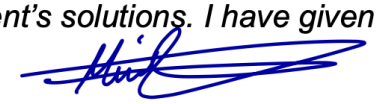
Hiva Mohammadzadeh
3036919598

Question 1: Honor Code

“I certify that all solutions are entirely in my own words and that I have not looked at another student’s solutions. I have given credit to all external sources I consulted.”



2 Gaussian Classification

Let $f(x \mid C_i) \sim \mathcal{N}(\mu_i, \sigma^2)$ for a two-class, one-dimensional classification problem with classes C_1 and C_2, $P(C_1) = P(C_2) = 1/2$, and $\mu_2 > \mu_1$.

1. Find the Bayes optimal decision boundary and the corresponding Bayes decision rule by finding the point(s) at which the posterior probabilities are equal. Use the 0-1 loss function.
2. Suppose the decision boundary for your classifier is $x = b$. The Bayes error is the probability of misclassification, namely,

   $$P_e = P((C_1 \text{ misclassified as } C_2) \cup (C_2 \text{ misclassified as } C_1)).$$

   Show that the Bayes error associated with this decision rule, in terms of b, is

   $$P_e(b) = \frac{1}{2\sqrt{2\pi}\sigma}\left(\int_{-\infty}^{b} \exp\left(-\frac{(x-\mu_2)^2}{2\sigma^2}\right) dx + \int_{b}^{\infty} \exp\left(-\frac{(x-\mu_1)^2}{2\sigma^2}\right) dx\right).$$

3. Using the expression above for the Bayes error, calculate the optimal decision boundary b^* that minimizes $P_e(b)$. How does this value compare to that found in part 1? *Hint: $P_e(b)$ is convex for $\mu_1 < b < \mu_2$.*

3 Isocontours of Normal Distributions

Let $f(\mu, \Sigma)$ be the probability density function of a normally distributed random variable in $\mathbb{R}^2$. Write code to plot the isocontours of the following functions, each on its own separate figure. Make sure it is clear which figure belongs to which part. You're free to use any plotting libraries or stats utilities available in your programming language; for instance, in Python you can use Matplotlib and SciPy. Choose the boundaries of the domain you plot large enough to show the interesting characteristics of the isocontours (use your judgment). Make sure we can tell what isovalue each contour is associated with—you can do this with labels or a colorbar/legend.

1. $f(\mu, \Sigma)$, where $\mu = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$.
2. $f(\mu, \Sigma)$, where $\mu = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 2 & 1 \\ 1 & 4 \end{bmatrix}$.
3. $f(\mu_1, \Sigma_1) - f(\mu_2, \Sigma_2)$, where $\mu_1 = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$, $\mu_2 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$ and $\Sigma_1 = \Sigma_2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$.

   (Yes, this is a difference between two PDFs. No, it is not itself a valid PDF. Just plot its isocontours anyway.)
4. $f(\mu_1, \Sigma_1) - f(\mu_2, \Sigma_2)$, where $\mu_1 = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$, $\mu_2 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$, $\Sigma_1 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$ and $\Sigma_2 = \begin{bmatrix} 2 & 1 \\ 1 & 4 \end{bmatrix}$.
5. $f(\mu_1, \Sigma_1) - f(\mu_2, \Sigma_2)$, where $\mu_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $\mu_2 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$, $\Sigma_1 = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ and $\Sigma_2 = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$.

(2) $P(C_1) = P(C_2) = 1/2$ and $\mu_2 > \mu_1$

(1) Bayes optimal decision boundary = ? Bayes decision rule = ?

To find the Bayes decision boundary: $P(C_1|x) = P(C_2|x) \rightarrow \frac{P(x|C_1)P(C_1)}{f(x)} = \frac{f(x|C_2)P(C_2)}{f(x)} \rightarrow f(x|C_1) = f(x|C_2)$

$\rightarrow N(\mu_1, \sigma^2) = N(\mu_2, \sigma^2) \rightarrow (x-\mu_1)^2 = (x-\mu_2)^2 \rightarrow x-\mu_1 = x-\mu_2 \rightarrow \boxed{x = \frac{\mu_1+\mu_2}{2}}$

using the 0-1 loss to find the decision rule: If $x < \frac{\mu_1+\mu_2}{2} \rightarrow x \Rightarrow$ class 1

otherwise $\longrightarrow x \Rightarrow$ class 2

---

(2) decision boundary: $x = b$. Bayes error: $P_e = P((C_1$ misclassified as $C_2) \cup (C_2$ misclassified as $C_1))$

Show that $P_e(b) = \frac{1}{2\sqrt{2\pi}\sigma}\left(\int_{-\infty}^{b} \exp\left(-\frac{(x-\mu_2)^2}{2\sigma^2}\right)dx + \int_{b}^{\infty} \exp\left(-\frac{(x-\mu_1)^2}{2\sigma^2}\right)dx\right)$

$P((C_1$ misclassified as $C_2) \cup P(C_2$ misclassified as $C_1)) = P(C_1$ misclassified as $C_2)P(C_1) + P(C_2$ misclassified as $C_1)P(C_2)$ ($P(C_1) = \frac{1}{2}$, $P(C_2) = \frac{1}{2}$)

$P((C_1$ misclassified as $C_2)) = \int_{\frac{\mu_1+\mu_2}{2}}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu_1)^2}{2\sigma^2}} dx = \int_{b}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma^2}} dx = P_e$

$P(C_2$ misclassified as $C_1) = \int_{-\infty}^{\frac{(\mu_1+\mu_2)}{2}} \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu_2)^2}{2\sigma^2}} dx = \int_{-\infty}^{-b} \frac{1}{\sqrt{2\pi}} e^{\frac{-(x-\mu_2)^2}{(2\sigma^2)}} dx = \frac{1}{\sqrt{2\pi}}\int_{b}^{\infty} e^{\frac{-(x-\mu_2)^2}{(2\sigma^2)}} dx = P_e$

$P(C_2$ misclassified as $C_1)P(C_2) + P(C_1$ misclassified as $C_2)P(C_1) = P_e\frac{1}{2} + P_e\frac{1}{2} = \frac{1}{2}P_e + \frac{1}{2}P_e = P_e$

$= \frac{1}{2\sqrt{2\pi}\sigma}\left(\int_{-\infty}^{b} e^{\frac{-(x-\mu_2)^2}{2\sigma^2}} dx + \int_{b}^{\infty} e^{\frac{-(x-\mu_1)^2}{2\sigma^2}} dx\right)$ ✓

---

(3) calculate the optimal decision boundary b^* that minimizes $P_e(b)$.

Using the hint: this is convex and $\mu_1 < b < \mu_2$. So, we can just take derivative and set it to zero. $\frac{d}{dx}\int_{-\infty}^{x} f(y)dy = f(x)$

$$\frac{dP_e(b)}{db} = \frac{d}{db}\left(\frac{1}{2\sqrt{2\pi}\sigma}\left[\int_{-\infty}^{b} e^{-\frac{(x-\mu_2)^2}{2\sigma^2}} dx + \int_{b}^{\infty} e^{-\frac{(x-\mu_1)^2}{2\sigma^2}} dx\right]\right) = \frac{1}{2\sqrt{2\pi}\sigma}\left(\frac{d}{db}\left[\int_{-\infty}^{b} e^{-\frac{(x-\mu_2)^2}{2\sigma^2}} dx\right] + \frac{d}{db}\left[\int_{b}^{\infty} e^{-\frac{(x-\mu_1)^2}{2\sigma^2}} dx\right]\right)$$

$$\frac{dP_e(b)}{db} = \frac{1}{2\sqrt{2\pi}\sigma}\left(e^{-\frac{(b^*-\mu_2)^2}{2\sigma^2}} - e^{-\frac{(b^*-\mu_1)^2}{2\sigma^2}}\right) = 0 \text{ (setting it to zero)} \rightarrow e^{-\frac{(b^*-\mu_2)^2}{2\sigma^2}} = e^{-\frac{(b^*-\mu_1)^2}{2\sigma^2}} \rightarrow \log e^{-\frac{(b^*-\mu_2)^2}{2\sigma^2}} = \log e^{-\frac{(b^*-\mu_1)^2}{2\sigma^2}}$$

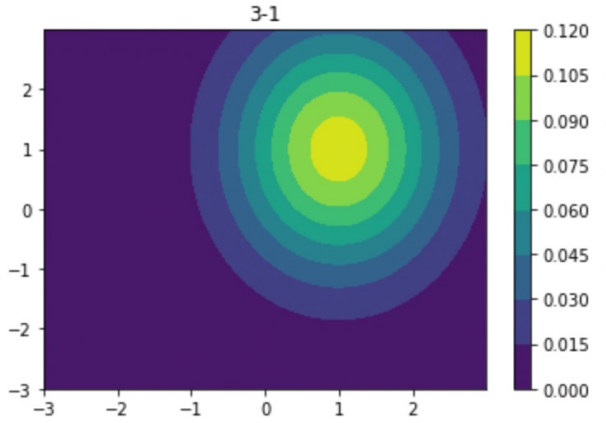
$$\rightarrow \frac{(b^*-\mu_2)^2}{2\sigma^2} = \frac{(b^*-\mu_1)^2}{2\sigma^2} \rightarrow (b^*-\mu_2)^2 = (b^*-\mu_1)^2 \rightarrow \boxed{b^* = \frac{\mu_1+\mu_2}{2}}$$

How does this value compare to that found in part 1?

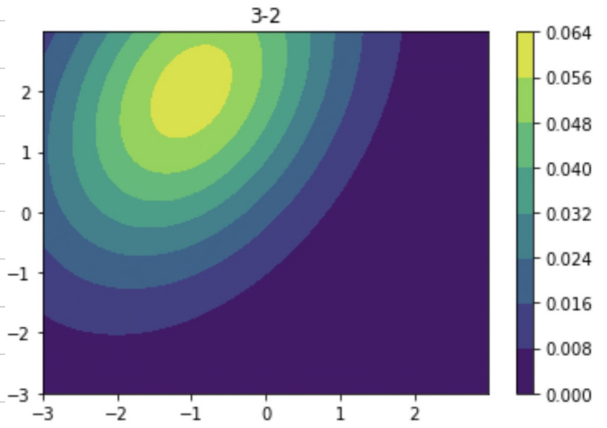
This is the same decision boundary as part 1.

3

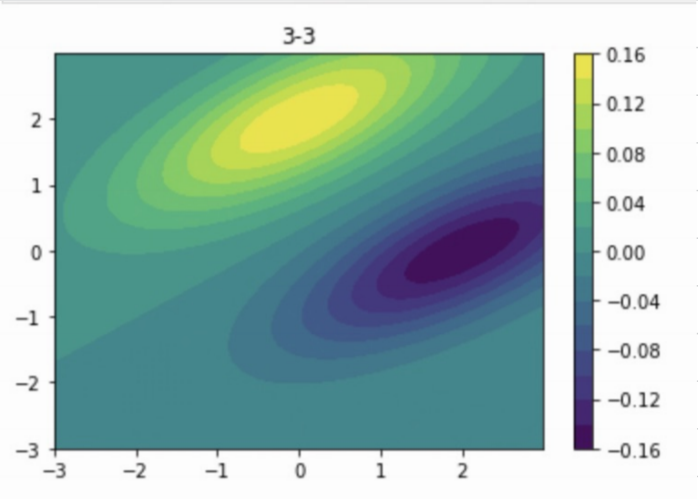
① $f(\mu, \Sigma)$ where $\mu = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$



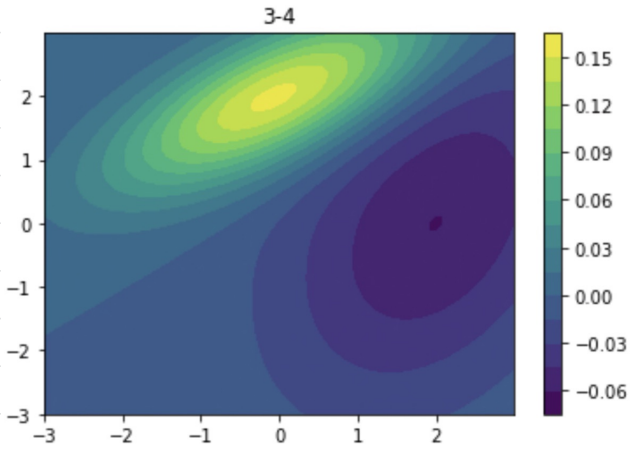
② $f(\mu, \Sigma)$ where $\mu = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 2 & 1 \\ 1 & 4 \end{bmatrix}$



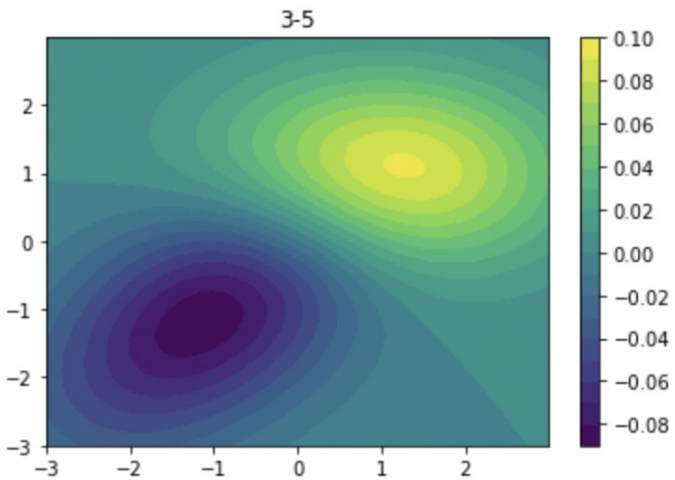
③ $f(\mu_1, \Sigma_1) - f(\mu_2, \Sigma_2)$ where $\mu_1 = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$, $\mu_2 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$ and $\Sigma_1 = \Sigma_2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$



④ $f(\mu_1, \Sigma_1) - f(\mu_2, \Sigma_2)$ where $\mu_1 = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$, $\mu_2 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$, $\Sigma_1 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$ and $\Sigma_2 = \begin{bmatrix} 2 & 1 \\ 1 & 4 \end{bmatrix}$

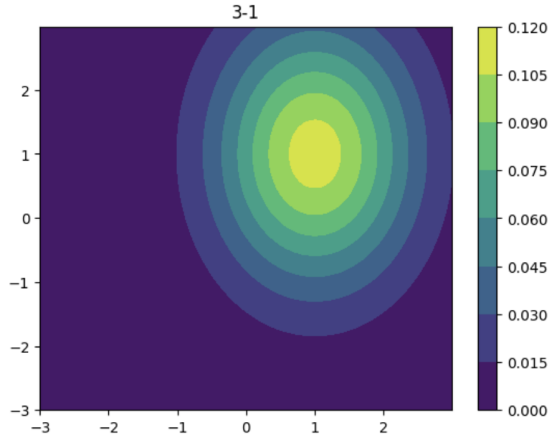


⑤ $f(\mu_1, \Sigma_1) - f(\mu_2, \Sigma_2)$ where $\mu_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $\mu_2 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$, $\Sigma_1 = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ and $\Sigma_2 = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$

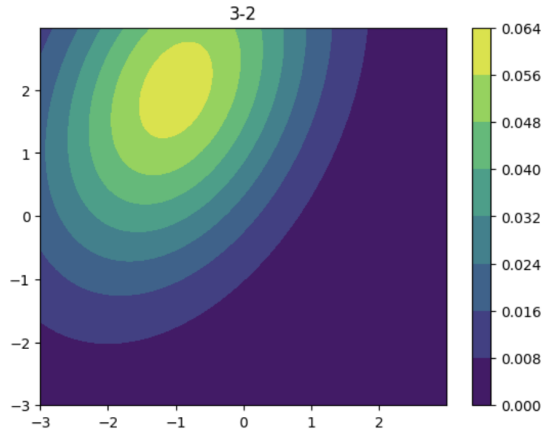


Question 3: Isocontours of Normal Distributions

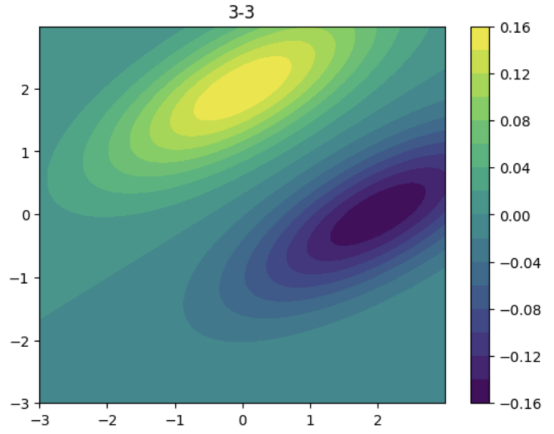
Part 1:



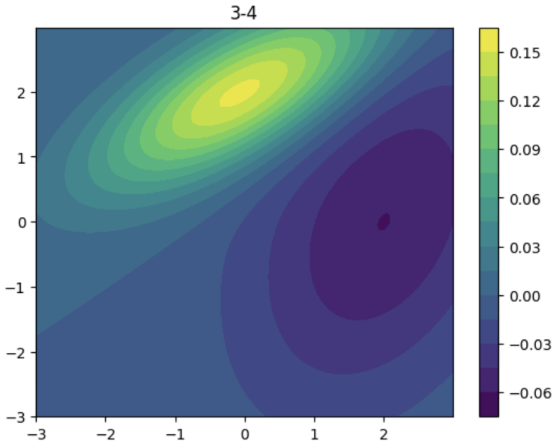
Part 2:



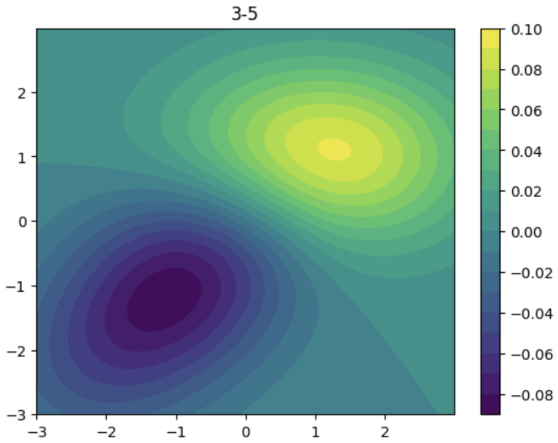
Part 3:



Part 4:


3-4
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
0.15
0.12
0.09
0.06
0.03
0.00
−0.03
−0.06

Part 5:


3-5
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08

4 Eigenvectors of the Gaussian Covariance Matrix

Consider two one-dimensional random variables $X_1 \sim \mathcal{N}(3, 9)$ and $X_2 \sim \frac{1}{2}X_1 + \mathcal{N}(4, 4)$, where $\mathcal{N}(\mu, \sigma^2)$ is a Gaussian distribution with mean μ and variance σ^2. Write a program that draws $n = 100$ random two-dimensional sample points from (X_1, X_2). For each sample point, the value of X_2 is a function of the value of X_1 for that *same* sample point, but the sample points are independent of each other. In your code, make sure to choose and set a fixed random number seed for whatever random number generator you use, so your simulation is reproducible, and document your choice of random number seed and random number generator in your write-up. For each of the following parts, include the corresponding output of your program.

1. Compute the mean (in $\mathbb{R}^2$) of the sample.
2. Compute the 2×2 covariance matrix of the sample (based on the sample mean, not the true mean—which you would not know given real-world data).
3. Compute the eigenvectors and eigenvalues of this covariance matrix.
4. On a two-dimensional grid with a horizonal axis for X_1 with range $[-15, 15]$ and a vertical axis for X_2 with range $[-15, 15]$, plot
   (i) all $n = 100$ data points, and
   (ii) arrows representing both covariance eigenvectors. The eigenvector arrows should originate at the mean and have magnitudes equal to their corresponding eigenvalues.

   Hint: make *sure* your plotting software is set so the figure is square (i.e., the horizontal and vertical scales are the same). Not doing that may lead to hours of frustration!
5. Let $U = [v_1 \quad v_2]$ be a 2×2 matrix whose columns are the **unit** eigenvectors of the covariance matrix, where v_1 is the eigenvector with the larger eigenvalue. We use $U^\top$ as a rotation matrix to rotate each sample point from the (X_1, X_2) coordinate system to a coordinate system aligned with the eigenvectors. (As $U^\top = U^{-1}$, the matrix U reverses this rotation, moving back from the eigenvector coordinate system to the original coordinate system). *Center* your sample points by subtracting the mean μ from each point; then rotate each point by $U^\top$, giving $x_{\text{rotated}} = U^\top(x - \mu)$. Plot these rotated points on a new two dimensional-grid, again with both axes having range $[-15, 15]$. (You are not required to plot the eigenvectors, which would be horizontal and vertical.)

In your plots, **clearly label the axes and include a title**. Moreover, **make sure the horizontal and vertical axis have the same scale!** The aspect ratio should be one.

Question 4: Eigenvectors of the Gaussian Covariance Matrix

Part 1:

```
Mean: [3.18174856 5.66142143]
```

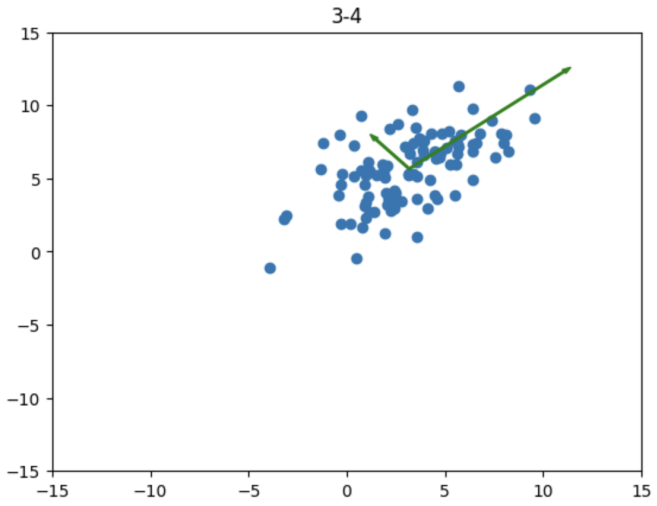
Part 2:

```
Covariance matrix:
 [[7.12274112 3.78326186]
 [3.78326186 5.82706493]]
```

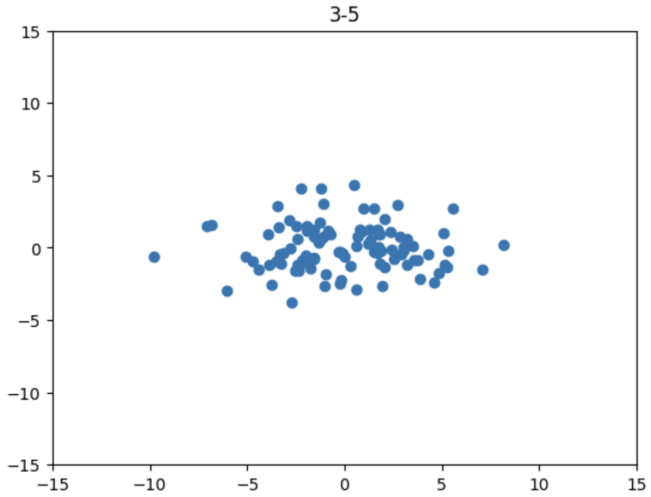
Part 3:

```
Eigenvectors:
 [[ 0.76445448 -0.64467771]
 [ 0.64467771  0.76445448]]
Eigenvalues:
 [10.31323137  2.63657468]
```

Part 4:



Part 5:


3-5
15
10
5
0
−5
−10
−15
−15
−10
−5
0
5
10
15

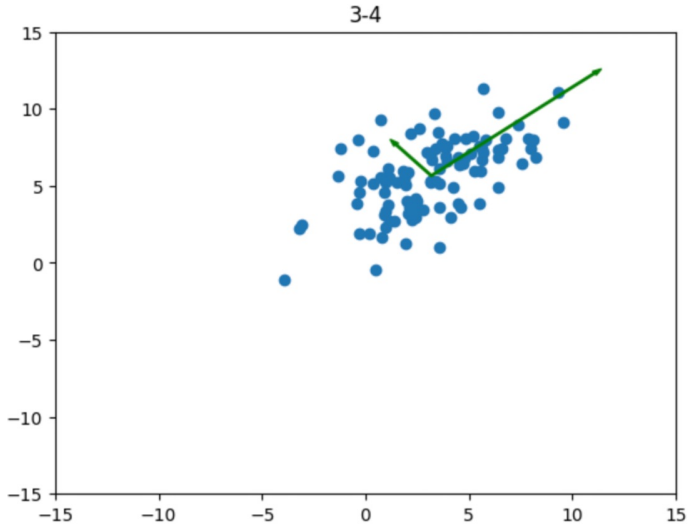
(4) $X_1 \sim N(3,9)$ and $X_2 \sim \frac{1}{2}X_1 + N(4,4)$. $N(\mu,\sigma^2)$. $n = 100$ random 2D sample pts.

(1) mean = $\begin{bmatrix} 3.1817 & 5.8614 \end{bmatrix}$
(x_1 x_2)

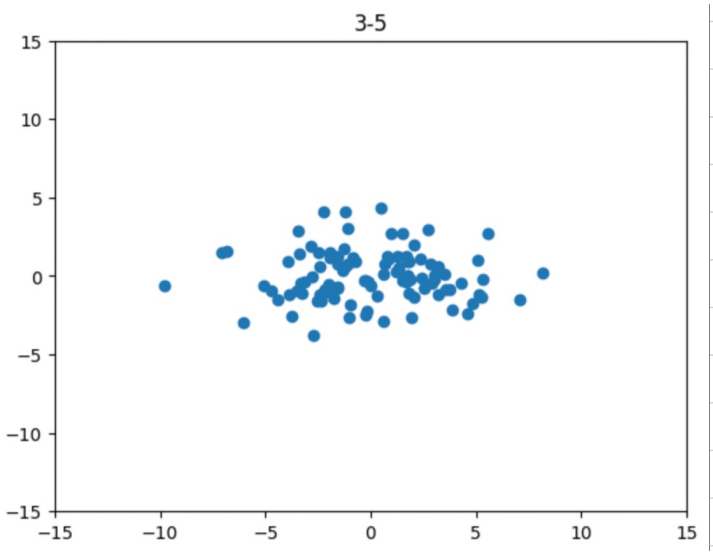
(2) Covariance matrix: $\begin{bmatrix} 7.1227 & 3.7833 \\ 3.7833 & 5.8271 \end{bmatrix}$

(3) Eigen vectors: $\begin{bmatrix} 0.7645 & -0.6447 \\ 0.6447 & 0.7645 \end{bmatrix}$ Eigenvalues: $\begin{bmatrix} 10.3132 & 2.6366 \end{bmatrix}$

(4)



(5)



5 Classification and Risk

Suppose we have a classification problem with classes labeled $1, \ldots, c$ and an additional "doubt" category labeled $c+1$. Let $r : \mathbb{R}^d \to \{1, \ldots, c+1\}$ be a decision rule. Define the loss function

$$L(r(x) = i, y = j) = \begin{cases} 0 & \text{if } i = j \quad i, j \in \{1, \ldots, c\}, \\ \lambda_r & \text{if } i = c+1, \\ \lambda_s & \text{otherwise,} \end{cases}$$

where $\lambda_r \geq 0$ is the loss incurred for choosing doubt, $\lambda_s \geq 0$ is the loss incurred for making a misclassification, and at least one of them is nonzero. Hence the risk of classifying a new data point x as class $i \in \{1, 2, \ldots, c+1\}$ is

$$R(r(x) = i|x) = \sum_{j=1}^{c} L(r(x) = i, y = j)\, P(Y = j|x).$$

To be clear, the actual label Y can *never* be $c+1$.

1. Show that the following predictor obtains the minimum risk when $\lambda_r \leq \lambda_s$:
   - Choose class i if $P(Y = i|x) \geq P(Y = j|x)$ for all j and $P(Y = i|x) \geq 1 - \lambda_r/\lambda_s$;
   - Choose doubt otherwise.
2. What happens if $\lambda_r = 0$? What happens if $\lambda_r > \lambda_s$? Explain why this is consistent with what one would expect intuitively.

6 Maximum Likelihood Estimation and Bias

Let $X_1, \ldots, X_n \in \mathbb{R}$ be n sample points drawn independently from univariate normal distributions such that $X_i \sim \mathcal{N}(\mu, \sigma_i^2)$, where $\sigma_i = \sigma/\sqrt{i}$ for some parameter σ. (Every sample point comes from a distribution with a **different variance**.) Note the word "univariate"; we are working in dimension $d = 1$, and our "points" are real numbers.

1. Derive the maximum likelihood estimates, denoted $\hat{\mu}$ and $\hat{\sigma}$, for the mean μ and the parameter σ. (The formulae from class don't apply here, because every point has a different variance.) You may write an expression for $\hat{\sigma}^2$ rather than $\hat{\sigma}$ if you wish—it's probably simpler that way. Show all your work.
2. Given the true value of a statistic θ and an estimator $\hat{\theta}$ of that statistic, we define the *bias* of the estimator to be the the expected difference from the true value. That is,

   $$bias(\hat{\theta}) = \mathbb{E}[\hat{\theta}] - \theta.$$

   We say that an estimator is *unbiased* if its bias is 0.

   Either prove or disprove the following statement: *The MLE sample estimator $\hat{\mu}$ is unbiased. Hint: Neither the true μ nor true σ^2 are known when estimating sample statistics, thus we need to plug in appropriate estimators.*

5

$$L(r(x)=i, y=j) = \begin{cases} 0 & \text{if } i=j \quad i,j \in \{1,\dots,C\} \\ \lambda_r & \text{if } i=C+1, \\ \lambda_s & \text{otherwise.} \end{cases}$$

$$R(r(x)=i|x) = \sum_{j=1}^{C} L(r(x)=i, y=j)\, P(Y=j|x)$$

1 Assuming that $P(Y=i|x) \geq P(Y=j|x)$ and $P(Y=i|x) \geq 1-\lambda_r/\lambda_s$ and $r(x)=i$.

Then, $R(r(x)=i|x) = \sum_{j=1}^{C} (r(x)=i, y=j)\, P(Y=j|x) = \lambda_s \sum_{j=1, j\neq i} P(Y=j|x) = \lambda_s (1-P(Y=i|x))$

But $f(x)$ can be $\begin{cases} f(x)=k \rightarrow R(f(x)=k|x) = \lambda_s (1-P(Y=k|x)) \rightarrow R(r(x)=i|x) \leq R(f(x)=k|x) \\ f(x)=C+1 \rightarrow R(f(x)=k|x) = \lambda_r \rightarrow R(r(x)=i|x) \leq R(f(x)=k|x) \end{cases}$

Now, assuming that we choose doubt and $r(x)=C+1$.

Then, $R(r(x)=i|x) = \lambda_r$

But $f(x)=k \rightarrow R(f(x)=k|x) = \lambda_s (1-P(Y=k|x)) \rightarrow$ The first bullet doesn't hold

$$\rightarrow \max_{j\in\{1,\dots,C\}} P(Y=j|x) < 1-\frac{\lambda_r}{\lambda_s} \rightarrow P(Y=k|x) < 1-\frac{\lambda_r}{\lambda_s} \rightarrow R(r(x)=i|x) < R(f(x)=k|x)$$

In every case the predictor obtains the minimum risk when $\lambda_r \leq \lambda_s$. ✓

---

2 If $\lambda_r=0$, then we have the first case where $P(r(x)=i|x)=1$ only when

SO: $\begin{cases} x \text{ in class } i & \underline{100\% \text{ sure}} \\ \text{choose doubt} & \underline{\text{otherwise}} \end{cases}$ } consistent with what we expect intuitively because if we choose doubt $\lambda_r=0$ and therefore we won't have any penalty.

If $\lambda_r > \lambda_s$, then x is <u>always</u> classified as class i. So, we have the highest probability of correctly classifying.

This is also consistent with what expect intuitively because the cost of choosing doubt is higher that classifying, so the best option is to just always classify it in order to get the highest probability.

(6) $X_1, \dots, X_n \in \mathbb{R}$ are sampled from univariate normal distributions: $X_i \sim N(\mu, \frac{\sigma}{\sqrt{i}})$. where every sample comes from a different variance.

(1) Deriving the MLE: $\hat{\mu}$ and $\hat{\sigma}$:

want $\underset{\theta}{\operatorname{argmax}}\, L(X_1, X_2, \dots, X_n \mid \theta) = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{n} L(X_i \mid \theta) = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{n} \log\left(L(X_i \mid \theta)\right)$

$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(X_i-\mu)^2}{2\sigma^2}}$

therefore: $\underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{n} -\frac{1}{2}\log 2\pi\sigma^2 - \frac{(X_i-\mu)^2}{2\sigma^2}$

to solve for $\hat{\theta}$, take derivative and set to 0: Assuming this is convex

$$\nabla_\mu \left(\sum_{i=1}^{n} \log i - \frac{1}{2}\log 2\pi\sigma^2 - \frac{i(\mu - X_i)^2}{2\sigma^2}\right) = \sum_{i=1}^{n} \frac{-i}{2\sigma^2}\left(2(\mu - X_i)\right) = \sum_{i=1}^{n} \frac{-i(\mu - X_i)}{\sigma^2} = \sum_{i=1}^{n} i(X_i - \mu) = 0$$

$$\hat{\mu} = \frac{\sum_{i=1}^{n} X_i\, i}{n(n+1)} = \frac{\sum_{i=1}^{n} X_i\, i}{\sum_{i=1}^{n} i}$$

$$\nabla_{\sigma^2}\left(\sum_{i=1}^{n} \log i - \frac{1}{2}\log 2\pi\sigma^2 - \frac{i(X_i-\mu)^2}{2\sigma^2}\right) = \sum_{i=1}^{n} -\frac{1}{2\sigma^2} + \frac{i}{2\sigma^4}(X_i - \mu)^2$$

$$= \frac{1}{2\sigma^2}\left(\left(\sum_{i=1}^{n} \frac{i}{\sigma^2}(X_i-\mu)^2\right) - n\right) = 0 \rightarrow \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n} i(X_i - \hat{\mu})^2$$

($\hat{\mu} \rightarrow \frac{1}{n}\sum_{i=1}^{n} X_i$)

---

(2) $\text{bias}(\hat{\theta}) = E[\hat{\theta}] - \theta$. "The MLE sample estimator $\hat{\mu}$ is unbiased." Prove? Disprove?

The estimator is unbiased if its bias is 0. so,

$$\text{bias}(\check{\mu}) = E[\check{\mu}] - \mu = E\left[\frac{\sum_{i=1}^{n} X_i\, i}{\sum_{i=1}^{n} i}\right] - \mu = \frac{\sum_{i=1}^{n} E[X_i]\, i}{\sum_{i=1}^{n} i} - \mu = \frac{\sum_{i=1}^{n} \mu i}{\sum_{i=1}^{n} i} - \mu = \mu\frac{\sum_{i=1}^{n} i}{\sum_{i=1}^{n} i} - \mu = \mu - \mu = 0 \checkmark$$

Therefore proven the statement.

---

(3) $\text{bias}(\hat{\theta}) = E[\hat{\theta}] - \theta$. "The MLE sample estimator $\hat{\sigma}^2$ is unbiased." Prove? Disprove?

The estimator is unbiased if its bias is 0. so,

$\text{bias}(\check{\sigma}^2) = E[\check{\sigma}^2] - \sigma^2$ from part a) $\check{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(X_i - \check{\mu})^2 i$

$$= E\left[\frac{1}{n}\sum_{i=1}^{n}(X_i - \check{\mu})^2 i\right] - \sigma^2 = \frac{1}{n}E\left[\sum_{i=1}^{n} X_i^2 i + \sum_{i=1}^{n}\check{\mu}^2 i - 2\check{\mu}\sum_{i=1}^{n} X_i\, i\right] - \sigma^2$$

$$= \frac{1}{n}\left[\sum_{i=1}^{n} E[X_i^2]\, i - 2E[\check{\mu}^2]\right] - \sigma^2 = \frac{1}{n}\left[n\sigma^2 + 2\mu^2 - \sigma^2 - 2\mu^2\right] - \sigma^2 = \frac{n-1}{n}\sigma^2 - \sigma^2 = -\frac{1}{n}\sigma^2 \neq 0$$ so disproved the statement.

3. Either prove or disprove the following statement: *The MLE sample estimator $\hat{\sigma^2}$ is unbiased. Hint: Neither the true μ nor true σ^2 are known when estimating sample statistics, thus we need to plug in appropriate estimators.*

7 Covariance Matrices and Decompositions

As described in lecture, the covariance matrix $\text{Var}(R) \in \mathbb{R}^{d \times d}$ for a random variable $R \in \mathbb{R}^d$ with mean $\mu \in \mathbb{R}^d$ is

$$\text{Var}(R) = \text{Cov}(R, R) = \mathbb{E}[(R - \mu)(R - \mu)^\top] = \begin{bmatrix} \text{Var}(R_1) & \text{Cov}(R_1, R_2) & \dots & \text{Cov}(R_1, R_d) \\ \text{Cov}(R_2, R_1) & \text{Var}(R_2) & & \text{Cov}(R_2, R_d) \\ \vdots & & \ddots & \vdots \\ \text{Cov}(R_d, R_1) & \text{Cov}(R_d, R_2) & \dots & \text{Var}(R_d) \end{bmatrix},$$

where $\text{Cov}(R_i, R_j) = \mathbb{E}[(R_i - \mu_i)(R_j - \mu_j)]$ and $\text{Var}(R_i) = \text{Cov}(R_i, R_i)$.

If the random variable R is sampled from the multivariate normal distribution $\mathcal{N}(\mu, \Sigma)$ with the PDF

$$f(x) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} e^{-((x-\mu)^\top \Sigma^{-1}(x-\mu))/2},$$

then $\text{Var}(R) = \Sigma$.

Given n points $X_1, X_2, \dots, X_n$ sampled from $\mathcal{N}(\mu, \Sigma)$, we can estimate Σ with the maximum likelihood estimator

$$\hat{\Sigma} = \frac{1}{n} \sum_{i=1}^{n} (X_i - \mu)(X_i - \mu)^\top,$$

which is also known as the *sample covariance matrix*.

1. The estimate $\hat{\Sigma}$ makes sense as an approximation of Σ only if $\hat{\Sigma}$ is invertible. Under what circumstances is $\hat{\Sigma}$ not invertible? Make sure your answer is complete; i.e., it includes all cases in which the covariance matrix of the sample is singular. Express your answer in terms of the geometric arrangement of the sample points X_i.

2. Suggest a way to fix a singular covariance matrix estimator $\hat{\Sigma}$ by replacing it with a similar but invertible matrix. Your suggestion may be a kludge, but it should not change the covariance matrix too much. Note that infinitesimal numbers do not exist; if your solution uses a very small number, explain how to calculate a number that is sufficiently small for your purposes.

3. Consider the normal distribution $\mathcal{N}(0, \Sigma)$ with mean $\mu = 0$. Consider all vectors of length 1; i.e., any vector x for which $\|x\| = 1$. Which vector(s) x of length 1 maximizes the PDF $f(x)$? Which vector(s) x of length 1 minimizes $f(x)$? Your answers should depend on the properties of Σ. Explain your answer.

4. Suppose we have $X \sim \mathcal{N}(0, \Sigma)$, $X \in \mathbb{R}^n$ and a unit vector $y \in \mathbb{R}^n$. We can compute the projection of the random vector X onto a direction y as $p = y^\top X$. First, compute the variance of p. Second, with this information, what does the eigenvector corresponding to the

7

$$\mathrm{Var}(R) = \mathrm{Cov}(R,R) = \mathbb{E}[(R-\mu)(R-\mu)^\top] = \begin{bmatrix} \mathrm{Var}(R_1) & \mathrm{Cov}(R_1,R_2) & \dots & \mathrm{Cov}(R_1,R_d) \\ \mathrm{Cov}(R_2,R_1) & \mathrm{Var}(R_2) & & \mathrm{Cov}(R_2,R_d) \\ \vdots & & \ddots & \vdots \\ \mathrm{Cov}(R_d,R_1) & \mathrm{Cov}(R_d,R_2) & \dots & \mathrm{Var}(R_d) \end{bmatrix}, \quad f(x) = \frac{1}{\sqrt{(2\pi)^d|\Sigma|}} e^{-((x-\mu)^\top \Sigma^{-1}(x-\mu))/2},$$

$$\hat{\Sigma} = \frac{1}{n}\sum_{i=1}^{n}(X_i-\mu)(X_i-\mu)^\top,$$

(1) when is $\hat{\Sigma}$ not invertible?

When all points lie on the same hyperplane in the feature space.

(2) To fix a singular covariance matrix estimator $\hat{\Sigma}$:

To fix and make sure we don't have singular covariances, we can do $\hat{\Sigma} = \hat{\Sigma} + \alpha I$ where α is some small number on the order of $10^{-2}, 10^{-6}$. we do this so that all of the eigenvalues of the covariance matrix are positive. This will make sure that the covariance matrix we have is invertible. we did this in Question 8 to fix the error of encountering singular covariance matrices.

(3) $N(0,\Sigma)$ and $\mu = 0$ which vectors x where $\|x\| = 1$ maximize the PDF $f(x)$?

The vector x that $\|x\| = 1$ maximizes PDF when it's the eigenvector of Σ that has the largest eigen value.

which vectors x where $\|x\| = 1$ minimize the PDF $f(x)$?

The vector x that $\|x\| = 1$ minimizes PDF when it's the eigenvector of Σ that has the smallest eigen value.

(4) $X \sim N(0,\Sigma) \in \mathbb{R}^n$ and unit vector $y \in \mathbb{R}^n$. $p = y^T x$. Compute variance of p:

$\mathrm{Var}(p) = \mathrm{var}(y^T x) = y^T \mathrm{cov}(x) y = y^T \Sigma y \qquad \|y\| = 1$

what does the eigenvector tell us about variances of $y^T x$?

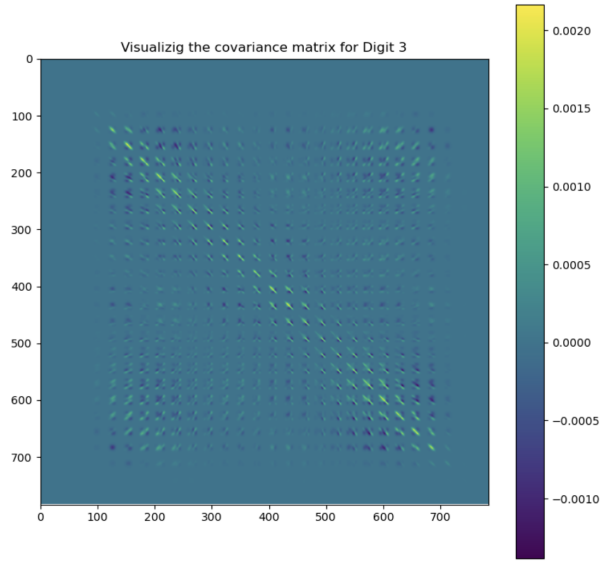
This shows that $\mathrm{var}(p) \le \lambda_{max}(\Sigma)$ as $\|\Sigma y\| \le \lambda_{max}(\Sigma)\|y\| = \lambda_{max}(\Sigma)$

Question 8: Gaussian Classifiers for Digits and Spam

Part 1:

```
Fitting Gaussian Distribution:
Mean: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Length of mean of digit 5: 784
Covariance: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Length of covariance of digit 7: 784
```

Part 2:



How do the diagonal terms compare to off-diagonal terms?

The covariances on the diagonal are brighter than the covariances that are not on the diagonal which means that they have a higher covariance value on the diagonal. The terms on the diagonal should be a higher value since the values on the diagonal are where the covariance is 1 (since it's the covariance of the same sample with itself). Everywhere else is less than or equal to 1 because it's the covariance between 2 different samples.

Part 3:

a) LDA:

b) QDA:

c) Which was better?

LDA performed better because generally LDA does better on larger datasets. Also, looking at their error rate graphs we can see that the validation accuracy for QDA decreases as our number of training points increases but LDA's validation increases as we increase the number of training points. The error graph also shows that LDA has a very smooth decreasing error rate but QDA does not and has some bumps before it starts to also decrease. ALso, QDA on larger datasets can cause overfitting.

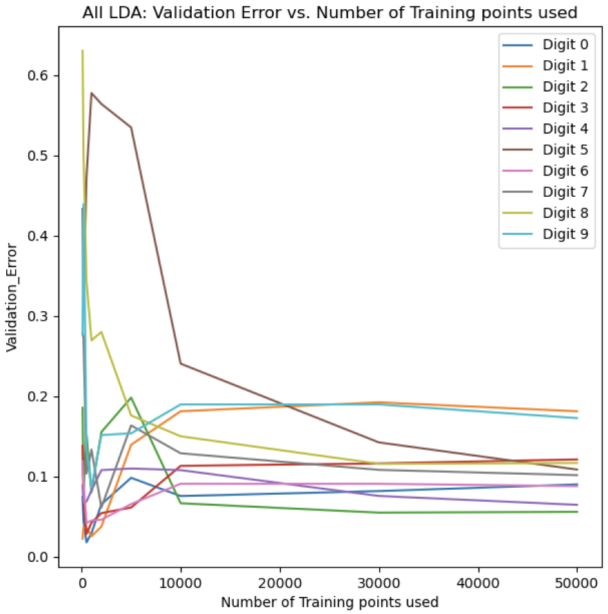
LDA:

```
LDA:
Training with 100 points
Validation accuracy: 0.6767

Training with 200 points
Validation accuracy: 0.6451

Training with 500 points
Validation accuracy: 0.5688

Training with 1000 points
Validation accuracy: 0.7404

Training with 2000 points
Validation accuracy: 0.8132

Training with 5000 points
Validation accuracy: 0.8541

Training with 10000 points
Validation accuracy: 0.86

Training with 30000 points
Validation accuracy: 0.8704

Training with 50000 points
Validation accuracy: 0.8707
```

QDA:

```
QDA:
Training with 100 points
Validation acc: 0.7803

Training with 200 points
Validation acc: 0.812

Training with 500 points
Validation acc: 0.8657

Training with 1000 points
Validation acc: 0.8694

Training with 2000 points
Validation acc: 0.8545

Training with 5000 points
Validation acc: 0.8345

Training with 10000 points
Validation acc: 0.8668

Training with 30000 points
Validation acc: 0.8833

Training with 50000 points
Validation acc: 0.8898
```

d) Plot all 10 curves:



Which digit is easiest to classify?

Digit 0 and 1 are the easiest digits to classify. I found these to be the easiest to classify because they have the most unique features as other digits. Looking at their LDA Loss graph as well, they have the smallest minimum value on the all digits LDA graph.

Part 4: Kaggle Submission

Kaggle username: **Hiva Mohammadzadeh**
Kaggle Scores:

a) MNIST: 0.87933 = 88%

Part 5: LDA and QDA on SPAM

Kaggle username: **Hiva Mohammadzadeh**
Kaggle Scores:

a) SPAM: 0.784 = 78%

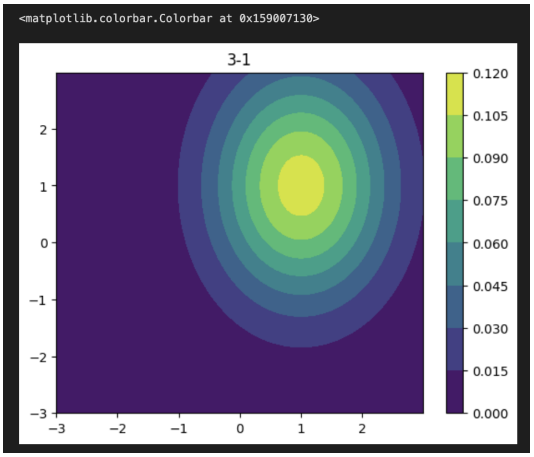
CODE APPENDIX:

Question 3: Isocontours of Normal Distributions

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
```

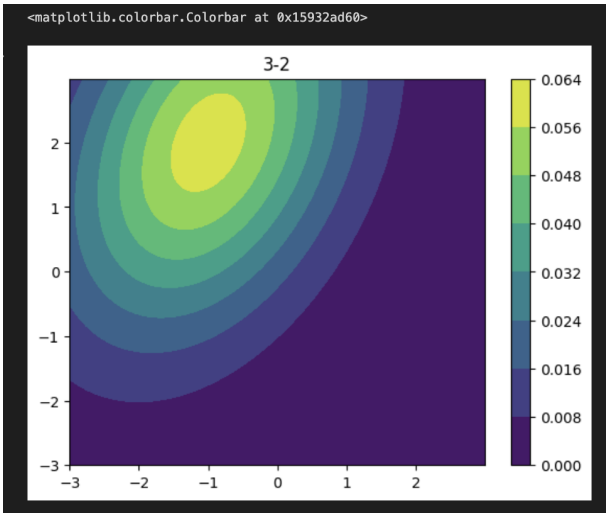
Part 1:

```
np.random.seed(0)
f = multivariate_normal([1,1], [[1,0],[0,2]])
x_domain, y_domain = np.mgrid[-3:3:.01, -3:3:.01]
axises = np.dstack((x_domain, y_domain))
plt.figure(0)
plt.title("3-1")
plt.contourf(x_domain, y_domain, f.pdf(axises))
plt.colorbar()
```



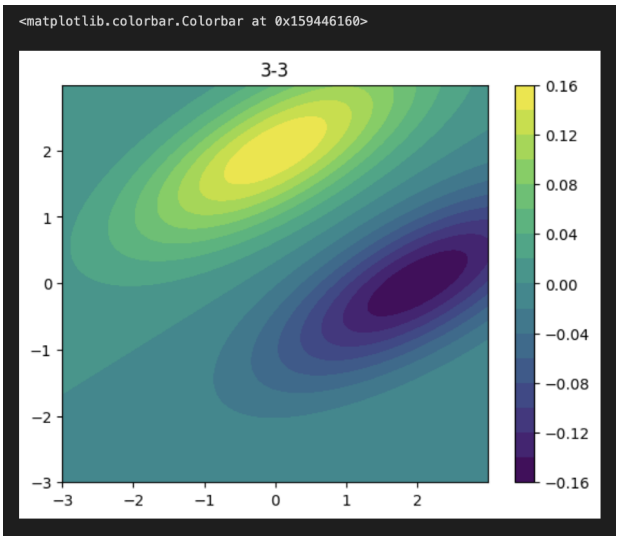
Part 2:

```
f = multivariate_normal([-1,2], [[2,1],[1,4]])
x_domain, y_domain = np.mgrid[-3:3:.01, -3:3:.01]
axises = np.dstack((x_domain, y_domain))
plt.figure(1)
plt.title("3-2")
plt.contourf(x_domain, y_domain, f.pdf(axises))
plt.colorbar()
```



Part 3:

```
f1 = multivariate_normal([0,2], [[2,1],[1,1]])
f2 = multivariate_normal([2,0], [[2,1],[1,1]])
x_domain, y_domain = np.mgrid[-3:3:.01, -3:3:.01]
axises = np.dstack((x_domain, y_domain))
plt.figure(2)
plt.title("3-3")
plt.contourf(x_domain, y_domain, f1.pdf(axises) - f2.pdf(axises), 20)
plt.colorbar()
```

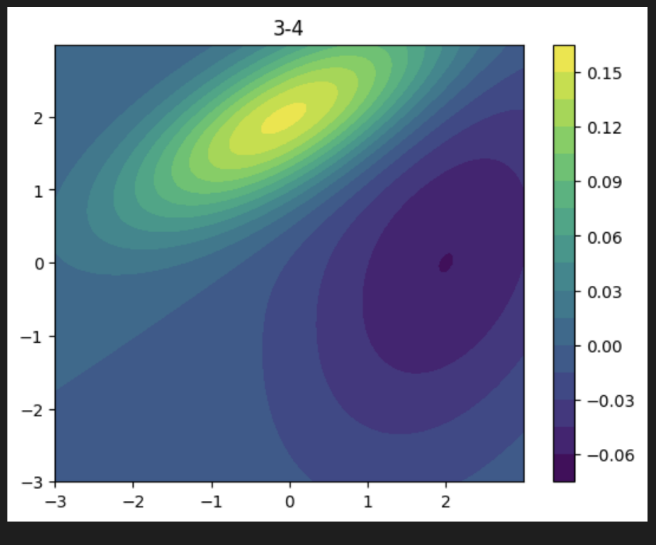


Part 4:

```
f1 = multivariate_normal([0,2], [[2,1],[1,1]])
f2 = multivariate_normal([2,0], [[2,1],[1,4]])
x_domain, y_domain = np.mgrid[-3:3:.01, -3:3:.01]
axises = np.dstack((x_domain, y_domain))
plt.figure(3)
```

```
plt.title("3-4")

plt.contourf(x_domain, y_domain, f1.pdf(axises) - f2.pdf(axises), 20)

plt.colorbar()
```

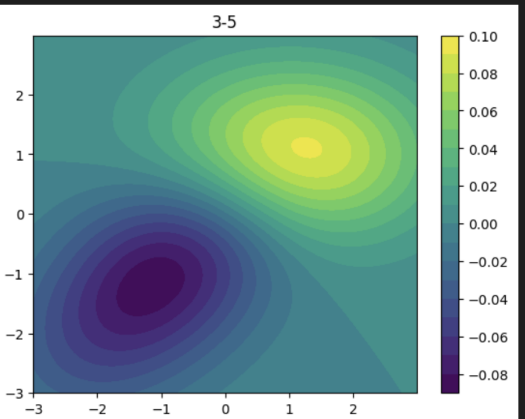
```
<matplotlib.colorbar.Colorbar at 0x159551550>
```



Part 5:

```
f1 = multivariate_normal([1,1], [[2,0],[0,1]])

f2 = multivariate_normal([-1,-1], [[2,1],[1,2]])

x_domain, y_domain = np.mgrid[-3:3:.01, -3:3:.01]

axises = np.dstack((x_domain, y_domain))

plt.figure(4)

plt.title("3-5")

plt.contourf(x_domain, y_domain, f1.pdf(axises) - f2.pdf(axises), 20)

plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x15965fa00>
```



Question 4: Eigenvectors of the Gaussian Covariance Matrix

```
x1 = multivariate_normal.rvs(mean=3.0, cov=9.0, size=100, random_state=1)

x2 = 0.5 * x1 + multivariate_normal.rvs(mean=4.0, cov=4.0, size=100,
random_state=4)

position_vectors = np.vstack((x1, x2))
```

Part 1:

```
## Part 1:

mean = np.mean(position_vectors,axis=0)

print("Mean:",mean)
```

```
Mean: [3.18174856 5.66142143]
```

Part 2:

```
## Part 2:

covariance = np.cov(position_vectors)

print("Covariance matrix:\n",covariance)
```

```
Covariance matrix:
 [[7.12274112 3.78326186]
 [3.78326186 5.82706493]]
```

Part 3:

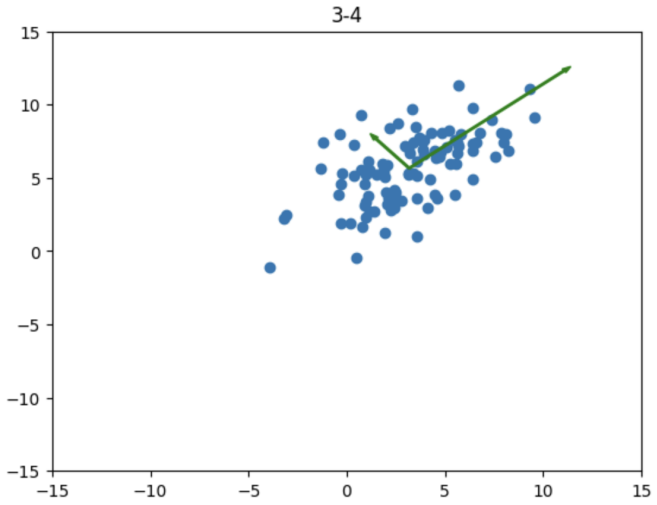
```
## Part 3:

eigenvalues, eigenvectors = np.linalg.eig(covariance)

print("Eigenvectors:\n",eigenvectors)

print("Eigenvalues:\n",eigenvalues)
```

```
Eigenvectors:
 [[ 0.76445448 -0.64467771]
 [ 0.64467771  0.76445448]]
Eigenvalues:
 [10.31323137  2.63657468]
```

Part 4:

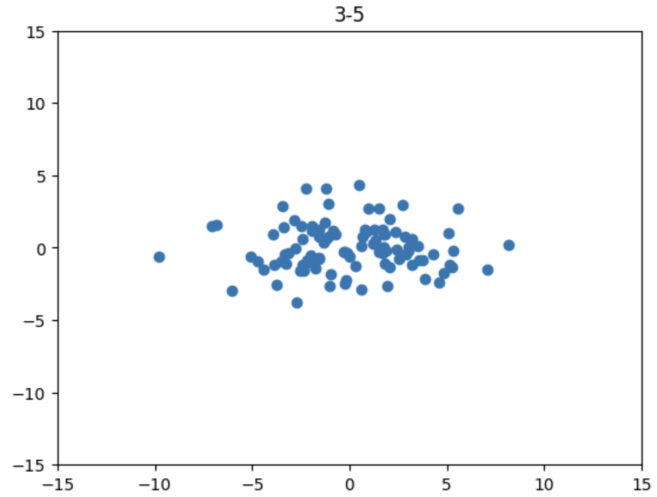
```
## Part 4:

plt.axis((-15,15,-15,15))

plt.title("3-4")

plt.scatter(position_vectors[0],position_vectors[1])

plt.arrow(*mean, *(eigenvalues[1]*eigenvectors[:,1]), color='green', width=0.1)
```

```
plt.arrow(*mean, *(eigenvalues[0]*eigenvectors[:,0]), color='green', width=0.1)
```



Part 5:

```
## Part 5:

rotated_points = eigenvectors.T @ (position_vectors.T - mean).T

plt.axis((-15,15,-15,15))

plt.title("3-5")

plt.scatter(rotated_points[0],rotated_points[1]);
```



Question 8: Gaussian Classifiers for Digits and Spam

```
import sys
if sys.version_info[0] < 3:
    raise Exception("Python 3 not detected.")
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from scipy import io
from save_csv import results_to_csv

if __name__ == "__main__":
    for data_name in ["mnist", "spam"]:
        data = np.load(f"../data/{data_name}-data-hw3.npz")
        print("\nloaded %s data!" % data_name)
        fields = "test_data", "training_data", "training_labels"
        for field in fields:
            print(field, data[field].shape)
```

```
loaded mnist data!
test_data (10000, 1, 28, 28)
training_data (60000, 1, 28, 28)
training_labels (60000,)

loaded spam data!
test_data (1000, 32)
training_data (4172, 32)
training_labels (4172,)
```

```
###### QUESTION 8: Gaussian Classifiers for Digits and SPAM
```

Part 1:

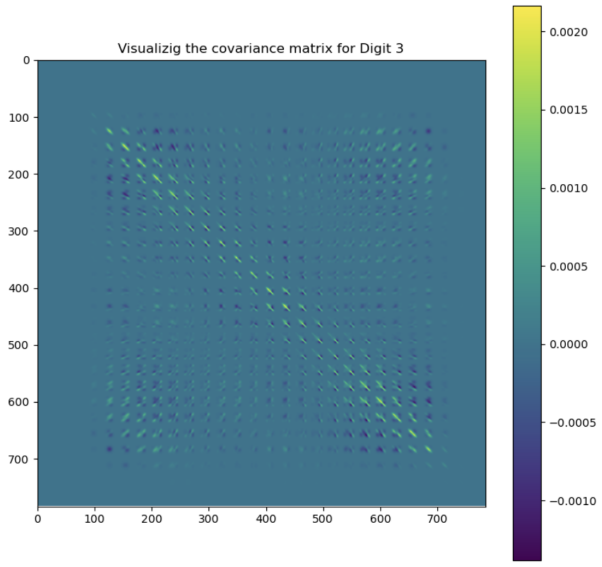
```
### Question 1 and 2
    # Load the MNIST training data
    print("\nMNIST:")
    data = np.load(f"../data/mnist-data-hw3.npz")
    mnist_training_data = data["training_data"]
    mnist_training_labels = data["training_labels"]
    # Reshape them to match
    mnist_training_data = np.reshape(mnist_training_data, (60000, 784))
    mnist_training_labels = np.reshape(mnist_training_labels,(60000, 1))
    # Check the shapes
    print(f"\nData size: {mnist_training_data.shape}")
    print(f"Labels size: {mnist_training_labels.shape}")

    # Contrast Normalizing the Mnist image data before using their values by
dividing it by its norm
    mnist_training_data = mnist_training_data / np.linalg.norm(mnist_training_data,
axis=1)[:, None]

    # Fitting a Gaussian distribution to each digit using MLE
    samples_for_mean = {}
    samples_for_covariance = {}

    # Computing a mean and a covariance matrix for each digit class
    for digit in list(np.unique(mnist_training_labels)):

        digits_data = mnist_training_data[mnist_training_labels.reshape(-1,) ==
digit, :]
        # print(digits_data.shape)
        mean = digits_data.mean(axis=0)
        covariance = np.cov(digits_data.T)
        # Set the mean and covariance of each digit.
        samples_for_mean[digit] = mean
        samples_for_covariance[digit] = covariance
```

```
   # Print and check the shapes
   print(f"\nFitting Gaussian Distribution: \nMean: {samples_for_mean.keys()}
\nLength of mean of digit 5: {len(samples_for_mean[5])}")
   print(f"Covariance: {samples_for_covariance.keys()} \nLength of covariance of
digit 7: {len(samples_for_covariance[7])}")
```

```
MNIST:

Data size: (60000, 784)
Labels size: (60000, 1)

Fitting Gaussian Distribution:
Mean: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Length of mean of digit 5: 784
Covariance: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Length of covariance of digit 7: 784
```

Part 2:

```
### Question 2: Visualising the covariance matrix for a particular class (I chose
the digit 3)

   figure = plt.figure(0)
   # print(samples_for_covariance.keys())
   plt.title("Visualizig the covariance matrix for Digit 3")
   plt.imshow(samples_for_covariance[3])
   plt.colorbar()
   # plt.show()
   plt.savefig('MNIST_visualization_of_covariance.png')
```



The covariances on the diagonal are brighter than the covariances that are not on the diagonal which means that they have a higher covariance value on the diagonal. The terms on the diagonal should be a higher value since the values on the diagonal are where the covariance is 1 (since it's the covariance of the same sample with itself). Everywhere else is less than or equal to 1 because it's the covariance between 2 different samples.

Part 3:

```
### Question 3: Classification of Digits
```

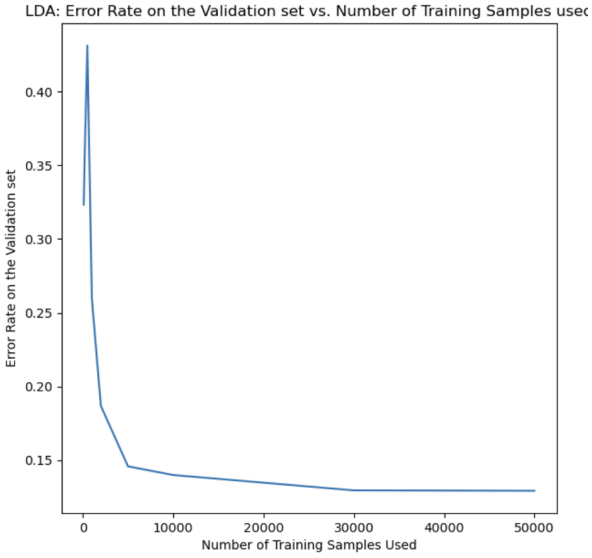
```
   ## Partition the data just like HW1:
   # Concatenating the labels with data in order to shuffle better
   concatenated_data = np.concatenate((mnist_training_data, mnist_training_labels),
axis=1)
   np.random.shuffle(concatenated_data)

   # Split using the amount we want in validation set
   amount_set_aside = 10000
   mnist_validation, mnist_training = concatenated_data[0:amount_set_aside, :],
concatenated_data[amount_set_aside:, :]

   mnist_validation_data = mnist_validation[:, :-1]
   mnist_validation_labels = mnist_validation[:, -1:]

   mnist_training_data = mnist_training[:, :-1]
   mnist_training_labels = mnist_training[:, -1:]

   #Print the datasets' shapes
   print(f"\nPartitioned: \nTraining data: {mnist_training_data.shape} \nTraining
labels: {mnist_training_labels.shape}")
   print(f"Validation data: {mnist_validation_data.shape} \nValidation labels:
{mnist_validation_labels.shape}")


   ## Normalizing the data
   training_normalized = np.sqrt((mnist_training_data ** 2).sum(axis = 1))
   mnist_training_data = mnist_training_data / training_normalized.reshape(-1, 1)

   validation_normalized = np.sqrt((mnist_validation_data ** 2).sum(axis = 1))
   mnist_validation_data = mnist_validation_data /
validation_normalized.reshape(-1, 1)

   mnist_testing_data = np.reshape(data["test_data"], (10000, 784))
   testing_normalized = np.sqrt((mnist_testing_data ** 2).sum(axis = 1))
   mnist_testing_data = mnist_testing_data / testing_normalized.reshape(-1, 1)
   # checking shapes
   print(f"\nAfter Normalization: \nTraining data: {mnist_training_data.shape}")
   print(f"Validation data: {mnist_validation_data.shape}")
   print(f"Testing data: {mnist_testing_data.shape}")
```

```
Partitioned:
Training data: (50000, 784)
Training labels: (50000, 1)
Validation data: (10000, 784)
Validation labels: (10000, 1)

After Normalization:
Training data: (50000, 784)
Validation data: (10000, 784)
Testing data: (10000, 784)
```

A)

```
### Question 8-3: Part a) Doing LDA for MNIST:

   training_points = [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000]
   validation_accuracy_dict = {"validation": []}
   predictions = []

   ## Function to train LDA for the MNIST
   def Mnist_lda(train_data, train_labels):
   #     print(train_labels.shape)
   #     print(train_labels.shape)
```

```
        a,b = train_data.shape
        priors = np.zeros((10,1))
        covariance = np.zeros((b,b)).astype(np.float32)
        samples_for_mean = np.zeros((10, b))
        train_labels = train_labels.reshape(-1,)

        ## Calculates prior, mean, and covariance for each digit
        for digit in range(0, 10):

            priors[int(digit)] = (train_labels == digit).astype(np.int32).sum() / a

            data = train_data[train_labels == digit, :]

            samples_for_mean[int(digit), :] = data.mean(axis=0)
            covariance += np.cov(data.T)

        covariance /= 10
        return priors, samples_for_mean, covariance

    # Training the LDA Model for MNIST for each training point
    print("\nLDA:")
    for training_point in training_points:
        print(f"Training with {training_point} points")

        training_data = mnist_training_data[:training_point, :]
        training_labels = mnist_training_labels[:training_point, :]

        priors, means, covariance = Mnist_lda(training_data, training_labels)
        # Fixing and preventing the singular covariance matrices. I just add a very
small value (1e-6) to the diagonal values of the covariance to make the eigenvalues
positive.
        covariance = covariance + (1e-6) * np.eye(*covariance.shape)

        validation_num = mnist_validation_data.shape[0]
        validation_out = np.zeros((validation_num, 10))

        # Calculate mean, prior and covariance for each digit
        for digit in range(0, 10):
            prior = priors[digit]
            mean = means[digit, :].reshape(-1, 1)

            weight = np.linalg.solve(covariance.T, mean)
            alpha = -0.5 * weight.T.dot(mean) + np.log(prior)

            validation_out[:, digit] = (mnist_validation_data.dot(weight) +
alpha).reshape(-1,)
        # Calculate validation predictions for all digits
        validation_prediction = np.argmax(validation_out, axis=1).reshape(-1, 1)

        predictions.append(validation_prediction)
        validation_accuracy_dict["validation"].append((mnist_validation_labels ==
validation_prediction).mean())

        print(f"Validation accuracy:
{validation_accuracy_dict['validation'][-1]}\n")
        print()


    figure = plt.figure(1)
    plt.plot(training_points, [1 - x for x in
validation_accuracy_dict["validation"]])
    plt.title("LDA: Error Rate on the Validation set vs. Number of Training Samples
used")
    plt.xlabel("Number of Training Samples Used")
    plt.ylabel("Error Rate on the Validation set")
    # plt.show()
    plt.savefig('LDA.png')
```




```
LDA:
Training with 100 points
Validation accuracy: 0.6334


Training with 200 points
Validation accuracy: 0.6434


Training with 500 points
Validation accuracy: 0.5534


Training with 1000 points
Validation accuracy: 0.7283


Training with 2000 points
Validation accuracy: 0.8116


Training with 5000 points
Validation accuracy: 0.846


Training with 10000 points
Validation accuracy: 0.8585


Training with 30000 points
Validation accuracy: 0.8648


Training with 50000 points
Validation accuracy: 0.8679
```

B)

```
### Question 8-3: Part b) Doing QDA for MNIST:
   training_points = [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000]
   validation_accuracy_dict = {"validation": []}
   predictions = []

   ## Function to train QDA for the MNIST
   def mnist_qda(train_data, train_labels):
       n,d = train_data.shape
       priors = np.zeros((10,1))
       covariances = np.zeros((10,d,d)).astype(np.float32)
       samples_for_mean = np.zeros((10, d))

       digits = list(np.unique(train_labels))
       # Calculate prior, mean, and covariance for each digit
       for digit in digits:
           priors[int(digit)] = ((train_labels == digit).astype(np.int32).sum() /
n)

           data = train_data[train_labels == digit, :]

           samples_for_mean[int(digit),:] = data.mean(axis=0)
           covariances[int(digit), :, :] = np.cov(data.T)

       return priors, samples_for_mean, covariances

   # Training QDA on MNIST
   print("\nQDA: ")
   for training_point in training_points:
       print(f"Training with {training_point} points")

       train_data = mnist_training_data[:training_point, :]
       train_labels = mnist_training_labels[:training_point, :].reshape(-1,)

       priors, means, covariances = mnist_qda(train_data, train_labels)
```

```
        validation_num = mnist_validation_data.shape[0]
        validation_out = np.zeros((validation_num, 10))

        train_num = mnist_training_data.shape[0]
        train_out = np.zeros((train_num, 10))

        validation_num = mnist_validation_data.shape[0]
        validation_out = np.zeros((validation_num, 10))
        # calculate mean, prior and covariance of a single digit and finding the
validation prediction
        for digit in range(0, 10):
            prior = priors[digit]
            mean = means[digit, :]
            covariance = covariances[digit, :, :]

        # Fixing and preventing the singular covariance matrices. I just add a very
small value (1e-6) to the diagonal values of the covariance to make the eigenvalues
positive.
            covariance = covariance + (1e-6) * np.eye(*covariance.shape)
            alpha = -0.5 * np.linalg.det(covariance) + prior

            mean_centered_validation = mnist_validation_data - mean
            validation_weight = np.linalg.solve(covariance,
mean_centered_validation.T)

            a, b = mean_centered_validation.shape
            out_diagonal = np.array([mean_centered_validation[n, :].reshape(1,
-1).dot(validation_weight[:, n].reshape(-1, 1)) for n in range(a)])

            validation_prediction = -0.5 * out_diagonal + alpha
            validation_out[:, digit] =
validation_prediction.reshape(validation_num,)

        validation_prediction_final = np.argmax(validation_out, axis=1).reshape(-1,
1)

        predictions.append(validation_prediction_final)
        validation_accuracy_dict["validation"].append((mnist_validation_labels ==
validation_prediction_final).mean())

        print(f"Validation acc: {validation_accuracy_dict['validation'][-1]}\n")


    figure = plt.figure(2)
    plt.plot(training_points, [1 - x for x in
validation_accuracy_dict["validation"]])
    plt.title("QDA: Error Rate on the Validation set vs. Number of Training Samples
used")
    plt.xlabel("Number of Training Samples Used")
    plt.ylabel("Error Rate on the Validation set")
    # plt.show()
    plt.savefig('QDA.png')
```


```
QDA:
Training with 100 points
Validation acc: 0.7529

Training with 200 points
Validation acc: 0.8232

Training with 500 points
Validation acc: 0.8831

Training with 1000 points
Validation acc: 0.9071

Training with 2000 points
Validation acc: 0.88

Training with 5000 points
Validation acc: 0.8262

Training with 10000 points
Validation acc: 0.8641

Training with 30000 points
Validation acc: 0.8805

Training with 50000 points
Validation acc: 0.8815
```

C) Which was better?

LDA performed better because generally LDA does better on larger datasets. Also, looking at their error rate graphs we can see that the validation accuracy for QDA decreases as our number of training points increases but LDA's validation increases as we increase the number of training points. The error graph also shows that LDA has a very smooth decreasing error rate but QDA does not and has some bumps before it starts to also decrease. ALso, QDA on larger datasets can cause overfitting.

LDA:

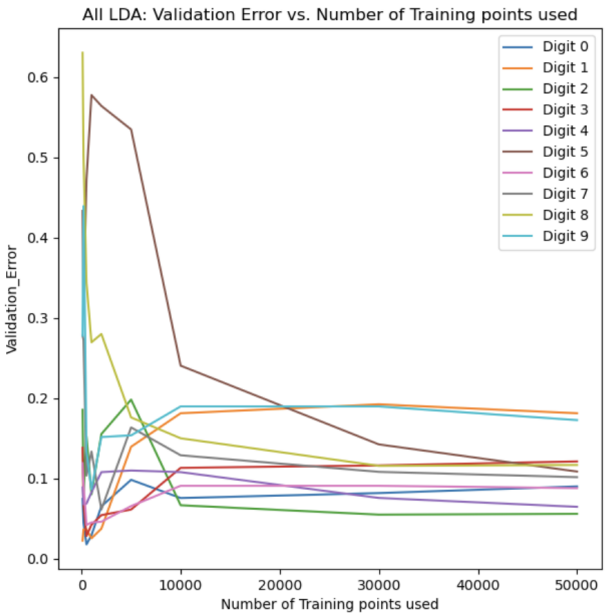
```
LDA:
Training with 100 points
Validation accuracy: 0.6767


Training with 200 points
Validation accuracy: 0.6451


Training with 500 points
Validation accuracy: 0.5688


Training with 1000 points
Validation accuracy: 0.7404


Training with 2000 points
Validation accuracy: 0.8132


Training with 5000 points
Validation accuracy: 0.8541


Training with 10000 points
Validation accuracy: 0.86


Training with 30000 points
Validation accuracy: 0.8704


Training with 50000 points
Validation accuracy: 0.8707
```

QDA:

```
QDA:
Training with 100 points
Validation acc: 0.7803

Training with 200 points
Validation acc: 0.812

Training with 500 points
Validation acc: 0.8657

Training with 1000 points
Validation acc: 0.8694

Training with 2000 points
Validation acc: 0.8545

Training with 5000 points
Validation acc: 0.8345

Training with 10000 points
Validation acc: 0.8668

Training with 30000 points
Validation acc: 0.8833

Training with 50000 points
Validation acc: 0.8898
```

D)

```
    #### QUESTION 8-3: Part d) plotting all 10 curves for all 10 digits for LDA
    figure = plt.figure(3)

    digits_error = {}
    # generate error rates for all predictions
    for prediction in predictions:
        for digit in range(0, 10):
            digit_predictions = prediction[mnist_validation_labels == digit]
            labels = mnist_validation_labels[mnist_validation_labels == digit]
            digits_error[digit] = digits_error.get(digit, []) + [(digit_predictions
== labels).mean()]

    # plot the error for each digit
    for digit in digits_error.keys():
        plt.plot(training_points, [1 - digit for digit in digits_error[digit]])

    plt.legend([f"Digit {digit}" for digit in list(range(0, 10))], loc='upper
right')
    plt.title("All LDA: Validation Error vs. Number of Training points used")
    plt.xlabel("Number of Training points used")
    plt.ylabel("Validation_Error")
    # plt.show()
    plt.savefig('LDA_all_10.png')
```



Which digit is easiest to classify?

Digit 0 and 1 are the easiest digits to classify. I found these to be the easiest to classify because they have the most unique features as other digits. Looking at their LDA Loss graph as well, they have the smallest minimum value on the all digits LDA graph.

Part 4: Kaggle Submission

Kaggle username: **Hiva Mohammadzadeh**

Kaggle Scores:

MNIST: 0.87933 = 88%

```
## QUESTION 8-4: Kaggle Submission on the better performed model on MNIST.

   # Use the calculated priors and means and covariance from training LDA to
generate predictions for test data
   testing_num = mnist_testing_data.shape[0]
   # print(mnist_testing_data.shape)
   test_out = np.zeros((testing_num, 10))

   # Run on test data and make predictions for each digit
   for digit in range(0, 10):
       prior = priors[digit]
       mean = means[digit, :].reshape(-1, 1)

       weight = np.linalg.solve(covariance.T, mean)

       alpha = -0.5 * weight.T.dot(mean) + np.log(prior)

       test_out[:, digit] = (mnist_testing_data.dot(weight) + alpha).reshape(-1,)

   test_predictions = np.argmax(test_out, axis=1).reshape(-1,)
   # Save the result to a csv file
   results_to_csv(test_predictions)
   print("Successfully ran on test data for MNIST and saved predictions to the csv
file\n"
```

```
Successfully ran on test data for MNIST and saved predictions to the csv file
```

Part 5: LDA and QDA on SPAM

```
#### QUESTION 8-5: SPAM Classification of Mails using LDA and QDA

   # Load the SPAM training data
   print("\nSPAM:\n")
   data = np.load(f"../data/spam-data-hw3.npz")
   spam_training_data = data["training_data"]
   spam_training_labels = data["training_labels"]
   #reshape them to match
   spam_training_labels = np.reshape(spam_training_labels,(4172, 1))
   # Check the shapes
   print(f"data size: {spam_training_data.shape}")
   print(f"labels size: {spam_training_labels.shape}")

   # Fitting a Gaussian distribution to each digit using MLE
   samples_for_mean = {}
   samples_for_covariance = {}

   # Computing a mean and a covariance matrix for each digit class
   for mail in list(np.unique(spam_training_labels)):

       data_for_mail = spam_training_data[spam_training_labels.reshape(-1,) ==
mail, :]

       mean = data_for_mail.mean(axis=0)
       covariance = np.cov(data_for_mail.T)

       samples_for_mean[mail] = mean
       samples_for_covariance[mail] = covariance

   # Print and check the shapes
   print(f"\nFitting Gaussian Distribution: \nMean: {samples_for_mean.keys()}
\nLength of mean of spam email: {len(samples_for_mean[1])}")
   print(f"Covariance: {samples_for_covariance.keys()} \nLength of covariance of
non spam email: {len(samples_for_covariance[0])}")


   # Partitioning dataset using bag of words just like HW1

   # Concatenating the labels with data in order to shuffle better
   concatenated_data = np.concatenate((spam_training_data, spam_training_labels),
axis = 1)

   #Shuffling the data
   np.random.shuffle(concatenated_data)
   spam_training_data = concatenated_data[:,:-1]
   spam_training_labels = np.reshape(concatenated_data[:,-1], (4172, 1))
   # print(training_labels.shape)

   #Split using the amount we want in validation set
   amount_set_aside = 500
   spam_validation_data, spam_validation_labels =
spam_training_data[:amount_set_aside,:], spam_training_labels[:amount_set_aside,:]
   spam_training_data, spam_training_labels =
spam_training_data[amount_set_aside:,:], spam_training_labels[amount_set_aside:,:]


   #Print the datasets' shapes
   print(f"\nPartitioned: \nTraining data: {spam_training_data.shape} \nTraining
labels: {spam_training_labels.shape}")
   print(f"Validation data: {spam_validation_data.shape} \nValidation labels:
{spam_validation_labels.shape}")

   spam_testing_data = data["test_data"]
   print(f"Testing data: {spam_testing_data.shape}")
```

```
### Question 8-5: LDA on Spam

   ## Function to train LDA for the SPAM
   def spam_lda(training_data, training_labels):
       a, b = training_data.shape
       priors = np.zeros((2,1))
       covariance = np.zeros((b,b)).astype(np.float32)
       samples_for_mean = np.zeros((2, b))
       training_labels = training_labels.reshape(-1,)

       ## Calculates prior, mean, and covariance for each mail
       for mail in [0,1]:

           priors[int(mail)] = (training_labels == mail).astype(np.int32).sum() / a

           data = training_data[training_labels == mail, :]
           samples_for_mean[int(mail), :] = data.mean(axis = 0)
           covariance += np.cov(data.T)

       covariance /= 2
       return priors, samples_for_mean, covariance

   # Training the LDA Model for MNIST
   priors, means, covariance = spam_lda(spam_training_data, spam_training_labels)

   train_num = spam_training_data.shape[0]
   train_out = np.zeros((train_num, 2))

   validation_num = spam_validation_data.shape[0]
   validation_out = np.zeros((validation_num, 2))

   # Calculate mean, prior and covariance for each digit
   for mail in [0,1]:
       prior = priors[mail]
       mean = means[mail, :].reshape(-1, 1)

       weight= np.linalg.solve(covariance.T, mean)
       alpha = -0.5 * weight.T.dot(mean) + np.log(prior)

       validation_out[:, mail] = (spam_validation_data.dot(weight) +
alpha).reshape(-1,)

   # Calculate validation predictions for all mails
   validation_predictions = np.argmax(validation_out, axis=1).reshape(-1, 1)

   print("\nLDA:")
   print(f"Validation accuracy: {(spam_validation_labels ==
validation_predictions).mean()}\n")

### QUESTION 8-5: Doing QDA on Spam

  ## Function to train QDA for the SPAM
   def spam_qda(training_data, training_labels):
       a, b = training_data.shape
       priors = np.zeros((2,1))
       covariances = np.zeros((2,b,b)).astype(np.float32)
       samples_for_mean = np.zeros((2, b))
       training_labels = training_labels.reshape(-1,)

       ## Computes prior, mean, and covariance for each mail
       for mail in [0,1]:

           priors[int(mail)] = ((training_labels == mail).astype(np.int32).sum() /
a)

           data = training_data[training_labels == mail, :]

           samples_for_mean[int(mail),:] = data.mean(axis=0)
```

```
            covariances[int(mail), :, :] = np.cov(data.T)

        return priors, samples_for_mean, covariances

    # Train QDA on SPAM
    priors, means, covariances = spam_qda(spam_training_data, spam_training_labels)
    validation_num = spam_validation_data.shape[0]
    validation_out = np.zeros((validation_num, 2))

    # calculate the qda validation accuracies for mails
    for mail in [0,1]:
        prior = priors[mail]
        mean = means[mail, :]
        covariance = covariances[mail, :, :]
        # Fixing and preventing the singular covariance matrices. I just add a very
small value (1e-6) to the diagonal values of the covariance to make the eigenvalues
positive.
        covariance = covariance + (1e-6) * np.eye(*covariance.shape)

        alpha = -0.5 * np.linalg.det(covariance) + prior

        mean_centered_validation= spam_validation_data - mean
        validation_weight = np.linalg.solve(covariance, mean_centered_validation.T)

        a, b = mean_centered_validation.shape
        out_diagonal = np.array([mean_centered_validation[n, :].reshape(1,
-1).dot(validation_weight[:, n].reshape(-1, 1)) for n in range(a)])

        validation_prediction = -0.5 * out_diagonal + alpha
        validation_out[:, mail] = validation_prediction.reshape(validation_num,)


    validation_prediction_final = np.argmax(validation_out, axis=1).reshape(-1, 1)
    print("QDA:")
    print(f"Validation accuracy: {(spam_validation_labels ==
validation_prediction_final).mean()}\n")



 ## QUESTION 8-4: Kaggle Submission on the better performed model on SPAM.

    # Use the calculated priors and means and covariance from training LDA to
generate predictions for test data
    testing_num = spam_testing_data.shape[0]
    # print(mnist_testing_data.shape)
    out_test = np.zeros((testing_num, 2))

    # Run on test data and make predictions for each mail
    for mail in [0,1]:
        prior = priors[mail]
        mean = means[mail, :].reshape(-1, 1)

        weight = np.linalg.solve(covariance.T, mean)

        alpha = -0.5 * weight.T.dot(mean) + np.log(prior)

        out_test[:, mail] = (spam_testing_data.dot(weight) + alpha).reshape(-1,)

    test_predictions = np.argmax(out_test, axis=1).reshape(-1,)
    # Save the result to a csv file
    results_to_csv(test_predictions)
    print("Successfully ran on test data for SPAM and saved predictions to the csv
file\n")
```

Kaggle username: **Hiva Mohammadzadeh**

Kaggle Scores:

b) SPAM: 0.784 = 78%

```
SPAM:

data size: (4172, 32)
labels size: (4172, 1)

Fitting Gaussian Distribution:
Mean: dict_keys([0, 1])
Length of mean of spam email: 32
Covariance: dict_keys([0, 1])
Length of covariance of non spam email: 32

Partitioned:
Training data: (3672, 32)
Training labels: (3672, 1)
Validation data: (500, 32)
Validation labels: (500, 1)
Testing data: (1000, 32)

LDA:
Validation accuracy: 0.76

Successfully ran on test data for SPAM and saved predictions to the csv file

QDA:
Validation accuracy: 0.788
```