \in C_1} \log N(y_i \mid \mu_1, \Sigma_1)$$

Since prob. follows normal distribution

$$\Downarrow$$

$$\log \mathcal{L} = N_1 \log(\pi_1) + \sum_{i \in C_1} \log\left(N(y_1 \mid \mu_1, \Sigma_1)\right)$$

$$+ \quad \vdots \quad + \qquad + \quad \vdots \quad +$$

$$N_k \log(\pi_k) + \sum_{i \in C_k} \log\left(N(y_k \mid \mu_k, \Sigma_k)\right)$$

Make some clever observations :)

$$= \sum_{i=1}^{k} N_i \log(\pi_i) + \sum_{q=1}^{k} \sum_{i \in C_q}^{N} \log N(y_i \mid \mu_q, \Sigma_q)$$

$$\Theta = \{ \underbrace{\mu_1, \mu_2 \ldots \mu_k}_{\mu_i \in \mathbb{R}^D}, \underbrace{\pi_1, \pi_2 \ldots \pi_k}_{\pi_i \in \mathbb{R}}, \underbrace{\Sigma_1, \Sigma_2 \ldots \Sigma_k}_{\Sigma_i \in \mathbb{R}^{D \times D}} \}$$

where D = dimension of neural data (D neurons)

Now, we want to maximize the parameters in Θ → μ_k, Σ_k make up param's for $p(y_i \mid C_k)$ in any given class C_k. $p(C_k) = \pi_k$.

// maximize parameters

$$\Theta^* = \arg\max_{\Theta} \log \mathcal{L}(\Theta) \quad \text{s.t.} \quad \sum_{i=1}^{k} \pi_i = 1$$

$\sum_{i=1}^{k} p(C_i) = 1$ (sum of probs = 1 ... should be a fact we can rely on & maximize against)

// μ_k maximize only μ_k, then generalize to all μ by observation

Recall (nothing to do w/ μ_k, so $\frac{\partial}{\partial \mu_k} = 0$; $f(\mu_k)$)

$$\log \mathcal{L} = \sum_{i=1}^{N} N_i \log(\pi_i) + \sum_{i \in C_1} \log N(y_i \mid \mu_1, \Sigma_1) + \ldots + \sum_{i \in C_k} \log N(y_i \mid \mu_k, \Sigma_k)$$

$$\log N(z \mid \mu, \Sigma) = -\frac{1}{2}(z-\mu)^T \Sigma^{-1} (z-\mu) - \frac{1}{2}\log|\Sigma| - \frac{D}{2}\log(2\pi)$$

$$f(\mu_k) = \sum_{i \in C_k} \log N(y_i \mid \mu_k, \Sigma_k)$$

Since that's the only term w/ dependence on μ_k.

$$\mu_k^* = \underset{\mu_k}{\operatorname{argmax}}\ f(\mu_k) = \underset{\mu_k}{\operatorname{argmax}} \left[\frac{1}{C_k} \frac{1}{2} (y_i - \mu_k)^T \Sigma_k^{-1} (y_i - \mu_k) \right]$$

↳ again, since we care ab argmax, remove all terms not related to μ_k.

Recall:

$$\frac{\partial y^T x}{\partial x} = y, \quad \frac{\partial y^T x}{\partial y} = x, \quad \frac{\partial x^T A x}{\partial x} = (A + A^T) x$$

all neurons from one trial

$y_i \in \mathbb{R}^D \rightarrow D \times 1$

mean vector per class: $\mu_k \in \mathbb{R}^D \rightarrow D \times 1$

$\Sigma, \Sigma^{-1} = D \times D$

$$\underset{\mu_k}{\operatorname{argmax}}\ f(\mu_k) = \sum_{i \in C_k} \nabla_{\mu_k} \left[-\frac{1}{2} (y_i - \mu_k)^T \Sigma_k^{-1} (y_i - \mu_k) \right]$$

$$= -\frac{1}{2} \sum_{i \in C_k} \frac{\partial}{\partial \mu_k} \left[y_i^T \Sigma_k^{-1} y_i - y_i^T \Sigma_k^{-1} \mu_k - \mu_k^T \Sigma_k^{-1} y_i + \mu_k^T \Sigma_k^{-1} \mu_k \right]$$

no μ_k terms

Dimension check: $1 \times D \cdot D \times D \cdot D \times 1 = 1 \times 1$; $1 \times D \cdot D \times D \cdot D \times 1 = 1 \times 1$; $1 \times D \cdot D \times D \cdot D \times 1 = 1 \times 1$

these two

$$= -\frac{1}{2} \sum_{i \in C_k} \frac{\partial}{\partial \mu_k} \left[-2 y_i^T \Sigma_k^{-1} \mu_k + \mu_k^T \Sigma_k^{-1} \mu_k \right]$$

$(y_i^T \Sigma_k^{-1} \mu_k)^T = (\mu_k^T \Sigma_k^{-1} y_i)$ MAT. ↑ COOKBOOK

but, scalar^T = itself

∴ they are the same

$$= -\frac{1}{2} \sum_{i \in C_k} \left[-2 \frac{\partial}{\partial \mu_k} y_i^T \Sigma_k^{-1} \mu_k + \frac{\partial}{\partial \mu_k} \mu_k^T \Sigma_k^{-1} \mu_k \right]$$

$\Sigma_k^{-1} = \Sigma_k^{-1\,T}$ Because Sigma is symmetric

$\frac{\partial y^T x}{\partial x} = y \cdot y = y_i^T \Sigma_k^{-1}$

$\frac{\partial x^T A x}{\partial x} = (A + A^T) x$

$= (\Sigma_k^{-1} + \Sigma_k^{-1\,T}) \mu_k$

$= 2 \Sigma_k^{-1} \mu_k$

$\rightarrow \text{Cov}(a,b) = \text{Cov}(b,a)$

$$= -\frac{1}{2} \sum_{i \in C_k} \left[-2 \cdot y_i \Sigma_k^{-1} + 2 \Sigma_k^{-1} \mu_k \right]$$

$$= \sum_{i \in C_k} \left[y_i \Sigma_k^{-1} - \Sigma_k^{-1} \mu_k \right] = 0$$

t_i = final # i's class value. left = 1, right = 0

$\sum_{i=1}^{N} t_i = \sum_{i \in C_1}$

lec. summing over one class..... only 2 classes in example

$$\Sigma_k^{-1} \sum_{i \in C_k} \left[y_i - \mu_k \right] = 0$$

intuition:

$$\sum_{i \in C_k} y_i - \underbrace{\sum_{i \in C_k} \mu_k}_{= \mu_k \sum_{i \in C_k} = \mu_k N_k} = 0$$

$$\sum_{i \in C_k} y_i = \mu_k N_k$$

$$\therefore \boxed{\mu_k = \frac{1}{N_k} \sum_{i \in C_k} y_i}$$

// ANSWER FOR μ_k maximized

μ_k: $D \times 1$
y_i: $D \times 1$
column first
Col x row in this class

$$\Sigma_k^* = \underset{\Sigma_k}{\operatorname{argmax}}\ f(\Sigma_k)$$

$$f(\Sigma_k) = \sum_{i \in C_k} \log N(y_i \mid \mu_k, \Sigma_k)$$

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

let

$f(A) \div -\frac{1}{2}$

$$= \sum_{i \in C_k} \left[-\frac{1}{2}(y_i - \mu_k)^T \Sigma_k^{-1} (y_i - \mu_k) \right]$$

$$+ \frac{N_k}{2}\log|\Sigma_k|^{-1}$$

← since we negated the $\ominus$ sign

$$\Sigma_k^* = \underset{\Sigma_k}{\operatorname{argmax}}\ f(\Sigma_k) \longrightarrow$$ per discussion, $y = y_i,\ x = \mu_k,\ A = \Sigma_k$

$* f(A) = f(\Sigma_k) - \frac{N_k}{2}\log|\Sigma_k|^{-1}$ → remember to map back at end!!

$$f(A) = (y - x)^T A^{-1} (y - x)$$

$$= (y^T - x^T)(A^{-1}y - A^{-1}x) = y^T A^{-1} y - y^T A^{-1} x - x^T A^{-1} y + x^T A^{-1} x$$

$$\nabla_A f(A) = \nabla_A (y^T A^{-1} y) - \nabla_A (y^T A^{-1} x) - \nabla_A (x^T A^{-1} y) + \nabla_A x^T A^{-1} x$$

① ② ③ ④

Recall: $\nabla_x \operatorname{Tr}(AX^{-1}B) = -(X^{-1}BAX^{-1})^T$ and $\operatorname{Tr}(\text{scalar}) = \text{scalar}$

1... $\nabla_A (y^T A^{-1} y) = \nabla_A \operatorname{Tr}(y^T A^{-1} y) = -(A^{-1} y y^T A^{-1})^T = -(y^T A^{-1})^T (A^{-1} y)^T$

(1×D) (D×D) (D×1)

Ⓡ Dimensions to prove it's scalar

D×1 → col

2... $-\nabla_A (y^T A^{-1} x) = -(A^{-1} x y^T A^{-1})^T = -(y^T A^{-1})^T (A^{-1} x)^T$

3... $-\nabla_A (x^T A^{-1} y) = -(x^T A^{-1})^T (A^{-1} y)^T$

4... $\nabla_A x^T A^{-1} x = -(x^T A^{-1})^T (A^{-1} x)^T$

IGNORE

$$\nabla_A f(A) = -(y^T A^{-1})^T (A^{-1} y)^T + (y^T A^{-1})^T (A^{-1} x)^T$$

$$+ (x^T A^{-1})^T (A^{-1} y)^T - (x^T A^{-1})^T (A^{-1} x)^T$$

$y = y_i,\ x = \mu_k,\ A = \Sigma_k$

$* -\frac{1}{2} f(\Sigma_k^{-1}) = f(A^{-1})$

$$\downarrow \quad f(\Sigma_k) = -\frac{1}{2} \cdot f(A) + \frac{N_k}{2} \log |\Sigma_k|^{-1}$$

$f(A)$

$$\nabla f(\Sigma_k) = -\frac{1}{2} \left[-(y_i^T \Sigma_k^{-1})^T (\Sigma_k^{-1} y_i)^T + (y_i^T \Sigma_k^{-1})^T (\Sigma_k^{-1} \mu_k)^T + (\mu_k^T \Sigma_k^{-1})^T (\Sigma_k^{-1} y_i)^T - (\mu_k^T \Sigma_k^{-1})^T (\Sigma_k^{-1} \mu_k)^T \right]$$

$$+ \nabla_{\Sigma_k} \left[\frac{N_k}{2} \log |\Sigma_k|^{-1} \right]$$

IGNORE

$$= -\frac{1}{2} [f(A)] + \frac{N_k}{2} \cdot \frac{1}{\Sigma_k^{-1}}$$

= 0

\\ removed abs. val.... am i allowed to?

NO it's a determ.

Re: matrix cookbook

$$\frac{N_k}{\Sigma_k^{-1}} = -(y_i^T \Sigma_k^{-1})^T (\Sigma_k^{-1} y_i)^T + (y_i^T \Sigma_k^{-1})^T (\Sigma_k^{-1} \mu_k)^T + (\mu_k^T \Sigma_k^{-1})^T (\Sigma_k^{-1} y_i)^T - (\mu_k^T \Sigma_k^{-1})^T (\Sigma_k^{-1} \mu_k)^T$$

Σ_k^* RETRY:

No need for constraint: one var, one unknown

$$\Sigma_k^* = \text{argmax } f(\Sigma_k)$$

det

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

$$\Sigma_k^* = \nabla_{\Sigma_k} \left[-\frac{1}{2} (y_i - \mu_k)^T \Sigma_k^{-1} (y_i - \mu_k) \right] - \nabla_{\Sigma_k} \left[\frac{1}{2} \log |\Sigma_k| \right]$$

$$\nabla_{\Sigma_k} \left[y_i^T \Sigma_k^{-1} y_i - y_i^T \Sigma_k^{-1} \mu_k - \mu_k^T \Sigma_k^{-1} y_i + \mu_k^T \Sigma_k^{-1} \mu_k \right]$$

$1 \times D \cdot D \times D \cdot D \times 1 = 1 \times 1$ | $1 \times D \cdot D \times D \cdot D \times 1 = 1 \times 1$ | $1 \times D \cdot D \times D \cdot D \times 1 = 1 \times 1$ | $1 \times D \cdot D \times D \cdot D \times 1 = 1 \times 1$

$$\text{Tr}\left(-\frac{1}{2} (y_i - \mu_k)^T \Sigma_k^{-1} (y_i - \mu_k) \right) = -\frac{1}{2} \text{Tr}\left(\Sigma_k^{-1} (y_i - \mu_k)(y_i - \mu_k)^T \right)$$

scalar!

$$\nabla_{\Sigma_1} \sum_{i=1}^{n} \sum_{i \in C_1} -\frac{1}{2} \operatorname{Tr}\left(\Sigma^{-1}(y_i - \mu_1)(y_i - \mu_1)^T\right) - \frac{1}{2}\log|\Sigma_1|$$

$$+ \cdots\cdots\cdots + \sum_{i \in C_k} -\frac{1}{2} \operatorname{Tr}\left(\Sigma^{-1}(y_i - \mu_k)(y_i - \mu_k)^T\right)$$

$$- \frac{1}{2}\log|\Sigma_k|$$

$$\frac{\partial}{\partial \Sigma_k}\left[\operatorname{Tr}\left(\Sigma_k^{-1}(y_i - \mu_k)(y_i - \mu_k)^T\right) - \frac{1}{2}\log|\Sigma_k|\right]$$

$$= -\Sigma_k^{-1}(y_i - \mu_k)(y_i - \mu_k)^T\Sigma_k^{-1} - \Sigma_k^{-1}$$

$$(AA^T)^T = AA^T$$

$$N_k\Sigma_k^{-1} = \sum_{i \in C_k}\left(\Sigma_k^{-1}(y_i - \mu_k)(y_i - \mu_k)^T\Sigma_k^{-1}\right)$$

$$N_k = \sum_{i \in C_k}(y_i - \mu_k)(y_i - \mu_k)^T\Sigma_k^{-1}$$

$$\boxed{\Sigma_k = \frac{1}{N_k}\sum_{i \in C_k}(y_i - \mu_k)(y_i - \mu_k)^T}$$

We need constraint here because we have unknown many variables (1, 2 ... k) for π_k. So the constraint helps us maximize π_i's without having k equations. Cool!

// π_k:

$$f(\pi_1, \pi_2 \cdots \pi_k) = \sum_{i=1}^{k} N_i \log \pi_i$$

→ we have a func. of all π_i [$i \in \{1, k\}$] since the likelihood equation comes with a summation.

$$\pi_1^*, \pi_2^* \ldots \pi_k^* = \underset{\{\pi_1, \ldots \pi_k\}}{\arg\max} \left[f(\pi_1, \ldots, \pi_k) - \lambda \left(\sum_{i=1}^{k} \pi_i P(C_i) - 1 \right) \right]$$

$\left[\sum_{i=1}^{k} \pi_i = 1 \right]$ condition

// Constraint: sum = 1

// λ forces whole term to be 0

↳ λ = LAG. MULTIPLIER

$$\frac{\partial}{\partial \pi_i} \left(\log 2 - \lambda \left(\sum_{i=1}^{k} \pi_i - 1 \right) \right) \rightarrow \frac{\partial}{\partial \pi_i} \lambda \sum_{i=1}^{k} \pi_i - \frac{\partial}{\partial \pi_i} 1 \rightarrow 0$$

↳ $= \lambda$

$$= \frac{\partial}{\partial \pi_i} \left[N_i \log \pi_i - \lambda \pi_i \right] = 0$$

↓

Remove summation since we're only talking about ONE π_i

$$N_i \cdot \frac{1}{\pi_i} - \lambda = 0 \quad \text{thus} \quad \lambda = \frac{N_i}{\pi_i} \quad \ldots (1)$$

→ Now we know our lagrange multiplier, $\lambda = \frac{N_i}{\pi_i}$

(2) ... $\sum_{i=1}^{k} \pi_i = 1 \rightarrow \sum_{i=1}^{k} \frac{N_i}{\lambda} = 1$

Σ # trials in each class = # trials total

$$\rightarrow \lambda = \sum_{i=1}^{k} N_i = N$$

// LAGRANGE:

$\mathcal{L}(x, \lambda) = f(x) + \lambda \cdot g(x)$

Subject to constraint $g(x) = 0$.

Here, our $g(x) = \sum_{i=1}^{k} \pi_i - 1$

Since the $\Sigma = 1$.

(1) & (2) ... $N = \frac{N_i}{\pi_i}$

thus $\pi_i = \frac{N_i}{N}$

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

Why do we maximize the func. + constraint?

↳ if you maximize just the function, then what's the point of the constraint?

↳ the constraint is, in a way, telling you where on the function to look.

1 (b)

This problem is "naively" saying that each neuron/firing rate is independent

$p(C_k)$, λ_{ki}? **NAIVE BAYES CLASSIFIER**

iii) Poisson → $p(y_i = y_i \mid C_k) \sim \text{Pois}(\lambda_{ki})$

$$y_i \mid C_k \sim \text{Poisson}(\lambda_{ki})$$

In iii), y_i is the ith element of the vector $\mathbf{y}$ (vector of neural activity), where $i = 1, \ldots, D$. This is called a *naive* Bayes model (naive / "strong" assumption of independence), since the y_i are independent conditioned on C_k. → each trial = indep. cond. on its class.

We take the same steps as in part (1a) to obtain the

$$\log \mathcal{L} = \sum_{i \in C_1} \log(p(C_1)) + \sum_{i \in C_1} \log(p(y_i \mid C_1)) + \cdots + \sum_{i \in C_k} \log(p(C_k)) + \sum_{i \in C_k} \log(p(y_i \mid C_k))$$

(π_1, π_k; Poisson distribution; y_i = i^{th} trial neural data)

$$p(y_i \mid C_k = a) = \frac{\lambda_{ki}^a e^{-\lambda_{ki}}}{a!} \qquad p(y_i \mid C_k) = p(y = y_i) = \frac{\lambda^{y_i} e^{-\lambda_{ki}}}{y_i!}$$

$$\log \mathcal{L} = \sum_{i=1}^{k} N_i \log(\pi_i) + \sum_{i \in C_1} \sum_j p(y_j \mid C_i) + \cdots + \sum \sum p(y_j \mid C_k) - \lambda\left(\sum_{k=1}^{k} \pi - 1\right)$$

(constraint; only stat π solution)

Conditional poisson

$$\frac{\lambda_{ki}^{y_i} e^{-\lambda_{ki}}}{y_i!}$$

y_i ↑ scalar

$p(y_i \mid C_k)$

$p(y_i = 3 \mid C_k)$

each neuron has its own FR

$p(C_k) = \pi_k$

$$\frac{\partial(\log \mathcal{L} - \lambda(\sum_{k=1}^{K} \pi - 1))}{\partial \pi} = 0 = \frac{N_k}{\pi} - \lambda$$

$$\pi = \frac{N_k}{\lambda} \quad \hookrightarrow = N$$

$$\sum_{k=1}^{K} \pi = 1 = \sum_{k=1}^{K} \frac{N_k}{\lambda}$$

$$\boxed{\pi = \frac{N_k}{N}}$$

λ_{kj}

$$\frac{\partial \log \mathcal{L}}{\partial \lambda_{kj}} = \frac{\partial}{\partial \lambda_{kj}} \left(\sum_{k \text{ classes}} \sum_{i}^{K} \sum_{j} \log p(y_j | C_i) \right)$$

$$\Rightarrow \frac{\partial}{\partial \lambda_{kj}} \log \mathcal{L} = \frac{\partial}{\partial N_{kj}} \left(\sum_{i \in C_k} \log \left(\frac{\lambda_{kj}^{y_j^{(i)}} e^{-\lambda_{kj}}}{y_j^{(i)}!} \right) \right)$$

$$= \frac{\partial}{\partial N_{kj}} \left(\sum_{i \in C_k} \left(y_j^{(i)} \log(\lambda_{kj}) - \lambda_{kj} - y_j^{(i)}! \right) \right)$$

$$= \sum_{i \in C_k} \left(\frac{y_j^{(i)}}{\lambda_{kj}} - 1 \right) = 0$$

$$N_k = \sum_{i \in C_k} \frac{y_j^{(i)}}{\lambda_{kj}}$$

$$\boxed{\lambda_{kj} = \frac{1}{N_k} \sum_{i \in C_k} y_j^{(i)}}$$

2. (20 points) Decision boundaries

(2) you can multiply probs

In class, we derived the decision boundary between class $\mathcal{C}_k$ and class $\mathcal{C}_j$ for model i):

$$(\mathbf{w}_k - \mathbf{w}_j)^{\mathrm{T}}\mathbf{x} + (w_{k0} - w_{j0}) = 0,$$

where

$$\mathbf{w}_k = \Sigma^{-1}\boldsymbol{\mu}_k \qquad w_{k0} = -\frac{1}{2}\boldsymbol{\mu}_k^{\mathrm{T}}\Sigma^{-1}\boldsymbol{\mu}_k + \log P(\mathcal{C}_k).$$

For each of the models ii) and iii), we'll want to derive the decision boundary between class $\mathcal{C}_k$ and class $\mathcal{C}_j$ and say whether it is linear in $\mathbf{y}$.

(a) (10 points) What is the decision boundary for model (ii)? Is it linear?

(b) (10 points) What is the decision boundary for model (iii)? Is it linear?

(a) MODEL 2

$$\hat{k} = \underset{k}{\operatorname{argmax}}\,(P(C_k \mid x)) = \underset{k}{\operatorname{argmax}}\,\frac{P(x \mid C_k)\cdot P(C_k)}{P(x)} \leftarrow \text{constant}$$

$$= \underset{k}{\operatorname{argmax}}\,(\log(P(x \mid C_k)) + \log(P(C_k)))$$

$$= \underset{k}{\operatorname{argmax}}\left[\log(P(C_k)) - \frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) - \frac{D}{2}\log 2\pi - \frac{1}{2}\log|\Sigma_k|\right]$$

$\log(P(C_k)) = \log(\pi)$

$$= \underset{k}{\operatorname{argmax}}\left[\log(\pi) - \frac{1}{2}x^T\Sigma_k^{-1}x + \frac{1}{2}x^T\Sigma_k^{-1}\mu_k + \frac{1}{2}\mu_k^T\Sigma_k^{-1}x - \frac{1}{2}\mu_k^T\Sigma_k^{-1}\mu_k - \frac{1}{2}\log|\Sigma_k|\right]$$

$$= \underset{k}{\operatorname{argmax}}\underbrace{\left[\log(\pi) - \frac{1}{2}x^T\Sigma_k^{-1}x + x^T\Sigma_k^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma_k^{-1}\mu_k - \frac{1}{2}\log|\Sigma_k|\right]}_{\alpha(x)}$$

$\alpha_i(x) = \alpha_j(x)$ \\ dec. boundary bt classes i, j

$$\log(\pi_i) - \frac{1}{2}x^T\Sigma_i^{-1}x + x^T\Sigma_i^{-1}\mu_i - \frac{1}{2}\mu_i^T\Sigma_i^{-1}\mu_i - \frac{1}{2}\log|\Sigma_i|$$

$$=$$

$$\log(\pi_j) - \frac{1}{2}x^T\Sigma_j^{-1}x + x^T\Sigma_j^{-1}\mu_j - \frac{1}{2}\mu_j^T\Sigma_j^{-1}\mu_j - \frac{1}{2}\log|\Sigma_j|$$

$\Sigma_i = \Sigma_j$

$$\log(\pi_i) + x^T\Sigma^{-1}\mu_i - \frac{1}{2}\mu_i^T\Sigma^{-1}\mu_i = \log(\pi_j) + x^T\Sigma^{-1}\mu_j - \frac{1}{2}\mu_j^T\Sigma^{-1}\mu_j$$

$$\log(\pi_i) - \log(\pi_j) + x^T\Sigma^{-1}(\mu_i - \mu_j) - \frac{1}{2}(\mu_i^T\Sigma^{-1}\mu_i - \mu_j^T\Sigma^{-1}\mu_j) = 0$$

$$\log\left(\frac{\pi_i}{\pi_j}\right) + x^T\Sigma^{-1}(\mu_i - \mu_j) - \frac{1}{2}(\mu_i^T\Sigma^{-1}\mu_i - \mu_j^T\Sigma^{-1}\mu_j) = 0$$

$$\log\left(\frac{\pi_i}{\pi_j}\right) + x^T\Sigma^{-1}(\mu_i - \mu_j) - \frac{1}{2}(\mu_i + \mu_j)^T\Sigma^{-1}(\mu_i - \mu_j) = 0$$

$$\log\left(\frac{\pi_i}{\pi_j}\right) + \left(x - \frac{1}{2}(\mu_i + \mu_j)\right)^T\Sigma^{-1}(\mu_i - \mu_j) = 0$$

$$\log\left(\frac{\pi_i}{\pi_j}\right) + x^T\Sigma^{-1}(\mu_i - \mu_j) - \frac{1}{2}(\mu_i + \mu_j)^T\Sigma^{-1}(\mu_i - \mu_j) = 0$$

$$\log\left(\frac{\pi_i}{\pi_j}\right) + x^T\Sigma^{-1}(\mu_i - \mu_j) - \frac{1}{2}(\mu_i^T\Sigma^{-1}\mu_i - \mu_j^T\Sigma^{-1}\mu_j) = 0$$

$$\log\left(\frac{\pi_i}{\pi_j}\right) + x^T\Sigma^{-1}(\mu_i - \mu_j) - \frac{1}{2}(\mu_i^T\Sigma^{-1}\mu_i - \mu_j^T\Sigma^{-1}\mu_j) = 0$$

$$\log\left(\frac{\pi_i}{\pi_j}\right) + x^T\Sigma^{-1}(\mu_i - \mu_j) - \frac{1}{2}(\mu_i + \mu_j)^T\Sigma^{-1}(\mu_i - \mu_j) = 0$$

$$\log\left(\frac{\pi_i}{\pi_j}\right) + \left(x - \frac{1}{2}(\mu_i + \mu_j)\right)^T\Sigma^{-1}(\mu_i - \mu_j) = 0$$

$x^T A x$ ⇓ $w^T x$

$$\frac{1}{2} x^T (\Sigma_j^{-1} - \Sigma_i^{-1}) x + (\mu_i^T \Sigma_i^{-1} - \mu_j^T \Sigma_j^{-1}) x$$

answer

b constant

$$+ \frac{1}{2} (\mu_j^T \Sigma_j^{-1} \mu_j - \mu_i^T \Sigma_i^{-1} \mu_i)$$

$$+ \log(\pi_i) - \log(\pi_j) - \frac{1}{2} (\log|\Sigma_i| + \log|\Sigma_j|) = 0$$

$x^T A x + w^T x + b = 0$ ⇒ quadratic decision boundary

NOT LINEAR

$$A = \frac{1}{2} (\Sigma_j^{-1} - \Sigma_i^{-1})$$

$$w = (\mu_i^T \Sigma_i^{-1} - \mu_j^T \Sigma_j^{-1})^T$$

$$b = \frac{1}{2} (\mu_j^T \Sigma_j^{-1} \mu_j - \mu_i^T \Sigma_i^{-1} \mu_i) + \log(\pi_i) - \log(\pi_j) - \frac{1}{2} (\log|\Sigma_i| + \log|\Sigma_j|)$$

(b) midterm ii

$$\hat{k} = \underset{k}{\operatorname{argmax}} \; P(x|C_k) \overbrace{p(C_k)}^{\pi_k}$$

→ independence / naive Bayes

$$= \underset{k}{\operatorname{argmax}} \left[\prod_{i=1}^{D} P(x_i|C_k) \right] \pi_k$$

$$= \underset{k}{\operatorname{argmax}} \left[\log(\pi_k) + \sum_{i=1}^{D} \log(P(x_i|C_k)) \right]$$

$$= \underset{k}{\operatorname{argmax}} \left[\log(\pi_k) + \sum_{i=1}^{D} \log\left(\frac{\lambda_{ki}^{x_i} e^{-\lambda_{ki}}}{x_i!} \right) \right]$$

$$= \arg\max_k \left[\log(\pi_k) + \sum_{i=1}^{D} (x_i \log \lambda_{ki} - \lambda_{ki} - \log x_i!) \right]$$

$\alpha_a = \alpha_b$:

$$\log \pi_a + \sum_{i=1}^{D} (x_i \log \lambda_a - \lambda_{ai}) = \log \pi_b + \sum_{i=1}^{D} (x_i \log \lambda_{bi} - \lambda_{bi})$$

$$\sum_{i=1}^{D} \left[(\log \lambda_{ai} - \log \lambda_{bi}) x_i - \lambda_{ai} + \lambda_{bi} \right] + \log \pi_a - \log \pi_b = 0$$

$$\vec{x} = [D \times 1] = \begin{bmatrix} x_1 \\ \vdots \\ x_D \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} \log \lambda_{a1} - \log \lambda_{b1} \\ \vdots \\ \log \lambda_{aD} - \log \lambda_{bD} \end{bmatrix}}_{w} x + \underbrace{\sum_{i=1}^{D} (-\lambda_{ai} + \lambda_{bi}) + \log \pi_a - \log \pi_b}_{b} = 0$$

$w^T x + b = 0$

linear // answer

Homework 4, Problem 3 Classification on simulated data

ECE C143A/C243A, Spring Quarter 2023, Prof. J.C. Kao, TAs T. Monsoor, R. Gore, D. Singla

Background

We will now apply the results of Problems 1 and 2 to simulated data. The dataset can be found on BruinLearn 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 [230…
```
pip install scipy
```

```
Requirement already satisfied: scipy in /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages (1.10.1)
Requirement already satisfied: numpy<1.27.0,>=1.19.5 in /Library/Frameworks/Python.framework/Versions/3.11/lib/
python3.11/site-packages (from scipy) (1.23.4)

[notice] A new release of pip is available: 23.0.1 -> 23.1.2
[notice] To update, run: pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

In [243…
```
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]
```

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

(a) Plot the data points

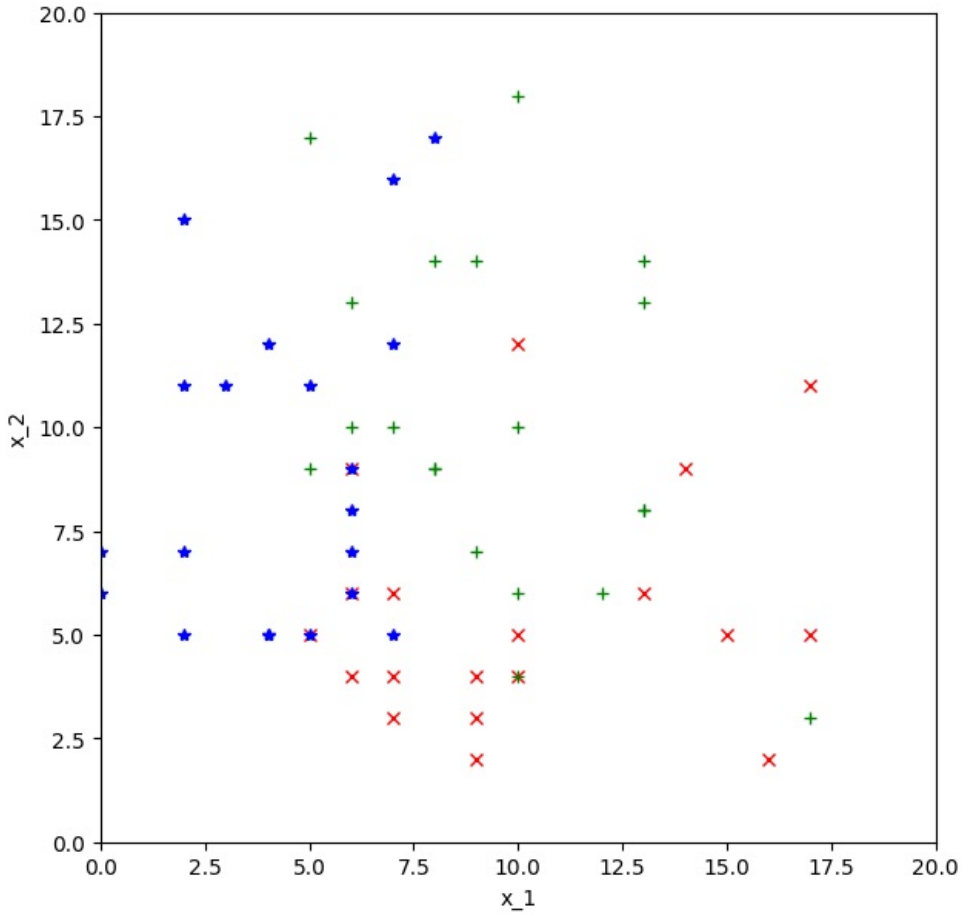
Here, to get you oriented on the dataset, we'll give you code that plots the data points. You do not have to write any new code here, but you should review this code to understand it, since we'll ask you to make plots in later parts of the notebook.

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 [244…
```
# 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[244]: 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 [245…

```
#=====================================================#
# 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.
#
#=====================================================#
# Only 2 neurons
n0 = np.size(dataArr[0]) # dataArr[0] is class k = 0's data
n1 = np.size(dataArr[1])
n2 = np.size(dataArr[2])
nTot = n0 + n1 + n2

pi0 = n0/nTot
pi1 = n1/nTot
pi2 = n2/nTot

#gaussian
modParam1 = {}
modParam1['pi'] = [pi0, pi1, pi2]
modParam1['mean'] = np.mean(dataArr,axis=1)
# COVARIANCE STEPS:
#     subtract class specific means for each class from each neuron
#     feed in [neuron1, neuron2] where data has had means subtracted
neuron1 = []
neuron2 = []
for classIX in range(3):
          #subtract class specific means
    arr1 = dataArr[classIX,:,0].flatten() - modParam1['mean'][classIX][0] #class IX neuron 0 mean
    arr2 = dataArr[classIX,:,1].flatten() - modParam1['mean'][classIX][1] #class IX neuron 1 mean
    neuron1 += [arr1]
    neuron2 += [arr2]
neuron1 = np.asarray(neuron1).flatten()
neuron2 = np.asarray(neuron2).flatten()

                          #neuron 1, neuron 2
modParam1['cov'] = np.cov([neuron1,neuron2], bias = True) #covariance ACROSS classes

#gaussian, class specific cov
modParam2 = {}
modParam2['pi'] = [pi0, pi1, pi2]
modParam2['mean'] = np.mean(dataArr,axis=1) # class 0 mean: [neuron 1, neuron 2] etc for classes 1 and 2
modParam2['cov'] = [None]*3
for classIX in range(3):
          #subtract class specific means
    arr1 = dataArr[classIX,:,0].flatten() - modParam1['mean'][classIX][0] #class IX neuron 0 mean
    arr2 = dataArr[classIX,:,1].flatten() - modParam1['mean'][classIX][1] #class IX neuron 1 mean
    modParam2['cov'][classIX] = np.cov([arr1,arr2], bias = True)

#poisson
modParam3 = {}
modParam3['pi'] = [pi0, pi1, pi2]
modParam3['mean'] = np.mean(dataArr,axis=1)
#=====================================================#
# 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:
[0.3333333333333333, 0.3333333333333333, 0.3333333333333333]
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:
[0.3333333333333333, 0.3333333333333333, 0.3333333333333333]
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:
[0.3333333333333333, 0.3333333333333333, 0.3333333333333333]
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***

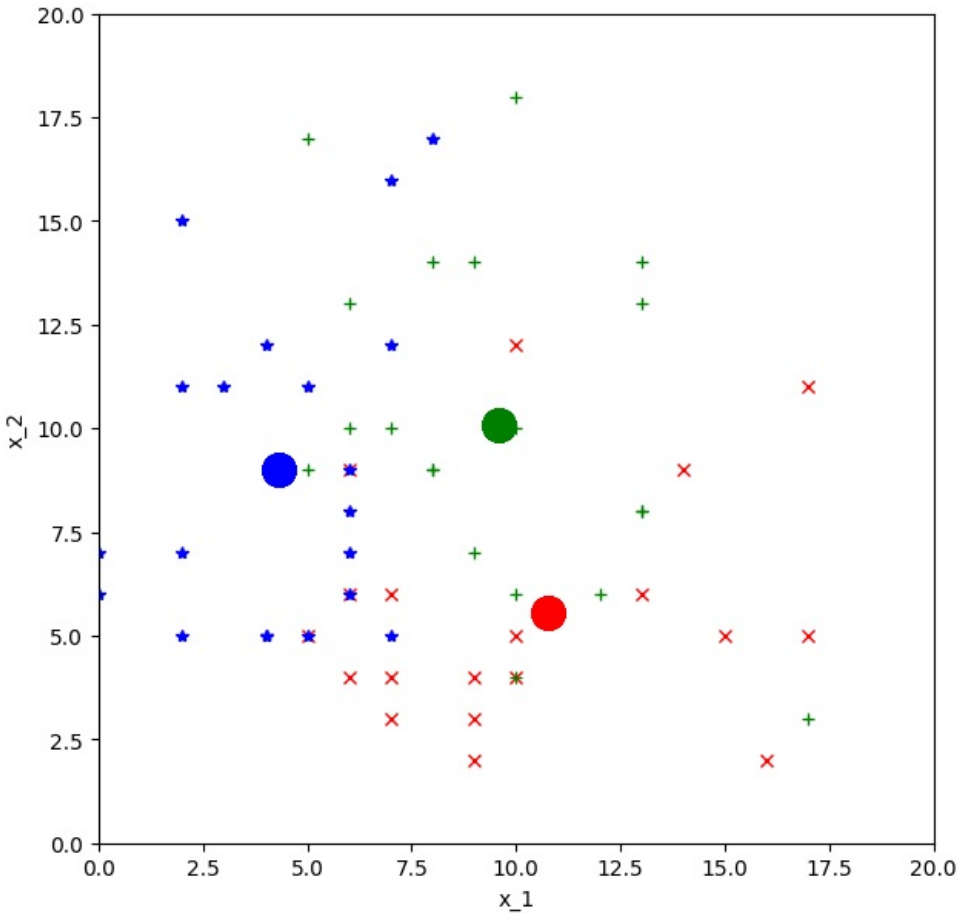
For each class, we plot the ML mean on top of the data using a solid dot of the appropriate color. We set the marker size of this dot to be much larger than the marker sizes you used in part a, so the dot is easy to see.

In [246…

```
# 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=30)
#=====================================================#
# END CODE
#=====================================================#
plt.axis([0,20,0,20])
plt.xlabel('x_1')
plt.ylabel('x_2')
```

Out[246]:

```
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)`.

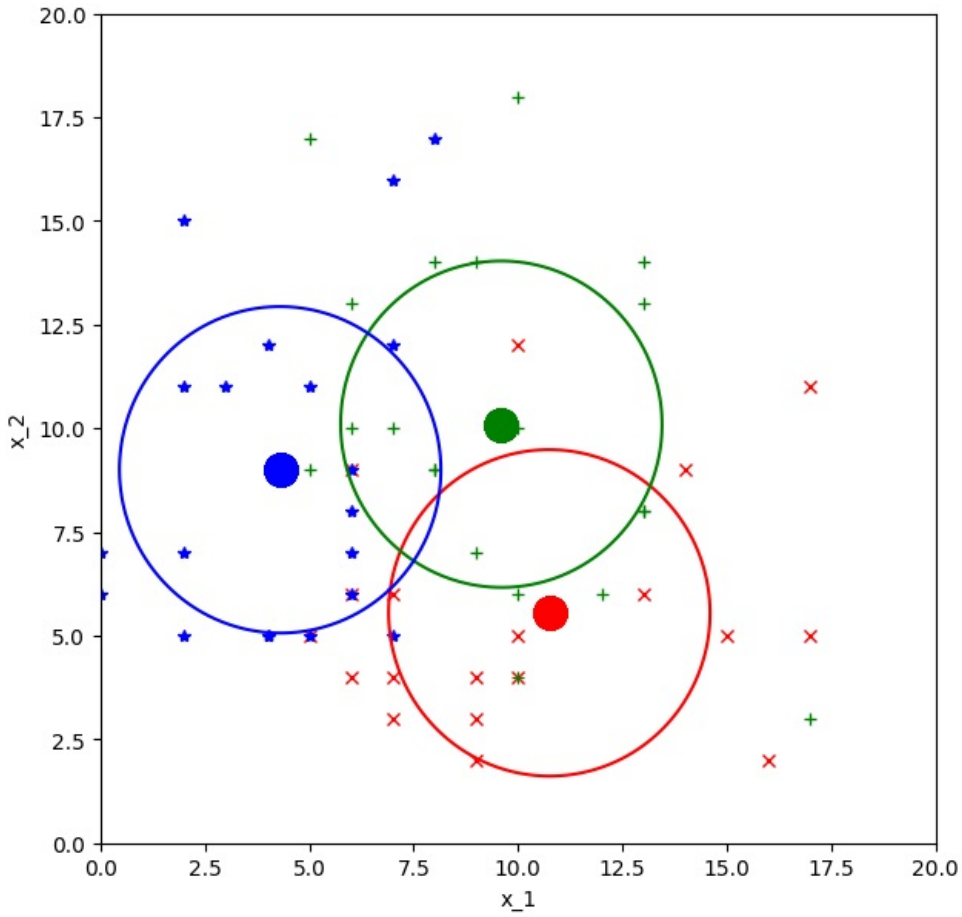
For this specific problem, we choose the contour level so you can see each ellipsoid reasonably, e.g. levels = 0.007, where X and Y are obtained via `[X,Y] = np.meshgrid(np.linspace(0, 20, N), np.linspace(0, 20, N))`, where N is the number of partitions, e.g. N = 20. Z is the function value, Please set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

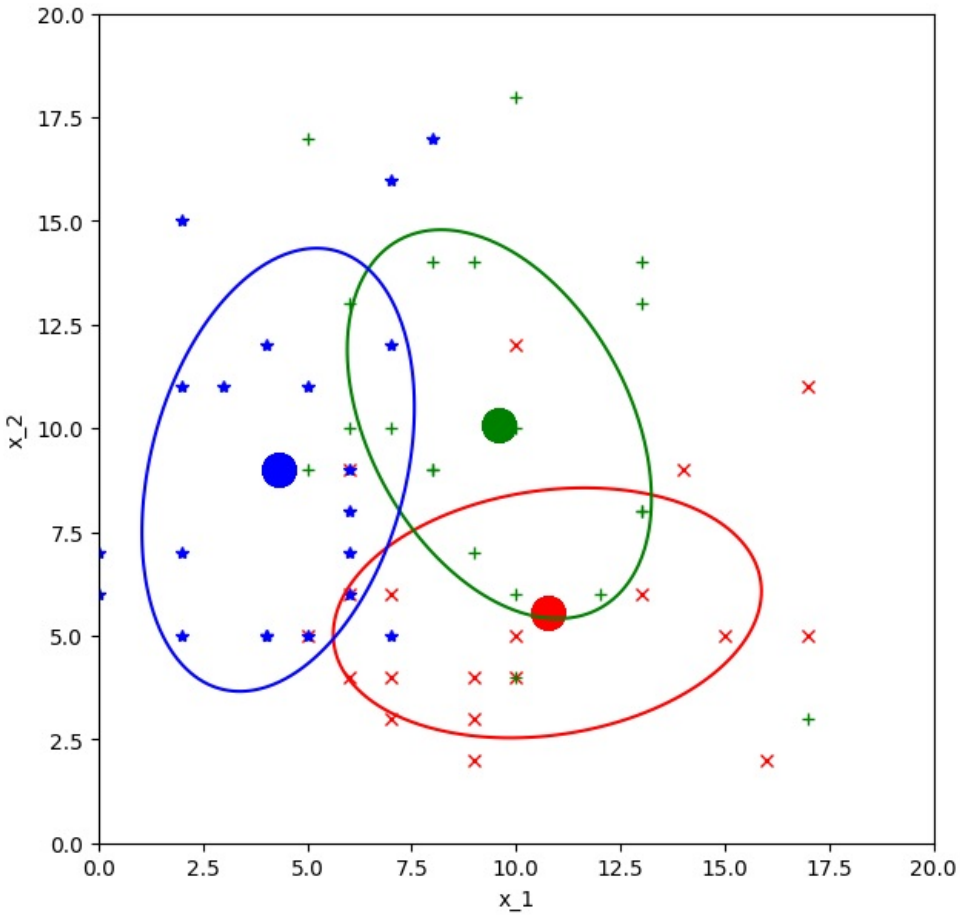
Please understand this code, as it will facilitate the last part of this notebook where we ask you to generate a plot with classification boundaries. In prior years we asked the students to generate this, but have provided it here to reduce the homework load.

In [247…

```
# 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[classIX],
        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]
        #THIS LINE IS THE PROBLEM:
        plt.contour(X,Y,Z,levels = isoThr,colors = colors2[classIX])

# no plot for poisson

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





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

Plot multi-class decision boundaries corresponding to the decision rule

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

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 [248…
```
#e argmax(p(ck | x))
# p(ck | x) equation -> take prob for each of 3 classes, then choose whichever one is highest

# make a grid
# write function that takes in data point and classifies it using model X (start with 1)

#===================================================#
# 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.
#===================================================#
```

In [249…
```
[X,Y] = np.meshgrid(np.linspace(0, 20, 81), np.linspace(0, 20, 81))
```

```
# classify 2 data points based off model 1 (gaussian, linear decision boundary)
def classify1(X,Y):
    cov = modParam1['cov']
    mean = modParam1['mean']
    pi = modParam1['pi'] # @ is matrix multiplication
    arr = np.array([X,Y])
    wk = [np.linalg.inv(cov) @ mean[k] for k in range(3)] #generate w for each class
    wk0 = [-0.5 * mean[k].T @ np.linalg.inv(cov) @ mean[k] + np.log(pi[k]) for k in range(3)]

    if ((wk[2]-wk[1]).T @ arr + (wk0[2]-wk0[1]) > 0) & ((wk[2]-wk[0]).T @ arr + (wk0[2]-wk0[0]) > 0):
        return 2
    if ((wk[1]-wk[0]).T @ arr + (wk0[1]-wk0[0]) > 0) & ((wk[1]-wk[2]).T @ arr + (wk0[1]-wk0[2]) > 0):
        return 1
    if ((wk[0]-wk[1]).T @ arr + (wk0[0]-wk0[1]) > 0) & ((wk[0]-wk[2]).T @ arr + (wk0[0]-wk0[2]) > 0):
        return 0
    return 0 # just for safety
```

In [250…
```
# classify 2 data points based off model 2 (gaussian, quadratic decision boundary)
def classify2(X,Y):
    cov = modParam2['cov'] #3 covariance matrices. cov[0] = cov for class 0, etc
    mean = modParam2['mean']
    pi = modParam2['pi'] # @ is matrix multiplication
    arr = np.array([X,Y])

    A = [1/2 * np.linalg.inv(modParam2['cov'][k]) for k in range(3)]
    W = [modParam2['mean'][k].T @ np.linalg.inv(modParam2['cov'][k]) for k in range(3)]
    b = [-1/2 * (modParam2['mean'][k].T @ np.linalg.inv(modParam2['cov'][k]) @ modParam2['mean'][k]) + np.log(m

    if (arr.T @ A[1] @ arr - arr.T @ A[0] @ arr) + (W[0] - W[1]).T @ arr + (b[0] - b[1]) > 0:
        if (arr.T @ A[2] @ arr - arr.T @ A[0] @ arr) + (W[0] - W[2]).T @ arr + (b[0] - b[2]) > 0: return 0
        else: return 2
    else:
        if (arr.T @ A[2] @ arr - arr.T @ A[1] @ arr) + (W[1] - W[2]).T @ arr + (b[1] - b[2]) > 0: return 1
        else: return 2
```

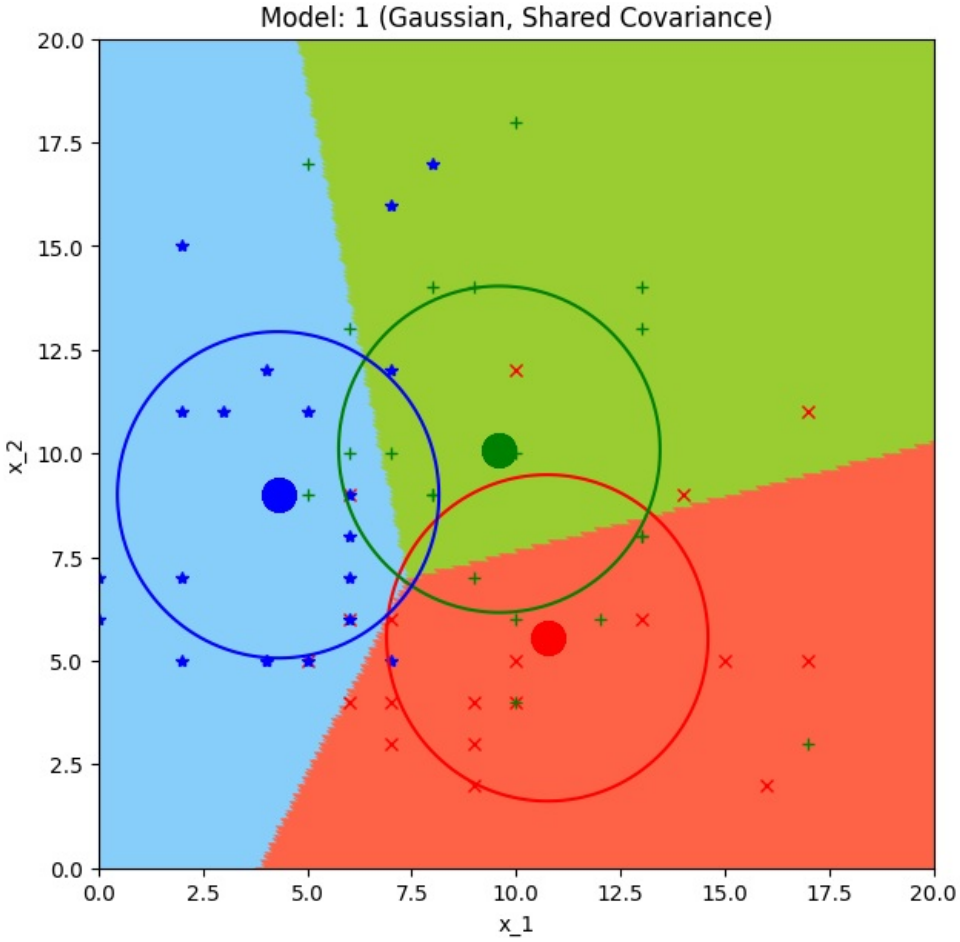
In [263…
```
# classify 2 data points based off model 3 (poisson)
def classify3(X,Y):
    numClass = 3
    arr = np.array([X,Y])

    W = [np.array([np.log(modParam3['mean'][k][i]) for i in range(len(modParam3['mean'][k]))]) for k in range(3
    b = [0]*3

    for k in range(numClass):
        for i in range(len(modParam3['mean'][k])): b[k] -= modParam3['mean'][k][i]
        b[k] += np.log(modParam3['pi'][k])

    if (W[0] - W[1]).T @ arr + (b[0] - b[1]) > 0:
        if (W[0] - W[2]).T @ arr + (b[0] - b[2]) > 0: return 0
        else: return 2
    else:
        if (W[1] - W[2]).T @ arr + (b[1] - b[2]) > 0: return 1
        else: return 2
```

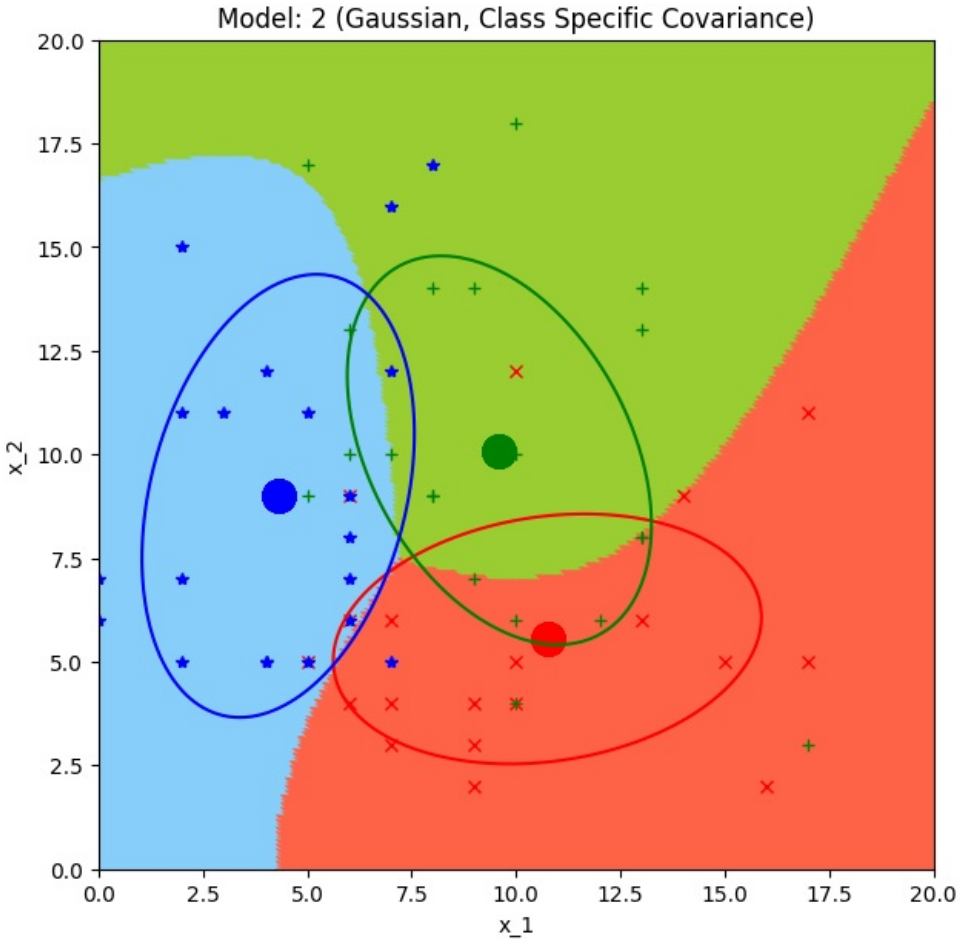
In [264…
```
# Call classify functions and put info inside dictionaries
dataMod1 = {} # all points in X,Y and their classifications according to model 1
dataMod2 = {}
dataMod3 = {}
# print((X))
# print(Y)
count = 0
for a in range(len(X)):
    for b in range(len(X[a])):
        point1 = X[a][b]
        point2 = Y[a][b]
        dataMod1[count] = [point1, point2, classify1(point1, point2)]
        dataMod2[count] = [point1, point2, classify2(point1, point2)]
        dataMod3[count] = [point1, point2, classify3(point1, point2)]
        count += 1
```

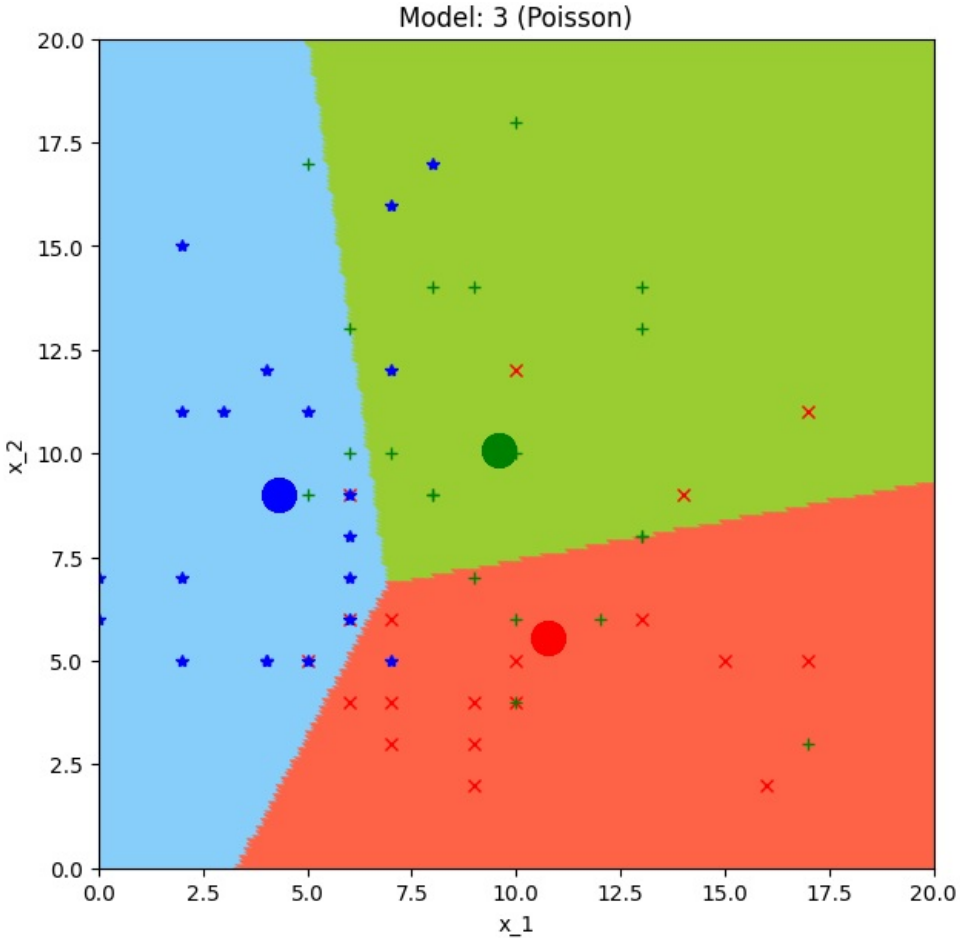
In [265…
```
# in main loop, access dataMod1/2/3 depending on which model we're talking about
colors2 = ['r','g','b']
modParam = [modParam1 , modParam2, modParam3]
clr = {0: 'tomato', 1: 'yellowgreen', 2: 'lightskyblue'}

for modelIX in range(3):
    plt.figure(modelIX,figsize=(7,7))

    for classIX in range(NumClass):
        if modelIX == 0:
            type = ' (Gaussian, Shared Covariance)'
```

```
            dic = dataMod1
        if modelIX == 1:
            type = ' (Gaussian, Class Specific Covariance)'
            dic = dataMod2
        if modelIX == 2:
            type = ' (Poisson)'
            dic = dataMod3

    # JUST FOR TESTING
    # dic = dataMod1

    # plot points of dic = {0: class, point1, point2, etc}
    for i in range(len(dic)):
        clss = dic[i][2]
        plt.plot(dic[i][0], dic[i][1], color = clr[clss], marker = 'o')
    title = "Model: " + str(modelIX + 1) + type
    plt.title(title)
    MarkerCol=['r','g','b']

    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[classIX],
        plt.axis([0,20,0,20])
        plt.xlabel('x_1')
        plt.ylabel('x_2')
        dic = {}

    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 = isoThr,colors = colors2[classIX])
    else:
        continue

''' to do tmrw:
    write decision boundary equations for classify2, 3
'''
#====================================================#
# END YOUR CODE
#====================================================#
```

Out[265]: ' to do tmrw:\n   write decision boundary equations for classify2, 3\n'


Model: 1 (Gaussian, Shared Covariance)
x_2
x_1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Model: 2 (Gaussian, Class Specific Covariance)
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



Homework 4, Problem 4 Classification on real data

ECE C143A/C243A, Spring Quarter 2023, Prof. J.C. Kao, TAs T. Monsoor, R. Gore, D. Singla

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 BruinLearn 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 [182…
```
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]
print(NumTrainData)
print(NumClass)
# Reloading any code written in external .py files.
%load_ext autoreload
%autoreload 2
```

```
91
8
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 [183…
```
##4a

# Calculate the firing rates.

trainDataArr =  np.zeros((NumClass,NumTrainData,97)) # contains the firing rates for all neurons on all 8 x 91
```

```
testDataArr =  np.zeros((NumClass,NumTestData,97)) # for the testing set.

for classIX in range(NumClass): # 8
    for trainDataIX in range(NumTrainData):   # 91
        trainDataArr[classIX,trainDataIX,:] = np.sum(data['train_trial'][trainDataIX,classIX][1][:,350:550],1)
    for testDataIX in range(NumTestData):  # 91
        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
#=====================================================#

# trainDataArr[0] = class 1
# trainDataArr[class][91 trials][97 neurons]

pis = []
tot_n = 8*91
for classIX in range(8):
    curr_n = len(trainDataArr[classIX])
    pis += [curr_n/tot_n]
# pis = 8 len array of each class's pi

modParam1 = {}
modParam1['pi'] = pis
modParam1['cov'] = np.zeros((97,97))
modParam1['mean'] = np.zeros((NumClass,97))

for classIX in range(NumClass):
    for i in range(97): modParam1['mean'][classIX][i] = np.mean(trainDataArr[classIX][:,i])
for classIX in range(NumClass): modParam1['cov'] += (trainDataArr[classIX].shape[0])/tot_n*(np.cov(trainDataArr

# COVARIANCE STEPS:
#     subtract class specific means for each class from each neuron
#     feed in [neuron1, neuron2] where data has had means subtracted
#=====================================================#
# END YOUR CODE
#=====================================================#
```

In [190…

```
#=====================================================#
# YOUR CODE HERE:
#   Classify the test data and print the accuracy
#=====================================================#
dic = np.zeros((8,91))
prob = np.zeros(8)
acc = 0

for tria in range(91):
    for i in range(8):
        y = testDataArr[i][tria][:]
        for classIX in range(8):
            meanIX = modParam1['mean'][classIX][:]
            covIX = modParam1['cov']
            prob[classIX] = -(0.5)*np.matmul((y - meanIX).T, np.matmul(np.linalg.inv(covIX),y - meanIX))
        dic[i][tria] = np.argmax(prob)
        if(dic[i][tria] == i):
            acc += 1

print('Accuracy: ', acc/(8*91))
#=====================================================#
# END YOUR CODE
#=====================================================#
```

```
Accuracy:  0.9601648351648352
```

Question:

What is the percent of test data points correctly classified?

Your answer:

96 percent!

(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 [180…

```
##4b
```

```
#=====================================================#
# YOUR CODE HERE:
# Fit the ML parameters of model ii) to training data
#=====================================================#
modParam2 = {}

# trainDataArr[0] = class 1
# trainDataArr[class][91 trials][97 neurons]

pis = []
tot_n = 8*91
for classIX in range(8):
    curr_n = len(trainDataArr[classIX])
    pis += [curr_n/tot_n]
# pis = 8 len array of each class's pi

modParam2['pi'] = pis
modParam2['cov'] = np.zeros((97,97))
modParam2['mean'] = np.zeros((NumClass,97))

for classIX in range(NumClass):
    for neur in range(97):
        modParam2['mean'][classIX][neur] = np.mean(trainDataArr[classIX][:,i])

modParam2['cov'] = np.zeros((NumClass,97,97))
for classIX in range(NumClass):
    modParam2['cov'][classIX] = np.cov(trainDataArr[classIX].T, bias=True)

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

In [191…

```
#=====================================================#
# YOUR CODE HERE:
#   Classify the test data and print the accuracy
#=====================================================#
dic = np.zeros((8,91))
prob = np.zeros(8)
acc = 0

for trial in range(91):
    for i in range(8):
        y = testDataArr[i][trial][:]
        for classIX in range(8):
            meanIX = modParam2['mean'][classIX][:]
            covIX = modParam2['cov']
            prob[classIX] = -(0.5)*np.matmul((y - meanIX).T, np.matmul(np.linalg.inv(covIX),y - meanIX))
        dic[i][trial] = np.argmax(prob)
        if(dic[i][trial] == i):
            acc += 1

print('Accuracy: ', acc/(8*91))
#=====================================================#
# END YOUR CODE
#=====================================================#
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
Cell In [191], line 15
     13     meanIX = modParam2['mean'][classIX][:]
     14     covIX = modParam2['cov']
---> 15     prob[classIX] = -(0.5)*np.matmul((y - meanIX).T, np.matmul(np.linalg.inv(covIX),y - meanIX))
     16 dic[i][trial] = np.argmax(prob)
     17 if(dic[i][trial] == i):

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

File /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/numpy/linalg/linalg.py:552
, in inv(a)
    550 signature = 'D->D' if isComplexType(t) else 'd->d'
    551 extobj = get_linalg_error_extobj( raise_linalgerror_singular)
--> 552 ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
    553 return wrap(ainv.astype(result_t, copy=False))

File /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/numpy/linalg/linalg.py:89,
in _raise_linalgerror_singular(err, flag)
     88 def _raise_linalgerror_singular(err, flag):
---> 89     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:

Because this model has class specific covariances, the matrix is not invertible. Therefore, np.linalg.inv(covIX) runs into an error. To overcome this, we may add random noise to the covariance matrix, or remove offending neurons.

(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 [192…

```
##4c
neuronsToRemove = []
#=====================================================#
# YOUR CODE HERE:
#   Detect and then remove the offending neurons, so that
#   you no longer run into the bug in part (b).
#=====================================================#
trainArr = np.zeros((8,91,97))
testArr = np.zeros((8,91,97))
for classIX in range(8):
    for trainDataIX in range(NumTrainData): trainArr[classIX][trainDataIX][:] = np.sum(data['train_trial'][trai
    for testDataIX in range(NumTestData): testArr[classIX][testDataIX][:] = np.sum(data['test_trial'][testDataI

for classIX in range(8):
    for neuron in range(97):
        fr = trainArr[classIX][:,neuron]
        if np.sum(fr) == 0:
            neuronsToRemove.append(neuron)

trainArr = np.delete(trainDataArr,list(set(neuronsToRemove)),2)
testArr = np.delete(testDataArr,list(set(neuronsToRemove)),2)
#=====================================================#
# END YOUR CODE
#=====================================================#
##
#=====================================================#
# YOUR CODE HERE:
# Fit the ML parameters,classify the test data and print the accuracy
#=====================================================#
modParam2 = {}

# trainDataArr[0] = class 1
# trainDataArr[class][91 trials][97 neurons]

pis = []
tot_n = 8*91
for classIX in range(8):
    curr_n = len(trainArr[classIX])
    pis += [curr_n/tot_n]
# pis = 8 len array of each class's pi

modParam2['pi'] = pis
modParam2['cov'] = np.zeros((87,87))
modParam2['mean'] = np.zeros((NumClass,87))

for classIX in range(NumClass):
    for i in range(87):
        modParam2['mean'][classIX][i] = np.mean(trainArr[classIX][:,i])

modParam2['cov'] = np.zeros((NumClass,87,87))
for classIX in range(NumClass):
    modParam2['cov'][classIX] = np.cov(trainArr[classIX].T, bias=True)

dic = np.zeros((8,91))
prob = np.zeros(8)
acc2 = 0

for trial in range(91):
    for i in range(8):
        y = testArr[i][trial][:]
        for classIX in range(8):
            meanIX = modParam2['mean'][classIX,:]
            covIX = modParam2['cov'][classIX]
            prob[classIX] = np.log(modParam2['pi'][classIX])-(1/2)*np.matmul((y-meanIX).T, np.matmul(np.linalg.

        dic[i][trial] = np.argmax(prob)

        if(dic[i][trial] == i):
```

```
            acc2 += 1

print('Accuracy: ', acc2/(8*91))
#================================================#
# END YOUR CODE
#================================================#
```

```
Accuracy:  0.4409340659340659
```

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: 44.1%, which is lower. We've pruned our data and lost out on some information, likely contributing to the decrease in performance. Our first model performed better.

(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 analysis.

In [168…

```
##4d
#================================================#
# YOUR CODE HERE:
# Fit the ML parameters,classify the test data and print the accuracy
#================================================#
modParam3 = {}
pis = []
tot_n = 8*91
for classIX in range(8):
    curr_n = len(trainArr[classIX])
    pis += [curr_n/tot_n]
# pis = 8 len array of each class's pi
modParam3['pi'] = pis
x, dud, y = np.shape(trainArr)
mean = np.zeros((x,y))
for classIX in range(8):
    for i in range(y): mean[classIX][i] = np.mean(trainArr[classIX][:][i])
modParam3['mean'] = mean

dic2 = np.zeros((8,91))
prob2 = np.zeros(8)
acc2 = 0
param = y
for trial in range(91):
    for i in range(8):
        ar = testArr[i][trial][:]
        for classIX in range(8):
            meanIX = modParam3['mean'][classIX][:]
            # covIX = modParam3['cov']
            prb = 0
            for el in range(param): prb += np.log(modParam3['pi'][classIX]) + (ar[el]*np.log(meanIX[el])-meanIX
            prob2[classIX] = prb
        dic2[i][trial] = np.argmax(prob2)
        if(dic2[i][trial] == i):
            acc2 += 1

print('Accuracy: ', (arr_x*acc2/(8*91)))
#================================================#
# END YOUR CODE
#================================================#
```

```
Accuracy:  0.92
```

Question:

what is the percent of test data points correctly classified?

Your answer:

92%

In [ ]: