COMPSCI 289A Homework 3 **Mitch Negus**
2/27/2017 **3032146443**

Problem 1

a.)

X and Y are independent if $P(X = x \cap Y = y) = P(X = x)P(Y = y)$. Given that $P(X = 0 \cap Y = 0) = 0$ since X and Y are never both zero, we find

$$P(X = 0 \cap Y = 0) \neq P(X = 0)P(Y = 0)$$

$$0 \neq 0.5 \cdot 0.5 = 0.25.$$

X and Y are not independent.

X and Y are uncorrelated if $E[XY] = E[X]E[Y]$. By using the definition of the expectation value, we can state

$$E[XY] = \int xyf(x,y)\; dx\, dy$$

Since either $X = 0$ or $Y = 0$,

$$E[XY] = \int (0)f(x,y)\; dx\, dy = 0$$

Similarly

$$E[X]E[Y] = \int xf(x,y)\; dx\, dy \int yf(x,y)\; dx\, dy$$

and, when solved for the discrete values for X and Y, is

$$E[X]E[Y] = ((1)(0.25) + (-1)(0.25))\,((1)(0.25) + (-1)(0.25))$$

$$E[X]E[Y] = 0$$

$$\boxed{E[XY] = E[X]E[Y] = 0}.$$

b.)

We are given that

$P(B = 0) = \frac{1}{2}$
$P(B = 1) = \frac{1}{2}$

$P(C = 0) = \frac{1}{2}$
$P(C = 1) = \frac{1}{2}$

$P(D = 0) = \frac{1}{2}$
$P(D = 1) = \frac{1}{2}$

and that

$P(X) = P(B \oplus C)$
$P(X) = P(B)P(\bar{C}) + P(\bar{B})P(C)$

$P(Y) = P(C \oplus D)$
$P(Y) = P(C)P(\bar{D}) + P(\bar{C})P(D)$

$P(Z) = P(B \oplus D)$
$P(Z) = P(B)P(\bar{D}) + P(\bar{B})P(D)$

$P(X), P(Y),$ and $P(Z)$ are mutually independent if $P(X \cap Y \cap Z) = P(X)P(Y)P(Z)$.

$$P(X \cap Y \cap Z) = P(X)P(Y)P(Z)$$

X	Y	Z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

By constructing a table (above) showing the 8 unique outcomes, all of equal probability, we can see that each outcome has a probability of 1/8; $P(X \cap Y \cap Z) = 1/8$.

Additionally, since

$$P(X) = P(B)P(\bar{C}) + P(\bar{B})P(C)$$

and if we let the positive case $P(B) = P(B = 0) \therefore P(\bar(B)) = P(B = 1)$ (and we use a similar convention for $P(C)$ and $P(D)$) then

$$P(X) = P(B = 0)P(C = 1) + P(B = 1)P(C = 0)$$

$$P(X) = \left(\frac{1}{2}\right)\left(\frac{1}{2}\right) + \left(\frac{1}{2}\right)\left(\frac{1}{2}\right)$$

$$P(X) = \left(\frac{1}{4}\right) + \left(\frac{1}{4}\right)$$

$$P(X) = \left(\frac{1}{2}\right).$$

Following the same procedure for $P(Y)$ and $P(Z)$, we find $P(X) = 1/2, P(Y) = 1/2,$ and $P(Z) = 1/2$. Then, we can write

$$P(X)P(Y)P(Z) = \left(\frac{1}{2}\right)^3$$

$$P(X)P(Y)P(Z) = \frac{1}{8}$$

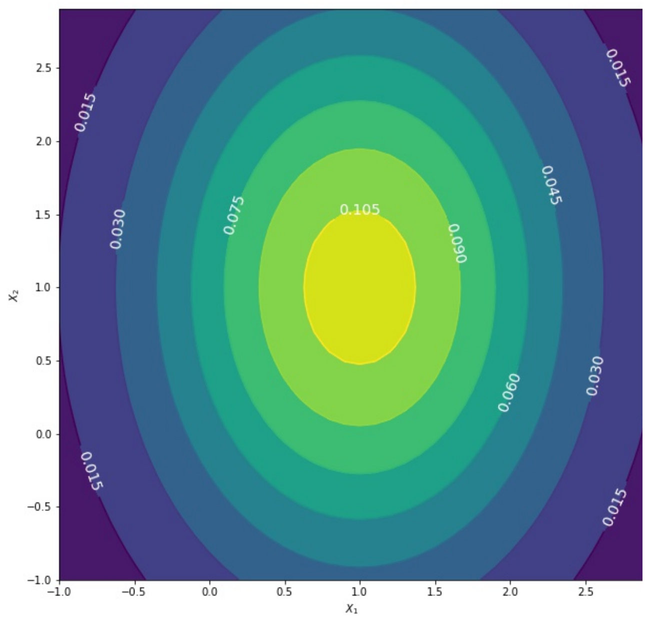
and so indeed

$$P(X \cap Y \cap Z) = P(X)P(Y)P(Z) = \frac{1}{8}.$$

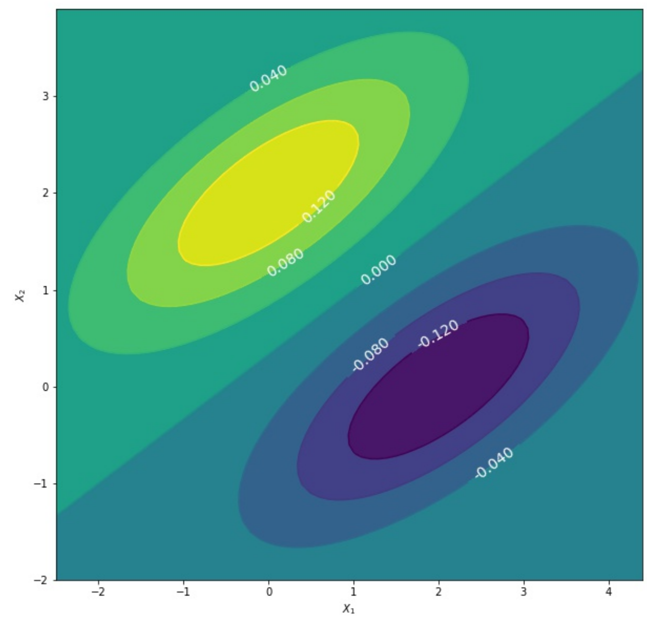
Therefore, we know that X, Y, and Z are mutually independent (and therefore also pairwise independent).

Problem 2

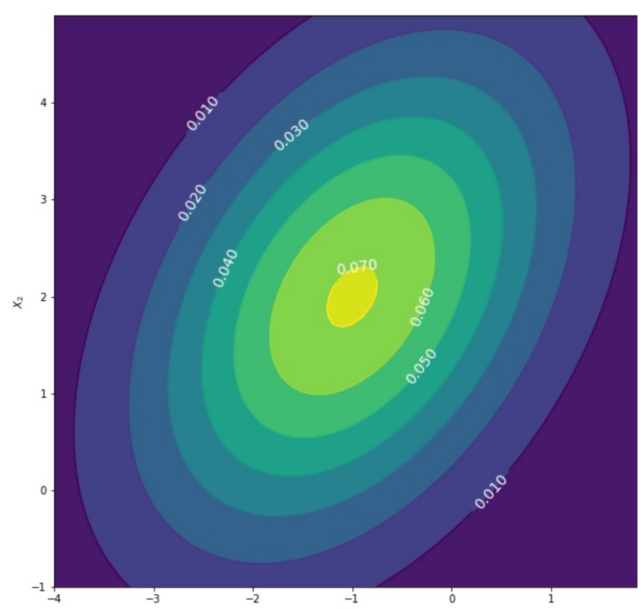
***a.*)**



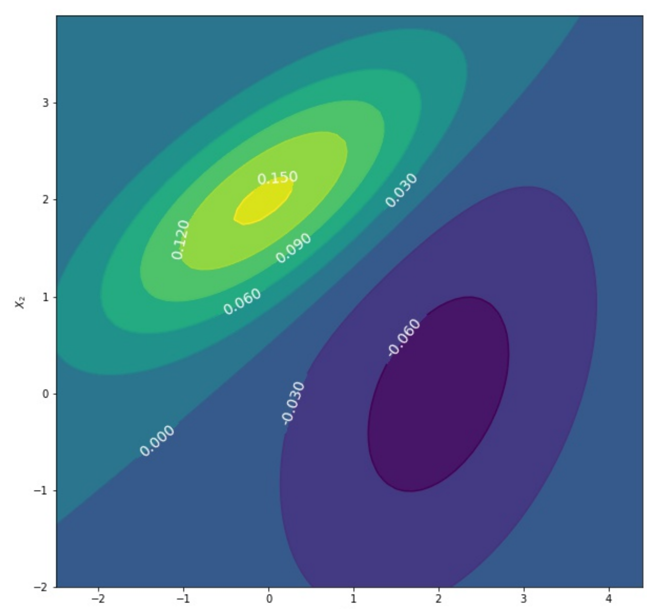
***c.*)**



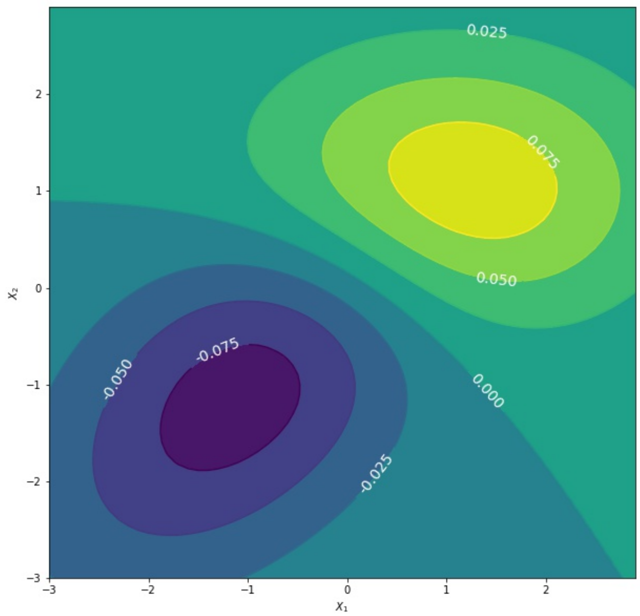
***b.*)**



***d.*)**



***e.*)**



Problem 3

a.)

Mean: $\begin{pmatrix} 3.18 \\ 5.09 \end{pmatrix}$

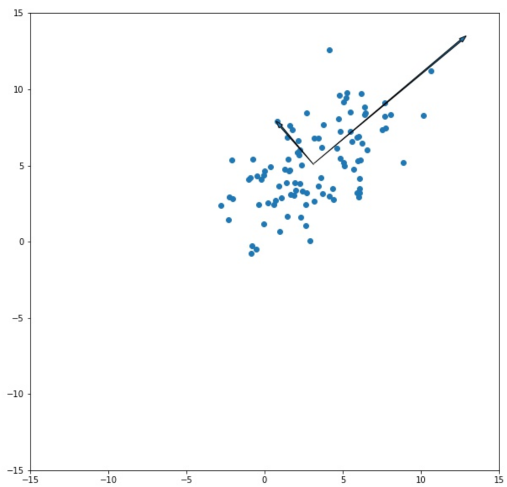
b.)

2 × 2 Covariance Matrix:
$\begin{pmatrix} 8.49 & 4.52 \\ 4.52 & 7.19 \end{pmatrix}$

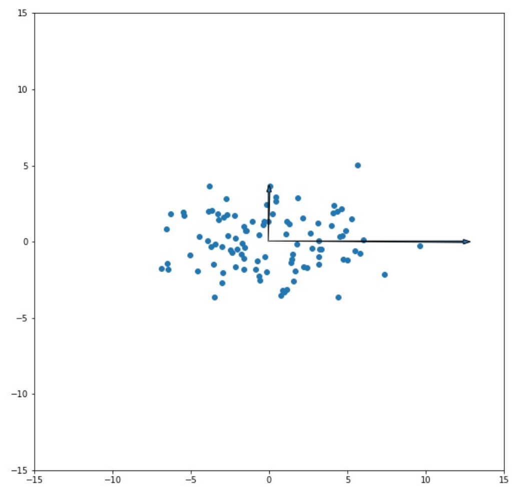
c.)

Eigenvalues: 12.41, 3.27
Eigenvectors: $\begin{pmatrix} 0.76 \\ 0.65 \end{pmatrix}$, $\begin{pmatrix} -0.65 \\ 0.76 \end{pmatrix}$

d.)



e.)



Problem 4

Let $X_1, ...X_n \in \mathbb{R}^d$ be draw independently from multivariate normal distribution $\mathcal{N}(\mu, \Sigma)$.

a.)

We can express the likelihood function of choosing $X_1, ...X_n$ as

$$\mathcal{L} = \prod_{i=1}^{n} P(X_i),$$

and where $P(X_i)$ is given by the normal distribution such that

$$P(X_i) = \frac{1}{\sqrt{(2\pi)^d|\Sigma|}} e^{-\frac{1}{2}(X_i-\mu)^T\Sigma^{-1}(X_i-\mu)}.$$

Then, taking the natural logarithm of the likelihood function we can express the log-likelihood function as

$$\ell = \ln\left(\prod_{i=1}^{n} P(X_i)\right) = \sum_{i=1}^{n} \ln P(X_i)$$

$$\ell = \sum_{i=1}^{n} \ln\left(\frac{1}{\sqrt{(2\pi)^d|\Sigma|}} e^{-\frac{1}{2}(X_i-\mu)^T\Sigma^{-1}(X_i-\mu)}\right)$$

$$\ell = \sum_{i=1}^{n} \ln\left(\frac{1}{\sqrt{(2\pi)^d|\Sigma|}}\right) + \sum_{i=1}^{n} \ln\left(e^{-\frac{1}{2}(X_i-\mu)^T\Sigma^{-1}(X_i-\mu)}\right)$$

$$\ell = \sum_{i=1}^{n} -\frac{1}{2}(X_i-\mu)^T\Sigma^{-1}(X_i-\mu) - \sum_{i=1}^{n} \frac{1}{2}\ln\left((2\pi)^d|\Sigma|\right)$$

$$\ell = -\frac{1}{2}\left(\sum_{i=1}^{n}(X_i-\mu)^T\Sigma^{-1}(X_i-\mu) + \sum_{i=1}^{n}\ln(2\pi)^d + \sum_{i=1}^{n}\ln|\Sigma|\right)$$

$$\ell = -\frac{1}{2}\left(\sum_{i=1}^{n}(X_i-\mu)^T\Sigma^{-1}(X_i-\mu) + nd\ln(2\pi) + n\ln|\Sigma|\right)$$

To calculate the maximum likelihood estimate for μ (actually maximizing $\hat{\mu}$, as we may only estimate the mean of the distribution), we take the gradient with respect to μ and set it equal to zero.

$$\nabla_\mu \ell = \nabla_\mu\left(-\frac{1}{2}\left(\sum_{i=1}^{n}(X_i-\mu)^T\Sigma^{-1}(X_i-\mu) + nd\ln(2\pi) + n\ln|\Sigma|\right)\right)$$

$$\nabla_\mu \ell = -\frac{1}{2}\nabla_\mu\left(\sum_{i=1}^{n}(X_i-\mu)^T\Sigma^{-1}(X_i-\mu)\right)$$

$$\nabla_\mu \ell = -\frac{1}{2}\sum_{i=1}^{n}\nabla_\mu\left((X_i-\mu)^T\Sigma^{-1}(X_i-\mu)\right)$$

In the previous homework (problem 2b.) we showed that if A is square and symmetric, then $\nabla_x(x^TAx) = 2Ax$. Applying the chain rule, if x is a function of y, then we can state that $\nabla_y(x^TAx) = \nabla_x(x^TAx)\nabla_y x = 2Ax\,\nabla_y x$. If we let $x = (X_i - \mu)$ and $\Sigma^{-1} = A$, then we find

$$\nabla_\mu \ell = -\frac{1}{2}\sum_{i=1}^{n}\nabla_\mu(x^TAx) = -\frac{1}{2}\sum_{i=1}^{n}\nabla_x(x^TAx)\,\nabla_\mu x$$

$$\nabla_\mu \ell = -\frac{1}{2}\sum_{i=1}^{n} 2Ax \, \nabla_\mu x$$

$$\nabla_\mu \ell = -\sum_{i=1}^{n} \Sigma^{-1}(X_i - \mu) \, \nabla_\mu (X_i - \mu)$$

$$\nabla_\mu \ell = \sum_{i=1}^{n} \Sigma^{-1}(X_i - \mu)$$

$$\nabla_\mu \ell = \sum_{i=1}^{n} \Sigma^{-1} X_i - \sum_{i=1}^{n} \Sigma^{-1}\mu$$

$$\nabla_\mu \ell = \Sigma^{-1}\sum_{i=1}^{n} X_i - n\Sigma^{-1}\mu$$

Setting $\nabla_\mu \ell = 0$,

$$0 = \Sigma^{-1}\sum_{i=1}^{n} X_i - n\Sigma^{-1}\mu$$

$$n\Sigma^{-1}\mu = \Sigma^{-1}\sum_{i=1}^{n} X_i$$

$$\boxed{\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} X_i}$$

To calculate the maximum likelihood estimate for $\hat{\sigma}_j$ we take the partial derivative with respect to σ_j and set it equal to zero (note that here I have redefined $\hat{\sigma}_i$—as given in the problem—as $\hat{\sigma}_j$ to avoid confusing counting indices for n and d).

$$\frac{\partial \ell}{\partial \sigma_j} = \frac{\partial}{\partial \sigma_j}\left(-\frac{1}{2}\left(\sum_{i=1}^{n}(X_i - \mu)^T\Sigma^{-1}(X_i - \mu) + nd\ln(2\pi) + n\ln|\Sigma|\right)\right)$$

$$\frac{\partial \ell}{\partial \sigma_j} = -\frac{1}{2}\frac{\partial}{\partial \sigma_j}\left(\sum_{i=1}^{n}(X_i - \mu)^T\Sigma^{-1}(X_i - \mu) + n\ln|\Sigma|\right)$$

We are given that the distribution has unknown diagonal covariance matrix, where the j^{th} element of the diagonal, $\Sigma_{jj} = \sigma_j^2$. From this, we can directly conclude that the inverse of the covariance matrix, Σ^{-1}, is also a diagonal matrix with diagonal elements $(\Sigma^{-1})_{jj} = 1/\sigma_j^2$. With this form, we can simplify the matrix product in the summation term above to yield

$$\frac{\partial \ell}{\partial \sigma_j} = -\frac{1}{2}\frac{\partial}{\partial \sigma_j}\left(\sum_{i=1}^{n}\sum_{j=1}^{d}\frac{|X_{ij} - \mu|^2}{\sigma_j^2} + n\ln|\Sigma|\right)$$

Furthermore, for diagonal matrices, we can express the determinant as the product of the diagonal elements.

$$\frac{\partial \ell}{\partial \sigma_j} = -\frac{1}{2}\frac{\partial}{\partial \sigma_j}\left(\sum_{i=1}^{n}\sum_{j=1}^{d}\frac{|X_{ij} - \mu_j|^2}{\sigma_j^2} + n\ln\left(\prod_{j=1}^{d}\sigma_j^2\right)\right)$$

$$\frac{\partial \ell}{\partial \sigma_j} = -\frac{1}{2}\frac{\partial}{\partial \sigma_j}\left(\sum_{i=1}^{n}\sum_{j=1}^{d}\frac{|X_{ij} - \mu_j|^2}{\sigma_j^2} + n\sum_{j=1}^{d}\ln\sigma_j^2\right)$$

$$\frac{\partial \ell}{\partial \sigma_j} = -\frac{1}{2}\left[\frac{\partial}{\partial \sigma_j}\left(\sum_{i=1}^{n}\sum_{j=1}^{d}\frac{|X_{ij} - \mu_j|^2}{\sigma_j^2}\right) + \frac{\partial}{\partial \sigma_j}\left(n\sum_{j=1}^{d}\ln\sigma_j^2\right)\right]$$

$$\frac{\partial \ell}{\partial \sigma_j} = -\frac{1}{2}\left[\sum_{i=1}^{n} \frac{\partial}{\partial \sigma_j}\left(\sum_{j=1}^{d} \frac{|X_{ij}-\mu_j|^2}{\sigma_j^2}\right) + n\frac{\partial}{\partial \sigma_j}\left(\sum_{j=1}^{d} \ln \sigma_j^2\right)\right]$$

$$\frac{\partial \ell}{\partial \sigma_j} = -\frac{1}{2}\left[\sum_{i=1}^{n} \frac{\partial}{\partial \sigma_j}\left(\frac{|X_{ij}-\mu_j|^2}{\sigma_j^2}\right) + \frac{\partial}{\partial \sigma_j}\left(\ln \sigma_j^2\right)\right]$$

$$\frac{\partial \ell}{\partial \sigma_j} = -\frac{1}{2}\left[\sum_{i=1}^{n} |X_{ij}-\mu_j|^2 \frac{\partial}{\partial \sigma_j}\left(\frac{1}{\sigma_j^2}\right) + 2n\frac{\partial}{\partial \sigma_j}\ln(\sigma_j)\right]$$

$$\frac{\partial \ell}{\partial \sigma_j} = -\frac{1}{2}\left[\sum_{i=1}^{n} |X_{ij}-\mu_j|^2 \left(\frac{-2}{\sigma_j^3}\right) + 2n\left(\frac{1}{\sigma_j}\right)\right]$$

$$\frac{\partial \ell}{\partial \sigma_j} = \sum_{i=1}^{n}\left(\frac{|X_{ij}-\mu_j|^2}{\sigma_j^3}\right) - \left(\frac{n}{\sigma_j}\right)$$

Setting $\frac{\partial \ell}{\partial \sigma_j} = 0$,

$$0 = \sum_{i=1}^{n}\left(\frac{|X_{ij}-\mu_j|^2}{\sigma_j^3}\right) - \left(\frac{n}{\sigma_j}\right)$$

$$0 = \sum_{i=1}^{n}\left(\frac{|X_{ij}-\mu_j|^2}{\sigma_j^3}\right) - \left(\frac{n\sigma_j^2}{\sigma_j^3}\right)$$

$$0 = \sum_{i=1}^{n} |X_{ij}-\mu_j|^2 - n\sigma_j^2$$

$$n\sigma_j^2 = \sum_{i=1}^{n} |X_{ij}-\mu_j|^2$$

$$\hat{\sigma}_j = \sqrt{\frac{1}{n}\sum_{i=1}^{n} |X_{ij}-\mu_j|^2}$$

If we let $\mu_j = \hat{\mu}_j = \frac{1}{n}\sum_{i=1}^{n} X_{ij}$, then

$$\boxed{\hat{\sigma}_j = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left|X_{ij} - \frac{1}{n}\sum_{i=1}^{n} X_{ij}\right|^2}}$$

b.)

Now the normal distribution has a known covariance matrix Σ, and an unknown mean $A\mu$. Σ and A are known $d \times d$ matrices, and A is invertible. The multivariate normal distribution is given by

$$\mathcal{N}(A\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d|\Sigma|}} e^{-\frac{1}{2}(X_i - A\mu)^T\Sigma^{-1}(X_i - A\mu)},$$

the likelihood function is again given by

$$\mathcal{L} = \prod_{i=1}^{n} P(X_i),$$

and the log-likelihood function is given by

$$\ell = \ln\left(\prod_{i=1}^{n} P(X_i)\right) = \sum_{i=1}^{n} \ln P(X_i).$$

With the normal distribution as the probability density function for a given X_i, we have

$$\ell = \sum_{i=1}^{n} \ln\left(\frac{1}{\sqrt{(2\pi)^d|\Sigma|}} e^{-\frac{1}{2}(X_i - A\mu)^T\Sigma^{-1}(X_i - A\mu)}\right),$$

and following similar simplification steps to those used in part (a), we find

$$\ell = -\frac{1}{2}\left(\sum_{i=1}^{n}(X_i - A\mu)^T\Sigma^{-1}(X_i - A\mu) + nd\ln(2\pi) + n\ln|\Sigma|\right).$$

Again, we find the maximum likelihood estimate $\hat{\mu}$ for μ by maximizing the function with respect to μ, namely where $\nabla_\mu \ell = 0$.

$$\nabla_\mu \ell = \nabla_\mu\left(-\frac{1}{2}\left(\sum_{i=1}^{n}(X_i - A\mu)^T\Sigma^{-1}(X_i - A\mu) + nd\ln(2\pi) + n\ln|\Sigma|\right)\right)$$

$$\nabla_\mu \ell = -\frac{1}{2}\sum_{i=1}^{n}\nabla_\mu\left((X_i - A\mu)^T\Sigma^{-1}(X_i - A\mu)\right)$$

$$\nabla_\mu \ell = -\frac{1}{2}\sum_{i=1}^{n}\nabla_\mu\left((X_i^T - (A\mu)^T)(\Sigma^{-1}X_i - \Sigma^{-1}A\mu)\right)$$

$$\nabla_\mu \ell = -\frac{1}{2}\sum_{i=1}^{n}\nabla_\mu\left[(X_i^T\Sigma^{-1}X_i - (A\mu)^T\Sigma^{-1}X_i - X_i^T\Sigma^{-1}A\mu + (A\mu)^T\Sigma^{-1}A\mu)\right]$$

$$\nabla_\mu \ell = -\frac{1}{2}\sum_{i=1}^{n}\left[-\nabla_\mu(\mu^TA^T\Sigma^{-1}X_i) - \nabla_\mu(X_i^T\Sigma^{-1}A\mu) + \nabla_\mu(\mu^TA^T\Sigma^{-1}A\mu)\right]$$

Now, let $b = A^T\Sigma^{-1}X_i$ and $B = A^T\Sigma^{-1}A$, then

$$\nabla_\mu \ell = -\frac{1}{2}\sum_{i=1}^{n}\left[-\nabla_\mu(\mu^Tb) - \nabla_\mu(b^T\mu) + \nabla_\mu(\mu^TB\mu)\right]$$

From the previous homework (problem 2a.) we showed that $\nabla_\mu(b^T\mu) = b$. Using a similar procedure, it can also be shown that $\nabla_\mu(\mu^Tb) = b$. Also in the previous homework (problem 2b.), we showed that $\nabla_\mu\mu^TB\mu = (B + B^T)x$. Using these equivalences, we find

$$\nabla_\mu \ell = -\frac{1}{2}\sum_{i=1}^{n}\left[-b - b + (B + B^T)\mu\right]$$

$$\nabla_\mu \ell = \sum_{i=1}^{n}\left[b - \frac{(B + B^T)}{2}\mu\right]$$

Setting $\nabla_\mu \ell = 0$,

$$0 = \sum_{i=1}^{n}\left[b - \frac{(B + B^T)}{2}\mu\right]$$

$$0 = \sum_{i=1}^{n} b - \sum_{i=1}^{n}\frac{(B + B^T)}{2}\mu$$

$$\sum_{i=1}^{n}\frac{(B + B^T)}{2}\mu = \sum_{i=1}^{n} b$$

$$\mu n\frac{(B + B^T)}{2} = \sum_{i=1}^{n} b$$

$$\mu = \frac{2\sum_{i=1}^{n} b}{n(B + B^T)}$$

$$\mu = \frac{2\sum_{i=1}^{n} A^T\Sigma^{-1}X_i}{n(A^T\Sigma^{-1}A + (A^T\Sigma^{-1}A)^T)}$$

Since covariance matrices are by definition symmetric, their inverses are also symmetric, and

$$\mu = \frac{2\sum_{i=1}^{n} A^T\Sigma^{-1}X_i}{n(A^T\Sigma^{-1}A + A^T\Sigma^{-1}A)} = \frac{2A^T\Sigma^{-1}\sum_{i=1}^{n} X_i}{2nA^T\Sigma^{-1}A}$$

$$\mu = \frac{\sum_{i=1}^{n} X_i}{nA}$$

$$\boxed{\hat{\mu} = \frac{1}{2}A^{-1}\sum_{i=1}^{n} X_i}$$

Problem 5

a.)

Any matrix is not invertible if and only if it's determinant is zero. Therefore, $\hat{\Sigma}$ is not invertible if and only if $|\hat{\Sigma}| = 0$. From this, we can use the property that the determinant of a square matrix is equal to the product of the eigenvalues of that matrix to deduce that if $|\hat{\Sigma}| = 0$, then at least one of the eigenvalues of $\hat{\Sigma}$ must be zero.

Geometrically, a matrix with n zero eigenvalues represents a transformation from a d-dimensional space to a $(d-n)$-dimensional space, with the dimensions corresponding to the zero eigenvalues vanishing.

This situation, in which a d-dimensional space collapses to a $(d-n)$-dimensional space could be visualized by a data set, in our case random values of X_i pulled from the multivariate normal distribution, in which the points have no variance in n-dimensions. Geometrically, these points would fall on the same hyperplane in $(d-n+1)$-dimensional space.

b.)

Given that at least one eigenvalue of a singular covariance matrix must be zero, we can deduce that there is no variance in that parameter. Without variance, our machine-learning algorithm will have no ability to use that parameter as a discriminator. A workaround could be that we eliminate all variables with zero variance from influencing our covariance matrix. In the equation for determining the covariance matrix estimator, this is achieved by removing the j^{th} row and column $\{j : 1, ..., d\}$ if $(X_{ij} - \mu_j) = 0 \;\; \forall i \in n$.

c.)

Maximizing $f(x)$ for vectors $|x| = 1$ requires maximizing the argument of the exponential. We are told that $\mu = 0$, so we are looking for the maximum of

$$g(x) = x^T \Sigma^{-1} x, \;\; |x| = 1$$

In the previous homework (problem 4a.) we proved that $\lambda_{\max}(A) = \max_{|x|=1} x^T A x$. From this, we can state

$$\max_{|x|=1} g(x) = \max_{|x|=1} x^T \Sigma^{-1} x = \lambda_{\max}(\Sigma^{-1})$$

We can show that the vector x which satisfies this equation is <u>the eigenvector corresponding to the maximum eigenvalue of Σ^{-1}</u>:

$$x x^T \Sigma^{-1} x = x \lambda_{\max}(\Sigma^{-1}), \; x x^T = 1$$
$$\Sigma^{-1} x = \lambda_{\max}(\Sigma^{-1}) x$$

Using the same procedure to calculate the minimum, and again using our results from the previous homework (problem 4b.) we can state

$$\min_{|x|=1} g(x) = \min_{|x|=1} x^T \Sigma^{-1} x = \lambda_{\min}(\Sigma^{-1})$$

and

$$x x^T \Sigma^{-1} x = x \lambda_{\min}(\Sigma^{-1}), \; x x^T = 1$$
$$\Sigma^{-1} x = \lambda_{\min}(\Sigma^{-1}) x$$

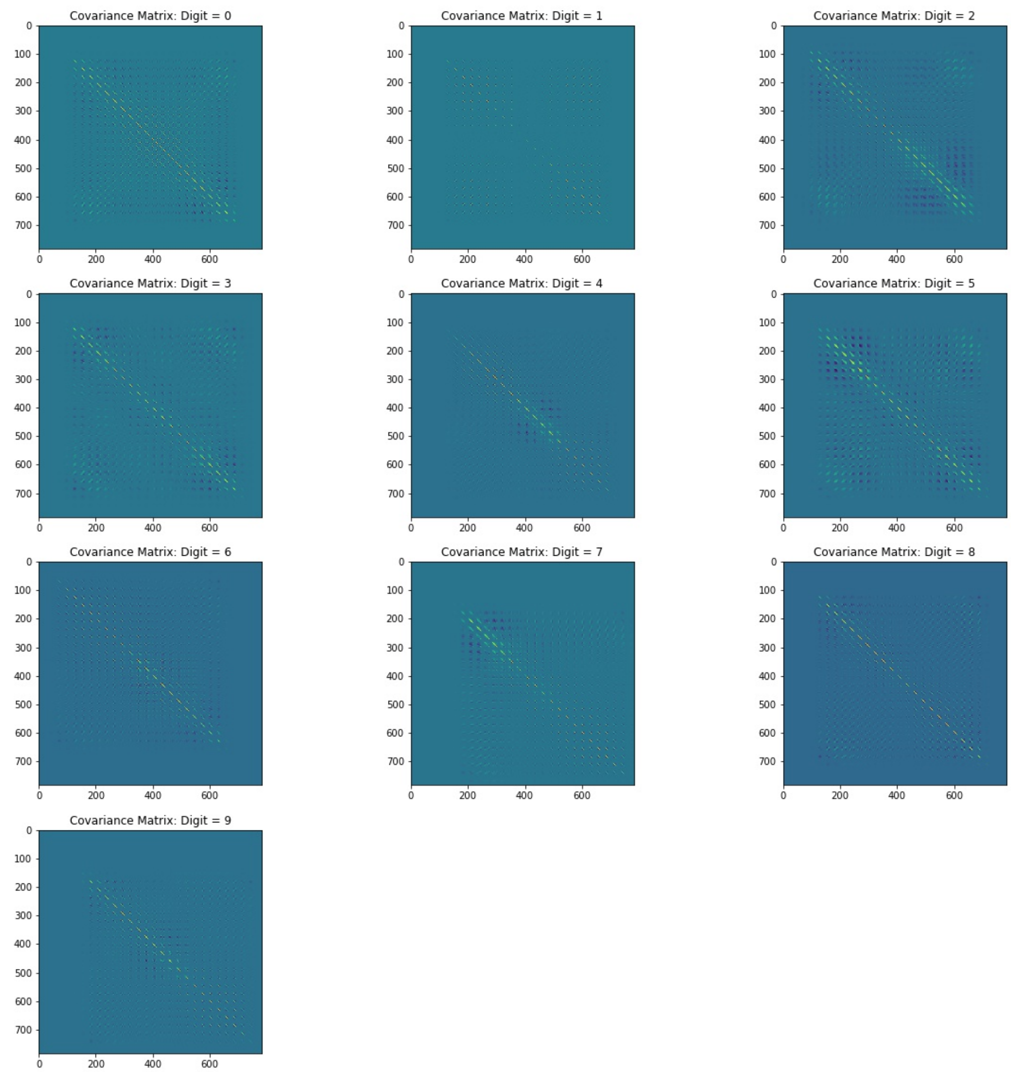
The vector x with $|x| = 1$ which gives the minimum of $f(x)$ is <u>the eigenvector correspondingto the minimum eigenvalue of Σ^{-1}</u>.

Problem 6

a.)

Mean and covariance matrices were calculated, see Appendix for code, and (b) for visualization.

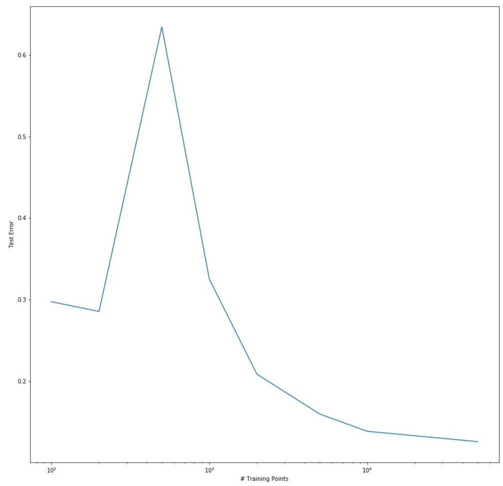
b.)



c.)

(i) LDA

Test errors plotted against number of sample points:
(note that there seems to be large variation among training sets of relatively few samples)



# Sample Points	Error Rate
100	29.74%
200	28.55%
500	63.45%
1,000	32.50%
2,000	20.86%
5,000	15.97%
10,000	13.85%
30,000	12.99%
50,000	12.58%

(ii) QDA

(code included; no plot as I was unsuccessful in resolving singular covariance matrix issue before due date)

CS289A_HW03_Prob2

February 27, 2017

```
In [18]: import math
         import numpy as np
         from matplotlib import pyplot as plt
         from mpl_toolkits.mplot3d import Axes3D

In [19]: def makesquarespace(xmin,xmax,ymin,ymax,delta):
         # Create arrays defining the space over which to evaluate density f'n Z

             xaxis = np.arange(xmin,xmax,delta)
             yaxis = np.arange(ymin,ymax,delta)
             X,Y = np.meshgrid(xaxis,yaxis)
             Z = np.empty_like(X)

             return(X,Y,Z)

In [20]: def calc_density_fn(X,Y,Z,mu,Sigma):
         # Given a space defined by X and Y, evaluate the density function Z for a
         # bivariate normal distribution with mean mu and covariance matrix Sigma
             n = len(mu)
             detCov = np.linalg.det(Sigma)
             SigInv = np.linalg.inv(Sigma)

             for i in range(len(X)):
                 for j in range(len(X[i])):
                     x = np.array([X[i,j],Y[i,j]])
                     Z[i,j] = 1/((2*math.pi)**(n/2)*detCov**(1/2))*math.exp(-0.5*np

             return Z

In [21]: def contourplots(X,Y,Z,filename=None):
         # Create contour plots with labeled isovalues

             fig = plt.figure(figsize=(10,10))
             cs = plt.contour(X,Y,Z)
             plt.contourf(X,Y,Z)
             cs = plt.clabel(cs, inline=1, fontsize=14, colors='white')
             plt.xlabel("$X_1$")
             plt.ylabel("$X_2$")
```

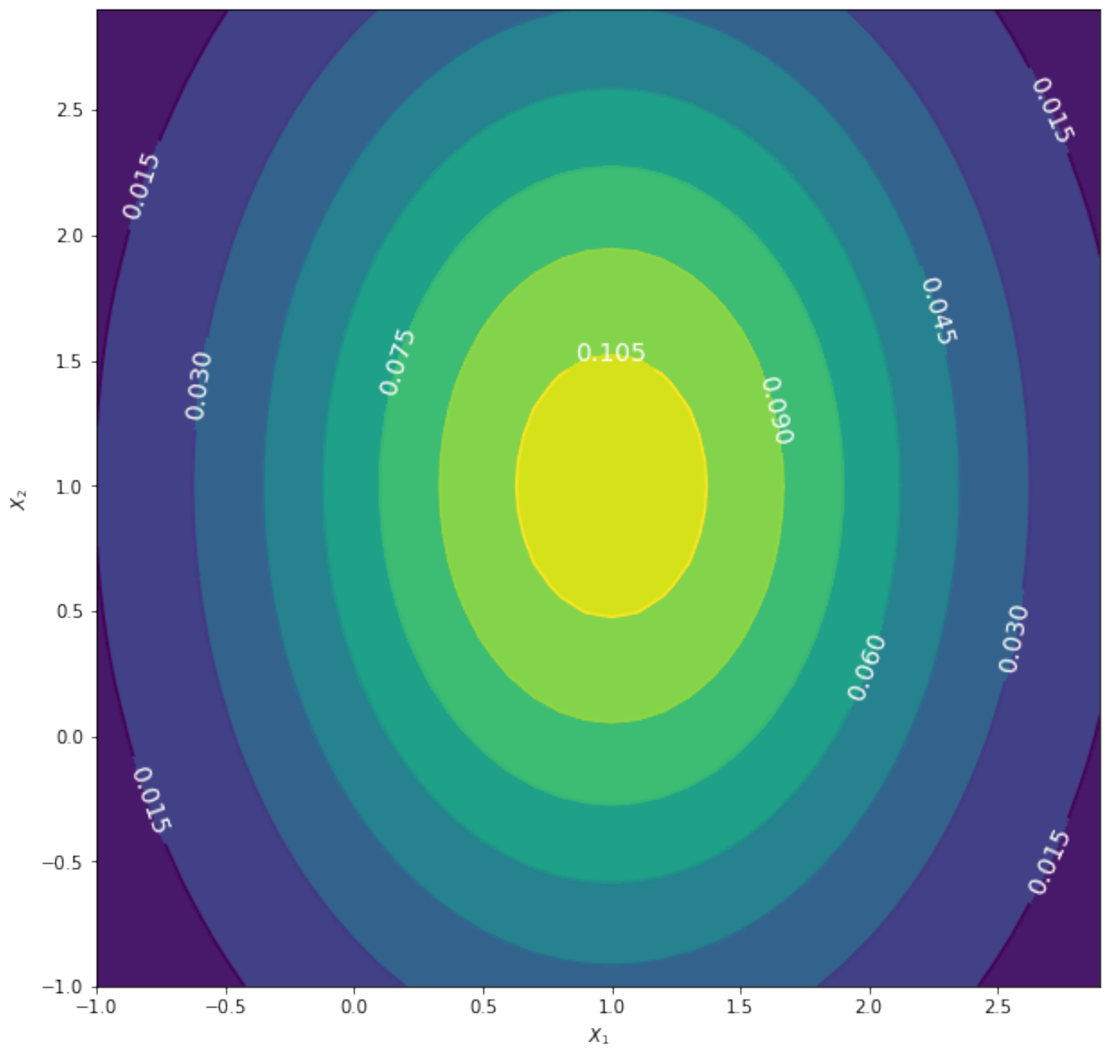
```
        if filename:
            plt.savefig(filename)
        plt.show()
```

```
In [22]: def SolveAndPlot(xmin,xmax,ymin,ymax,delta,mu,Sigma,figname=None):
             X1,X2,Z = makesquarespace(xmin,xmax,ymin,ymax,delta)
             Z = calc_density_fn(X1,X2,Z,mu,Sigma)
             contourplots(X1,X2,Z,figname)
```

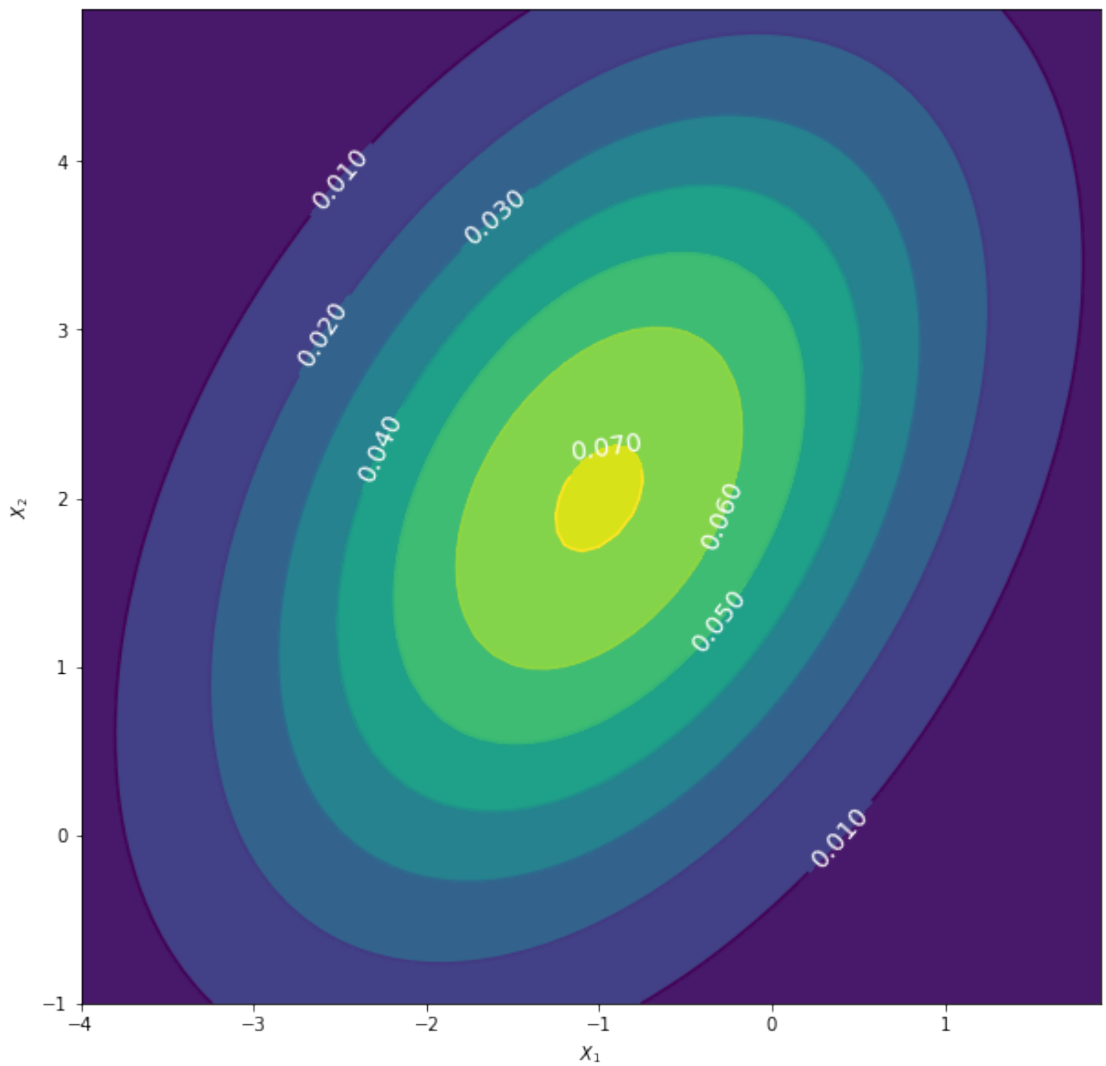
```
In [23]: def SolveSubAndPlot(xmin,xmax,ymin,ymax,delta,mu1,Sigma1,mu2,Sigma2,fignam
             X1,X2,Z = makesquarespace(xmin,xmax,ymin,ymax,delta)
             Y1,Y2 = np.empty_like(Z),np.empty_like(Z)
             Y1 = calc_density_fn(X1,X2,Y1,mu1,Sigma1)
             Y2 = calc_density_fn(X1,X2,Y2,mu2,Sigma2)
             Z = Y1-Y2
             contourplots(X1,X2,Z,figname)
```

```
In [24]: delta = 0.1
```

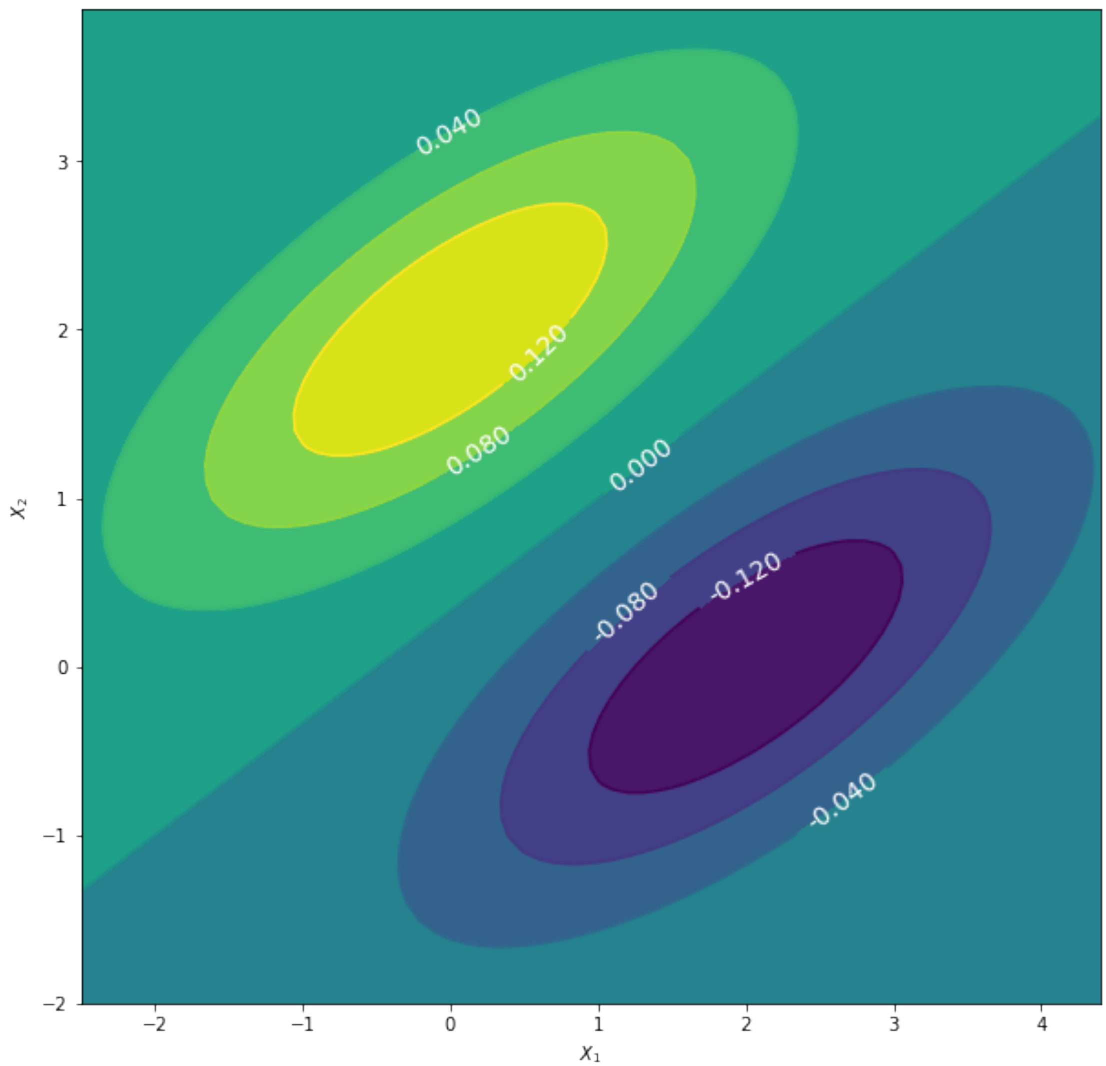
```
In [25]: # (2a)

         # Give the mean and covariance matrix (Sigma) of the multivariate distribu
         mu = np.array([1,1])
         Sigma = np.array([[1,0],[0,2]])

         figname = 'HW02_prob2a.jpg'
         SolveAndPlot(-1,3,-1,3,delta,mu,Sigma,figname)
```



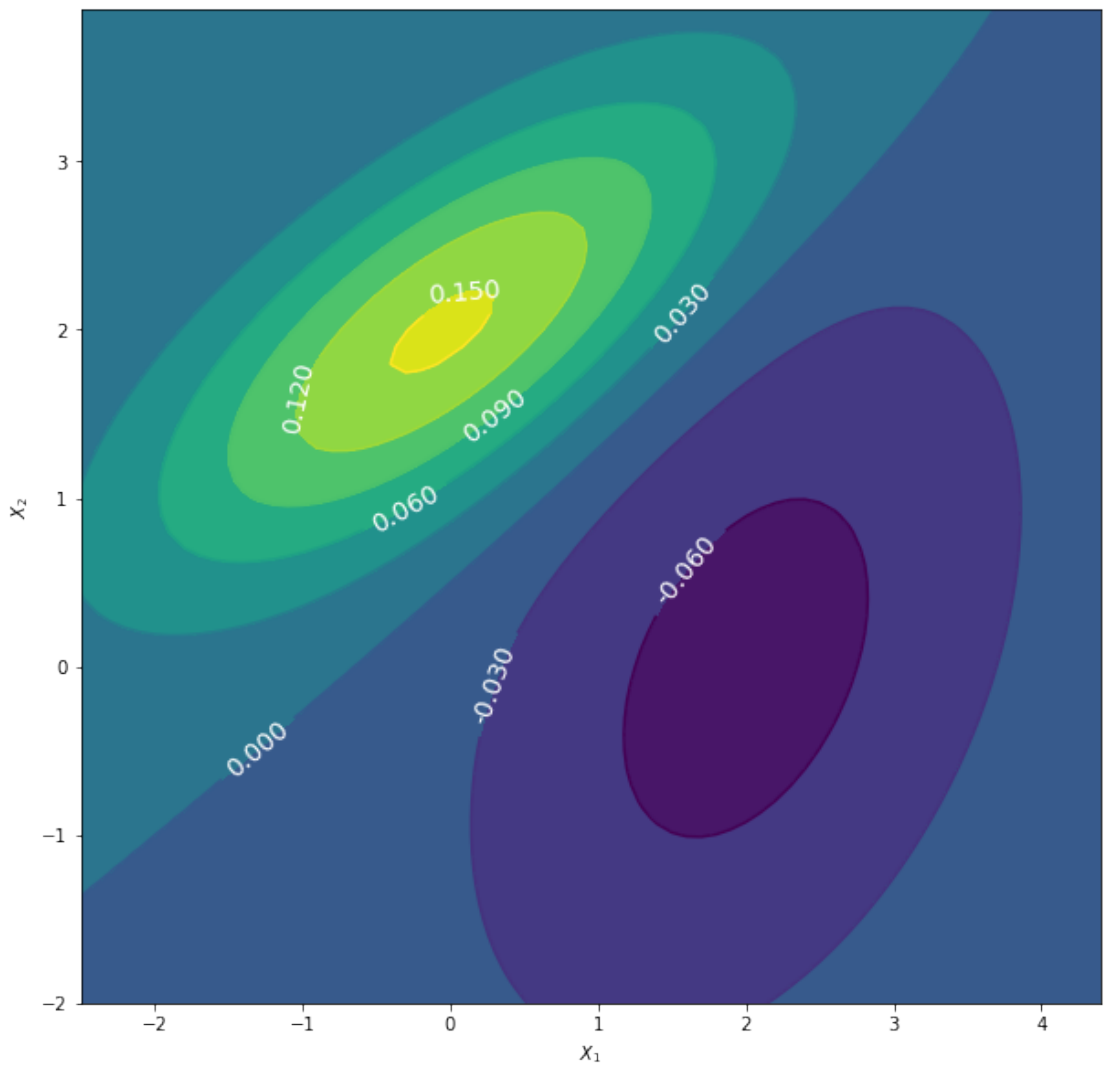

```
In [26]: # (2b)

         # Give the mean and covariance matrix (Sigma) of the multivariate distribu
         mu = np.array([-1,2])
         Sigma = np.array([[2,1],[1,3]])

         figname = 'HW02_prob2b.jpg'
         SolveAndPlot(-4,2,-1,5,delta,mu,Sigma,figname)
```



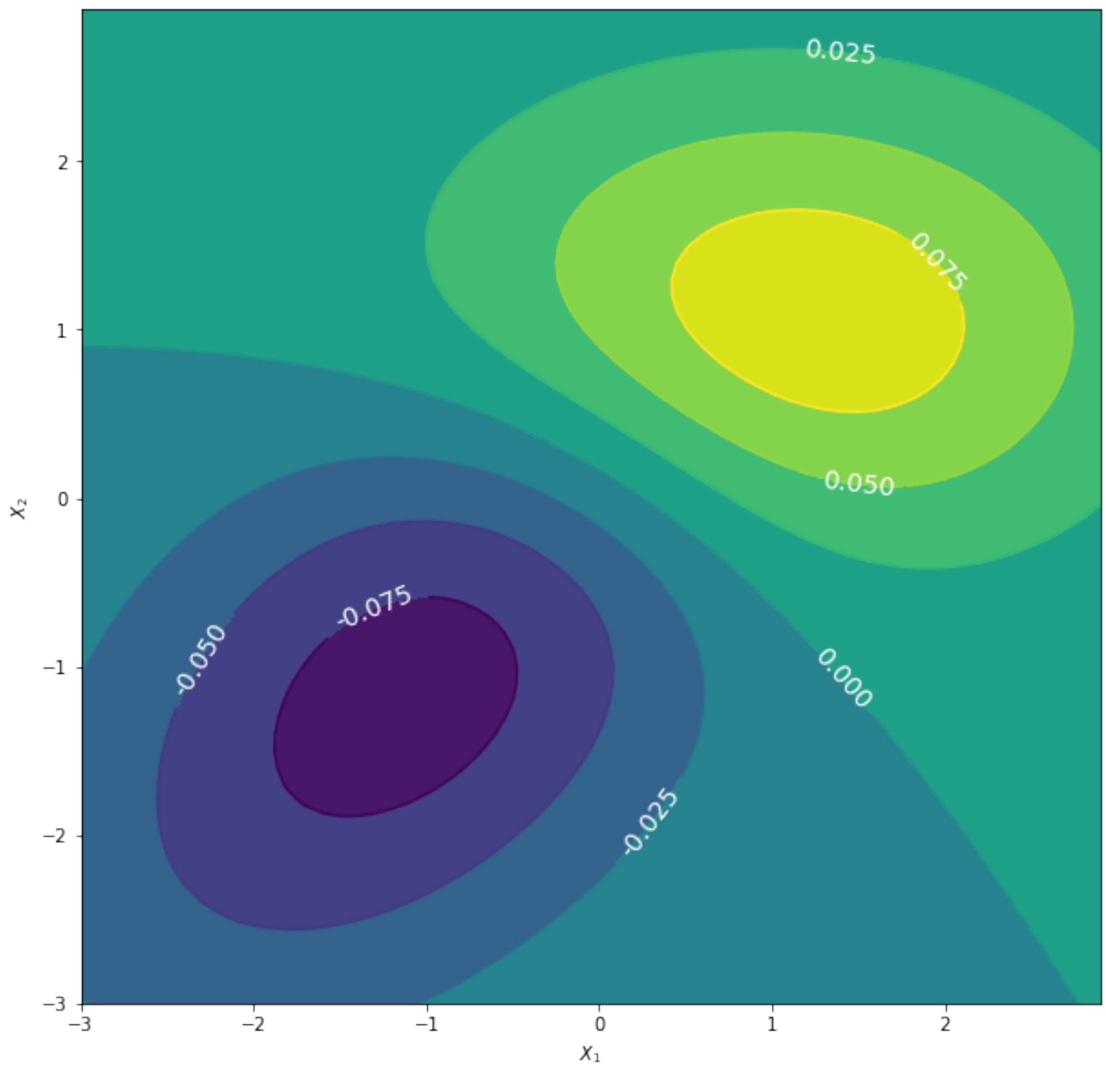

```
In [27]: # (2c)

         # Give the means and covariance matrices(Sigma1,Sigma2) of the multivariat
         mu1,mu2 = np.array([0,2]),np.array([2,0])
         Sigma1 = np.array([[2,1],[1,1]])
         Sigma2 = np.copy(Sigma1)

         figname = 'HW02_prob2c.jpg'
         SolveSubAndPlot(-2.5,4.5,-2,4,delta,mu1,Sigma1,mu2,Sigma2,figname)
```




```
In [28]: # (2d)

         # Give the means and covariance matrices(Sigma1,Sigma2) of the multivariat
         mu1,mu2 = np.array([0,2]),np.array([2,0])
         Sigma1,Sigma2 = np.array([[2,1],[1,1]]),np.array([[2,1],[1,3]])

         figname = 'HW02_prob2d.jpg'
         SolveSubAndPlot(-2.5,4.5,-2,4,delta,mu1,Sigma1,mu2,Sigma2,figname)
```




```
In [29]: # (2e)

         # Give the means and covariance matrices(Sigma1,Sigma2) of the multivariat
         mu1,mu2 = np.array([1,1]),np.array([-1,-1])
         Sigma1,Sigma2 = np.array([[2,0],[0,1]]),np.array([[2,1],[1,2]])

         figname = 'HW02_prob2e.jpg'
         SolveSubAndPlot(-3,3,-3,3,delta,mu1,Sigma1,mu2,Sigma2,figname)
```




```
In [ ]:
```

CS289A_HW03_Prob3

February 27, 2017

In [121]:
```
import random
import numpy as np
from matplotlib import pyplot as plt
```

In [122]:
```
def PlotWithArrows(X,Y,arrowwidth,xmin,xmax,ymin,ymax,figname=None):
# Generate scatter plot of points (X,Y), with arrows representing the eig
# of the covariance matrix, with lengths of the corresponding eigenvalues

    mu = np.array([np.mean(X),np.mean(Y)])
    Sigma = np.cov(X,Y)
    eigvals, eigvecs = np.linalg.eig(Sigma)

    # Adjust eigenvector arrows to have length of eigenvalue
    arrow1lenx,arrow1leny = eigvals[0]*eigvecs[0,0],eigvals[0]*eigvecs[1,
    arrow2lenx,arrow2leny = eigvals[1]*eigvecs[0,1],eigvals[1]*eigvecs[1,

    fig1 = plt.figure(figsize=(10,10))
    plt.scatter(X,Y)
    plt.arrow(mu[0],mu[1],arrow1lenx,arrow1leny,width=arrowwidth)
    plt.arrow(mu[0],mu[1],arrow2lenx,arrow2leny,width=arrowwidth)
    plt.xlim(xmin,xmax)
    plt.ylim(ymin,ymax)
    if figname:
        plt.savefig(figname)
    plt.show()
```

In [123]:
```
# Draw N 2-dimensional points from X1 and X2 given:
#     X1 ~ N(3,9)    and    X2 ~ N(4,4)
# *here, N(u,s) is a normal distribution with mean u and variance s

N = 100
points = np.empty((100,2))
for i in range(N):
    x1 = random.gauss(3,3)
    x2 = 0.5*x1+random.gauss(4,2)
    points[i] = [x1,x2]
X1 = points[:,0]
X2 = points[:,1]
```

```
In [124]: # (3a) Calculate the mean of the sample

          mu = np.mean(points,axis=0)
          print(mu)
```

```
[ 3.18273256  5.09248548]
```

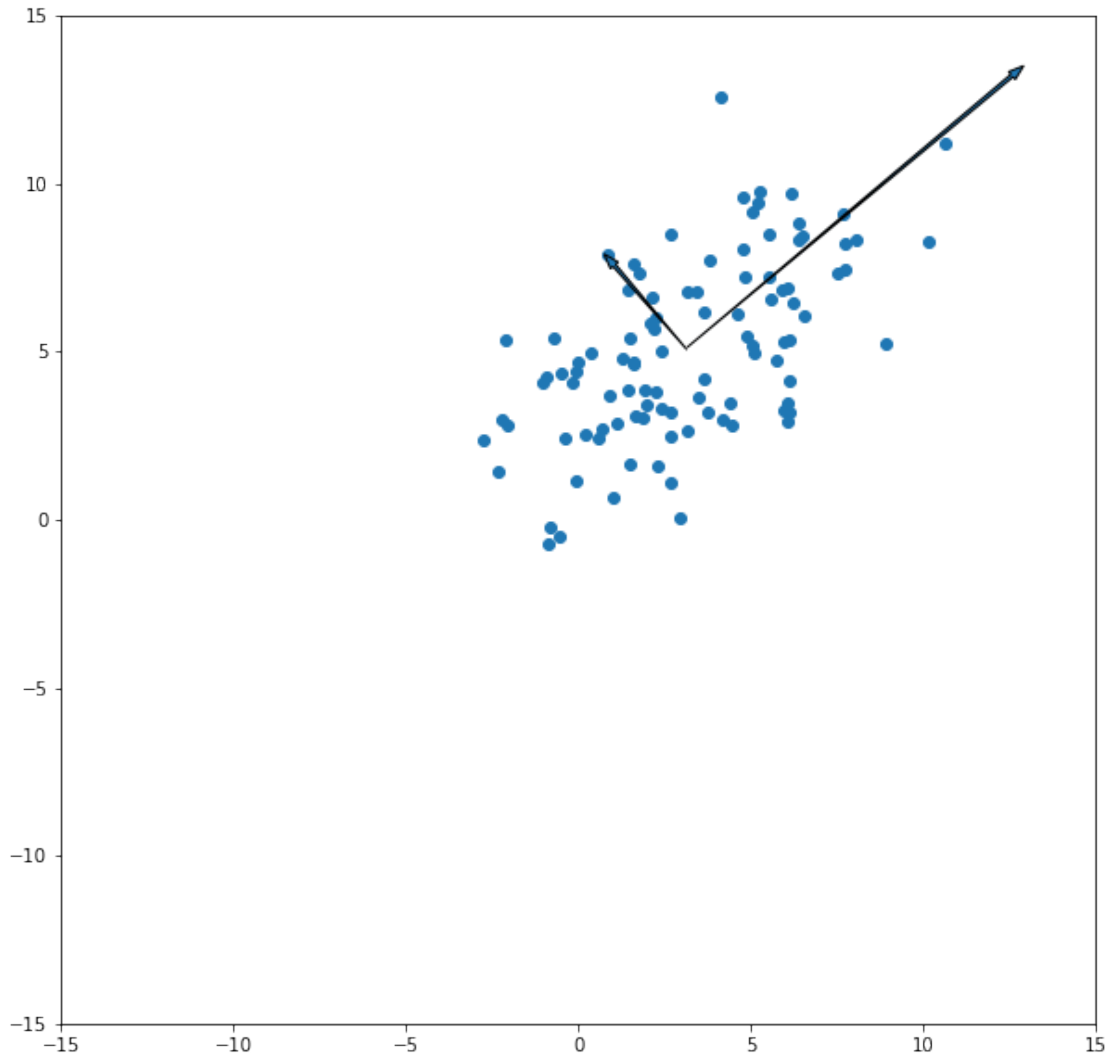
```
In [125]: # (3b) Compute the 2x2 covariance matrix

          Sigma = np.cov(X1,X2)
          print(Sigma)
```

```
[[ 8.4943665   4.52103804]
 [ 4.52103804  7.18995243]]
```

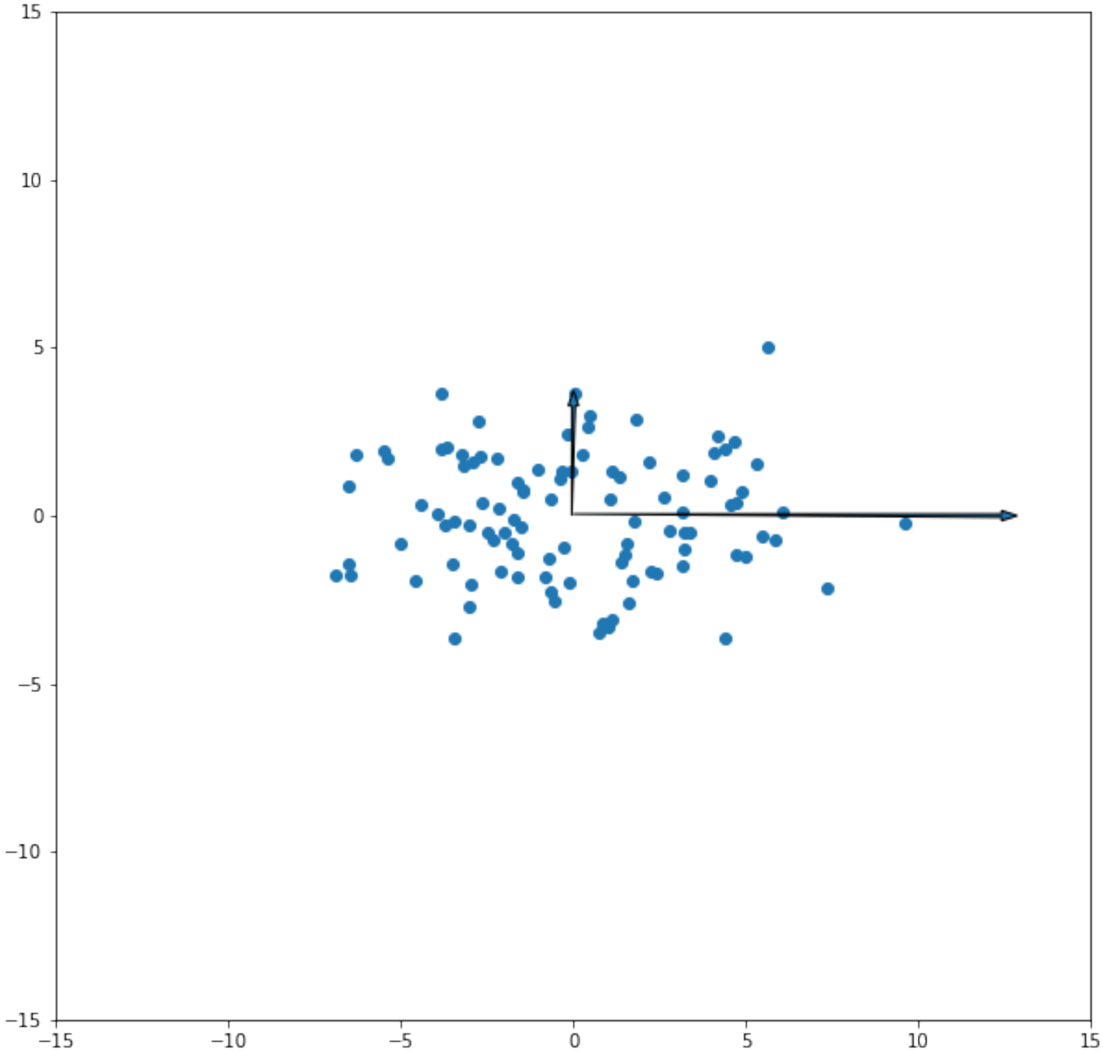
```
In [126]: # (3c) Compute the eigenvectors and eigenvalues of this covariance matrix

          eigvals, eigvecs = np.linalg.eig(Sigma)
          print(eigvals)
          print(eigvecs)
```

```
[ 12.4099991    3.27431983]
[[ 0.75590422 -0.65468222]
 [ 0.65468222  0.75590422]]
```

```
In [127]: # (3d) Plot data points on grid, with arrows representing covariance eige

          PlotWithArrows(X1,X2,0.1,-15,15,-15,15,"HW03_prob3d.jpg")
```




```
In [128]: # (3e) Plot data points on grid, with arrows representing covariance eige

          if eigvals[0] >= eigvals[1]:
              U = eigvecs
          else:
              U = eigvecs[:,::-1]
          centeredpoints = points - mu*np.ones_like(points)
          rotatedpoints = np.empty_like(centeredpoints)
          for i in range(len(rotatedpoints)):
              rotatedpoints[i] = np.dot(U.T,centeredpoints[i])
          Y1 = rotatedpoints[:,0]
          Y2 = rotatedpoints[:,1]

          PlotWithArrows(Y1,Y2,0.1,-15,15,-15,15,"HW03_prob3e.jpg")
```


15
10
5
0
−5
−10
−15
−15
−10
−5
0
5
10
15

CS289A_HW03_Prob6ab

February 27, 2017

First, I load modules to be used in the execution of the problem:

```
In [1]: %load_ext autoreload

In [2]: %autoreload 2

In [3]: import math
        import HW03_utils as ut
        import numpy as np
        from matplotlib import pyplot as plt
```

Then, I define custom functions to be used in the program (the last two are used to calculate the mean and covariance matrix):

```
In [4]: def normalize_images(image_vectors):
        # Function to normalize pixel contrast of images

                magnitudes = np.linalg.norm(image_vectors,axis=1)
                normalized_ims = image_vectors/magnitudes[:,None]
                return normalized_ims

In [5]: def get_class_bounds(classid,labels):
        # Function to extract index bounds of the specified class from the dataset

            for i in range(len(labels)):
                if labels[i] == classid:
                    startindex = i
                    break
            stopindex = len(labels)
            for i in range(i,len(labels)):
                if labels[i] != classid:
                    stopindex = i
                    break

            return startindex,stopindex

In [6]: def get_class_from_data(classid,data,labels):
        # Find the start (inclusive) and end (exclusive) of a class within the data
```

```
        startindex,stopindex = get_class_bounds(classid,labels)

        # Separate the specified class
        class_data = data[startindex:stopindex]

        return class_data
```

```
In [7]: def mean_of_class(classid,data,labels):
        # Calculate the mean value when the class is fit to a normal distribution

            class_data = get_class_from_data(classid,data,labels)
            # Calculate the mean of the class data
            class_mu = np.mean(class_data,axis=0)

            return class_mu
```

```
In [8]: def cov_of_class(classid,data,labels):
        # Calcualte the covariance matrix when the class is fit to a normal distrib

            class_data = get_class_from_data(classid,data,labels)
            # Calculate the covariance matrix from the class data
            class_Sigma = np.cov(class_data,rowvar=False)

            return class_Sigma
```

Now comes the program execution. To start, I specify local paths to the data and then load it into memory.

```
In [9]: CS_DIR = r"/Users/mitch/Documents/Cal/2 - 2017 Spring/COMPSCI 289A - Intro
```

```
In [10]: # Load MNIST data
         data_array = ut.loaddata("hw3_mnist_dist/hw3_mnist_dist/train.mat",CS_DIR+
```

Immediately after loading the data, I shuffle it and then separate it into data and labels.

```
In [11]: # Shuffle data and set aside validation set
         np.random.shuffle(data_array)

         trainarray = data_array[:-10000]
         valarray = data_array[-10000:]

         # Organize array by digit
         trainarray_byclass = trainarray[trainarray[:,-1].argsort()]
         valarray_byclass = valarray[valarray[:,-1].argsort()]
```

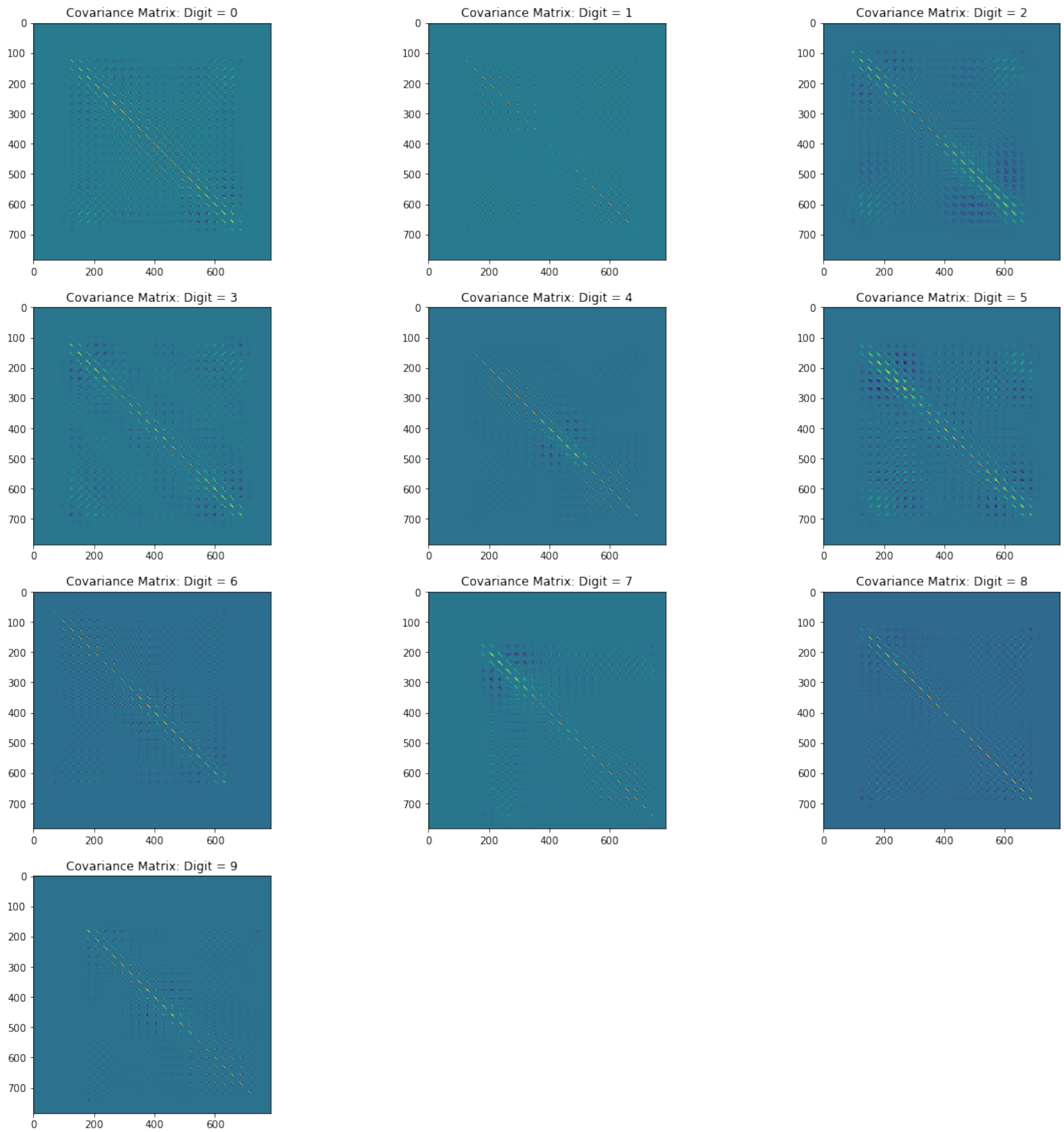
```
In [12]: train_data = trainarray_byclass[:,:-1]
         train_labels = trainarray_byclass[:,-1]

         val_data = valarray_byclass[:,:-1]
         val_labels = valarray_byclass[:,-1]
```

To maintain consistency between calculations, I normalize all the images using a custom defined function (given above).

```
In [13]: normalized_traindata = normalize_images(train_data)
         normalized_valdata = normalize_images(val_data)
```

For each digit, I calculate the mean and covariance matrix and then plot both.

```
In [14]: fig = plt.figure(figsize=(20,20))
         for i in range(10):
             mu_i = mean_of_class(i,normalized_traindata,train_labels)
             Sigma_i = cov_of_class(i,normalized_traindata,train_labels)

             plt.subplot(4,3,i+1)
             plt.imshow(Sigma_i)
             plt.title('Covariance Matrix: Digit = %i' %i)
         plt.savefig('VisualCovMatrices.jpg')
         plt.show()
```


Covariance Matrix: Digit = 0
Covariance Matrix: Digit = 1
Covariance Matrix: Digit = 2
Covariance Matrix: Digit = 3
Covariance Matrix: Digit = 4
Covariance Matrix: Digit = 5
Covariance Matrix: Digit = 6
Covariance Matrix: Digit = 7
Covariance Matrix: Digit = 8
Covariance Matrix: Digit = 9
0
100
200
300
400
500
600
700

CS289A_HW03_Prob6c

February 28, 2017

Load modules to be used in the execution of the problem.

```
In [1]: %load_ext autoreload
```

```
In [2]: %autoreload 2
```

```
In [3]: import math
        import HW03_utils as ut
        import numpy as np
        from matplotlib import pyplot as plt
```

```
In [4]: def normalize_images(image_vectors):
        # Function to normalize pixel contrast of images

                magnitudes = np.linalg.norm(image_vectors,axis=1)
                normalized_ims = image_vectors/magnitudes[:,None]
                return normalized_ims
```

```
In [5]: def get_class_bounds(classid,labels):
        # Function to extract index bounds of the specified class from the dataset

            for i in range(len(labels)):
                if labels[i] == classid:
                    startindex = i
                    break
            stopindex = len(labels)
            for i in range(i,len(labels)):
                if labels[i] != classid:
                    stopindex = i
                    break

            return startindex,stopindex
```

```
In [6]: def get_class_from_data(classid,data,labels):
        # Find the start (inclusive) and end (exclusive) of a class within the data

            startindex,stopindex = get_class_bounds(classid,labels)
```

```
        # Separate the specified class
        class_data = data[startindex:stopindex]

        return class_data
```

```
In [7]: def mean_of_class(classid,data,labels):
        # Calculate the mean value when the class is fit to a normal distribution

            class_data = get_class_from_data(classid,data,labels)
            # Calculate the mean of the class data
            class_mu = np.mean(class_data,axis=0)

            return class_mu
```

```
In [8]: def cov_of_class(classid,data,labels):
        # Calcualte the covariance matrix when the class is fit to a normal distrib

            class_data = get_class_from_data(classid,data,labels)
            # Calculate the covariance matrix from the class data
            class_Sigma = np.cov(class_data,rowvar=False)

            return class_Sigma
```

```
In [9]: def calc_SigmaHat(data,labels,muCs):
        # Function to calculate the average covariance matrix for the distribution

            SigmaHat = np.zeros((len(data[0]),len(data[0])))
            for i in range(len(data)):
                Xi_minus_muC = data[i] - muCs[labels[i]]
                SigmaHat += np.outer(Xi_minus_muC,Xi_minus_muC)

            SigmaHat = SigmaHat/len(labels)

            return SigmaHat
```

```
In [10]: def zero_rows(sym_matrix):
         # Take a symmetric matrix and find rows/columns that are empty

             zero_rows = []
             for i in range(len(sym_matrix)):
                 if not np.any(sym_matrix[i]):
                     zero_rows.append(i)

             return zero_rows
```

```
In [11]: def makeInvertible(sym_matrix):
         # Take a symmetric non-invertible matrix, and eliminate rows/columns to ma

             ZR = zero_rows(sym_matrix)
```

```
        newlen = len(sym_matrix)-len(ZR)
        invmatrix = np.empty((newlen,newlen))
        I = 0
        for i in range(len(sym_matrix)):
            if i in ZR:
                continue
            J = 0
            for j in range(len(sym_matrix)):
                if j in ZR:
                    continue
                invmatrix[I,J] = sym_matrix[i,j]
                J += 1
            I += 1

        return invmatrix
```

In [12]:
```
def removeZeroVariance(cov_matrix,valdata):
    # Remove variables with zero variance in the covariance matrix from the va
    #       -valdata is a nxd array with n rows of samples and d-variables per
    #       -Cov_matrix is a dxd matrix giving the covariances of the d-variabl

        ZR = zero_rows(cov_matrix)
        # Create a new array with validation data corresponding to variables i
        NZV_data = np.empty((len(valdata),len(valdata[0])-len(ZR)))
        columnI = 0
        for columni in range(len(valdata[0])):
            if columni in ZR:
                continue
            NZV_data[:,columnI] = valdata[:,columni]
            columnI += 1

        return NZV_data
```

In [13]:
```
def MuAndPi(train_labels,muCs,cov_matrix,ZR):
        mu_i = []
        pi_i = []
        for i in range(10):
            # Calculate the mean (without zero variance variables)
            mu_i.append(np.zeros(np.shape(cov_matrix)[0]))
            J = 0
            for j in range(np.shape(muCs[i])[0]):
                if j in ZR:
                    continue
                mu_i[i][J] = muCs[i][j]
                J += 1

            # Calculate the prior probability
            startindex,stopindex = get_class_bounds(i,train_labels)
```

```
            nPoints = stopindex-startindex
            pi_i.append(nPoints/np.shape(train_labels)[0])

        return mu_i,pi_i
```

In [14]:
```
def LDF_solve(X,mu_C,Sigma,pi_C=0.1):
    # Function to solve the linear discriminant function for class C (will co

        LDFs_C = np.zeros(len(X))
        muCinvSigma = np.dot(mu_C,np.linalg.pinv(Sigma))
        muCinvSigmamuC = np.dot(muCinvSigma,mu_C)
        logpiC = math.log(pi_C)
        for i in range(len(X)):
            x = X[i]
            LDFs_C[i] = np.dot(muCinvSigma,x) - 0.5*muCinvSigmamuC + logpiC

        return LDFs_C
```

In [15]:
```
def maximize_LDFs(valdata,mu_i,cov_matrix,pi_i):
        lin_disc_fns = np.empty((len(valdata),10))
        for i in range(10):
            mu_C = mu_i[i]
            pi_C = pi_i[i]
            lin_disc_fns[:,i] = LDF_solve(valdata,mu_C,cov_matrix,pi_C)
        max_LDF_indices = np.empty(len(valdata))
        for i in range(len(valdata)):
            max_LDF_indices[i] = np.argmax(lin_disc_fns[i])

        return max_LDF_indices
```

In [16]:
```
CS_DIR = r"/Users/mitch/Documents/Cal/2 - 2017 Spring/COMPSCI 289A - Intro
```

In [17]:
```
# Load MNIST data
data_array = ut.loaddata("hw3_mnist_dist/hw3_mnist_dist/train.mat",CS_DIR+
```

In [18]:
```
# Shuffle data and set aside validation set
np.random.shuffle(data_array)

trainarray = data_array[:-10000]
valarray = data_array[-10000:]
```

In [19]:
```
def main(traindata,trainlabels,valdata,vallabels):
    # Main block of code

        # Create a list of the means for each class
        muCs = np.empty((10,len(traindata[0])))
        for i in range(10):
            muCs[i] = mean_of_class(i,traindata,trainlabels)
```

```
        SigmaHat = calc_SigmaHat(traindata,trainlabels,muCs)

        newcov = makeInvertible(SigmaHat)
        newvaldata = removeZeroVariance(SigmaHat,valdata)

        ZR = zero_rows(SigmaHat)

        mu_i,pi_i = MuAndPi(trainlabels,muCs,newcov,ZR)

        digitPicks = maximize_LDFs(newvaldata,mu_i,newcov,pi_i)
        count, total = 0,0
        for i in range(len(digitPicks)):
            if digitPicks[i] == vallabels[i]:
                count += 1
            total += 1

        # VERBOSE COMMANDS FOR WATCHING PROGRESS [OPTIONAL]
        #    if total%200 == 0:
        #        print(total,'points evaluated; current score =',count/total)

        score = count/total

        return score
```

```
In [20]: # Organize array by digit
         trainarray_byclass = trainarray[trainarray[:,-1].argsort()]
         valarray_byclass = valarray[valarray[:,-1].argsort()]

         train_data = trainarray_byclass[:,:-1]
         train_labels = trainarray_byclass[:,-1]

         val_data = valarray_byclass[:,:-1]
         val_labels = valarray_byclass[:,-1]

         normalized_traindata = normalize_images(train_data)
         normalized_valdata = normalize_images(val_data)
```

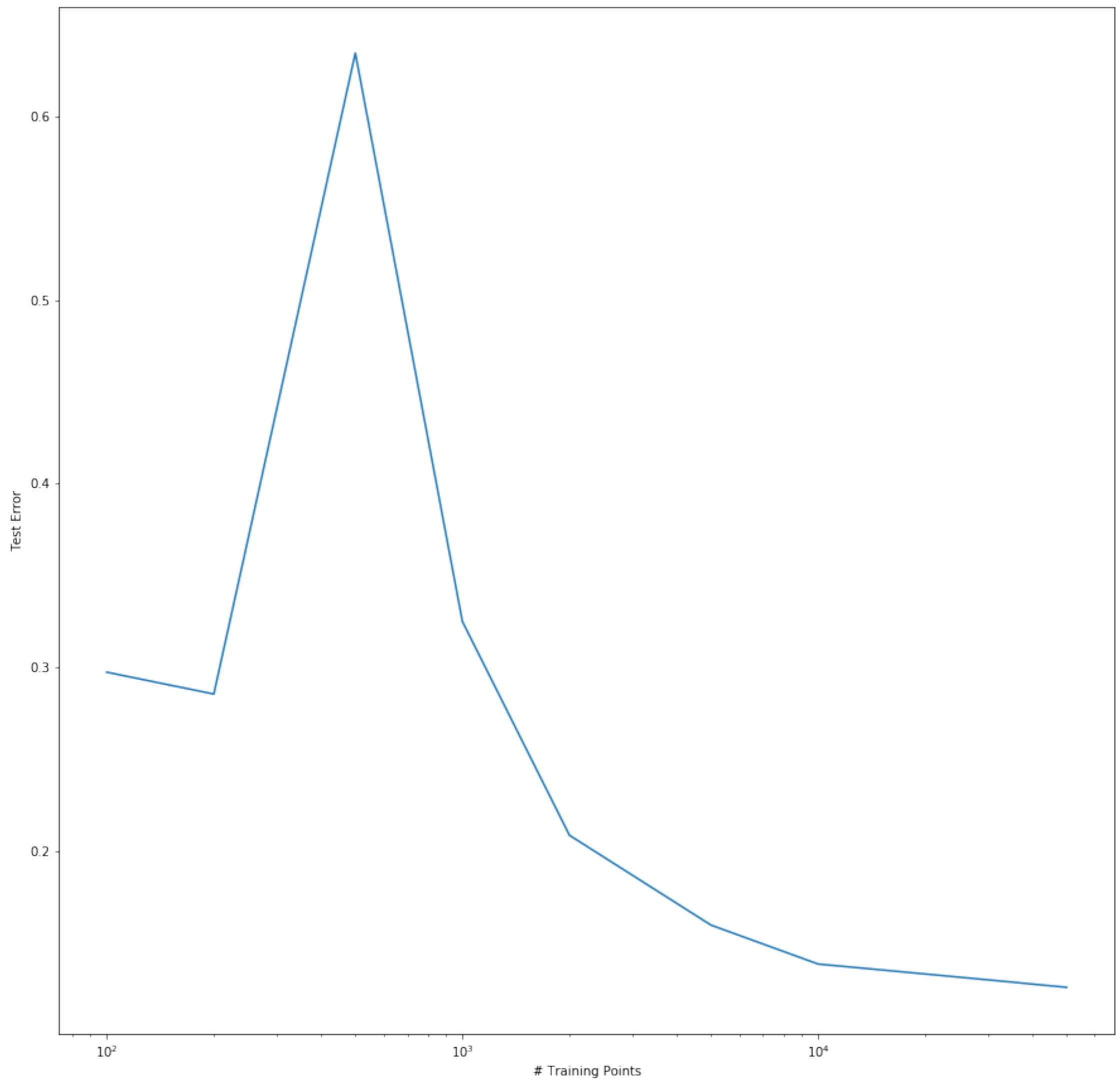
```
In [25]: samples = [100,200,500,1000,2000,5000,10000,30000,50000]
```

```
In [21]: # Train on subsets of full training data set
         scores = []
         for number in samples:
             trainarraysubset = trainarray[:number]

             # Organize array by digit
             trainarray_byclass = trainarraysubset[trainarraysubset[:,-1].argsort()
```

```
        valarray_byclass = valarray[valarray[:,-1].argsort()]

        # Separate data and labels
        train_data = trainarray_byclass[:,:-1]
        train_labels = trainarray_byclass[:,-1]
        val_data = valarray_byclass[:,:-1]
        val_labels = valarray_byclass[:,-1]

        # Normalize training and validation data
        normalized_train_data = normalize_images(train_data)
        normalized_val_data = normalize_images(val_data)

        print(number,"training samples: ")
        score = main(normalized_train_data,train_labels,normalized_val_data,va
        scores.append(score)
        print(score)
```

```
100 training samples:
0.7026
200 training samples:
0.7145
500 training samples:
0.3655
1000 training samples:
0.675
2000 training samples:
0.7914
5000 training samples:
0.8403
10000 training samples:
0.8615
30000 training samples:
0.8701
50000 training samples:
0.8742
```

```
In [30]: errors = np.ones(len(scores))-np.array(scores)

         fig = plt.figure(figsize=(15,15))
         plt.semilogx(samples,error)
         plt.xlabel("# Training Points")
         plt.ylabel("Test Error")
         plt.savefig("LDA_errors.jpg")
         plt.show()
```




```
In [31]: print(errors)

[ 0.2974  0.2855  0.6345  0.325   0.2086  0.1597  0.1385  0.1299  0.1258]
```

```
In [ ]:
```

CS289A_HW03_Prob6c2

February 28, 2017

Load modules to be used in the execution of the problem.

```
In [1]: %load_ext autoreload
```

```
In [2]: %autoreload 2
```

```
In [3]: import math
        import HW03_utils as ut
        import numpy as np
        from matplotlib import pyplot as plt
```

```
In [4]: def normalize_images(image_vectors):
        # Function to normalize pixel contrast of images

                magnitudes = np.linalg.norm(image_vectors,axis=1)
                normalized_ims = image_vectors/magnitudes[:,None]
                return normalized_ims
```

```
In [5]: def get_class_bounds(classid,labels):
        # Function to extract index bounds of the specified class from the dataset

            for i in range(len(labels)):
                if labels[i] == classid:
                    startindex = i
                    break
            stopindex = len(labels)
            for i in range(i,len(labels)):
                if labels[i] != classid:
                    stopindex = i
                    break

            return startindex,stopindex
```

```
In [6]: def get_class_from_data(classid,data,labels):
        # Find the start (inclusive) and end (exclusive) of a class within the data

            startindex,stopindex = get_class_bounds(classid,labels)
```

```
        # Separate the specified class
        class_data = data[startindex:stopindex]

        return class_data
```

```
In [7]: def mean_of_class(classid,data,labels):
    # Calculate the mean value when the class is fit to a normal distribution

        class_data = get_class_from_data(classid,data,labels)
        # Calculate the mean of the class data
        class_mu = np.mean(class_data,axis=0)

        return class_mu
```

```
In [8]: def cov_of_class(classid,data,labels):
    # Calcualte the covariance matrix when the class is fit to a normal distrib

        class_data = get_class_from_data(classid,data,labels)
        # Calculate the covariance matrix from the class data
        class_Sigma = np.cov(class_data,rowvar=False)

        return class_Sigma
```

```
In [9]: def Prior(classid,data_labels):

        # Calculate the prior probability
        startindex,stopindex = get_class_bounds(classid,data_labels)
        nPoints = stopindex-startindex
        pi_i = nPoints/len(data_labels)

        return pi_i
```

```
In [10]: def QDF_solve(X,muC,SigmaC,piC=0.1):
     # Function to solve the linear discriminant function for class C (will con

         QDFs_C = np.zeros(len(X))
         invSigmaC = np.linalg.pinv(SigmaC)
         detSigmaC = np.linalg.det(SigmaC)
         print('det ',detSigmaC)
         lndetSigmaC = np.log(detSigmaC)
         lnpiC = math.log(piC)
         for i in range(len(X)):
             x = X[i]
             QDFs_C[i] = -0.5*np.dot(np.dot((x-muC),invSigmaC),(x-muC))-0.5*lnd

         return QDFs_C
```

```
In [11]: def maximize_QDFs(quad_disc_fns):
         max_QDF_indices = np.empty(len(quad_disc_fns))
```

```
        for i in range(len(max_QDF_indices)):
            max_QDF_indices[i] = np.argmax(quad_disc_fns[i])

        return max_QDF_indices
```

```
In [12]: CS_DIR = r"/Users/mitch/Documents/Cal/2 - 2017 Spring/COMPSCI 289A - Intro
```

```
In [13]: # Load MNIST data
         data_array = ut.loaddata("hw3_mnist_dist/hw3_mnist_dist/train.mat",CS_DIR+
```

```
In [14]: # Shuffle data and set aside validation set
         np.random.shuffle(data_array)

         trainarray = data_array[:-10000]
         valarray = data_array[-10000:]
```

```
In [15]: def findRedundants(sym_matrix):
         # Take a symmetric matrix and find rows/columns that are redundant

             red_rows = []
             for i in range(len(sym_matrix)):
                 if not np.any(sym_matrix[i]):
                     red_rows.append(i)

             return red_rows
```

```
In [16]: def removeRedundants(matrix,red_vecs_inds):
         # Eliminate redundant vectors from a matrix, or elements from
         # a vector corresponding to redundant rows/columns in a matrix

             newlen = len(matrix)-len(red_vecs_inds)
             if len(np.shape(matrix))==2:
                 newmatrix = np.empty((newlen,newlen))
                 I = 0
                 for i in range(len(matrix)):
                     if i in red_vecs_inds:
                         continue
                     J = 0
                     for j in range(len(matrix)):
                         if j in red_vecs_inds:
                             continue
                         newmatrix[I,J] = matrix[i,j]
                         J += 1
                     I += 1

                 return newmatrix

             if len(np.shape(matrix))==1:
                 newvector = np.empty(newlen)
```

```
    I = 0
    for i in range(len(matrix)):
        if i in red_vecs_inds:
            continue
        newvector[I] = matrix[i]
        I+=1

    return newvector
```

In [17]:
```
def main(traindata,trainlabels,valdata,vallabels):
# Main block of code

    quad_disc_fns = np.empty((len(valdata),10))
    for i in range(10):
        muC = mean_of_class(i,traindata,trainlabels)
        SigmaC = cov_of_class(i,traindata,trainlabels)
        sigvals = []
        for u in SigmaC:
            for v in u:
                if v!= 0:
                    sigvals.append(v)
        print(sigvals)
        piC = Prior(i,trainlabels)

        RedVarInds = findRedundants(SigmaC)
        newmuC = removeRedundants(muC,RedVarInds)
        newSigmaC = removeRedundants(SigmaC,RedVarInds)
        print(np.shape(newSigmaC))
        newvaldata = np.empty((len(valdata),len(valdata[0])-len(RedVarInds
        for datapointi in range(len(valdata)):
            newvaldata[datapointi] = removeRedundants(valdata[datapointi],

        quad_disc_fns[:,i] = QDF_solve(newvaldata,newmuC,newSigmaC,piC)

        digitPicks = maximize_QDFs(quad_disc_fns)

    count, total = 0,0
    for i in range(len(digitPicks)):
        if digitPicks[i] == vallabels[i]:
            count += 1
        total += 1

    # VERBOSE COMMANDS FOR WATCHING PROGRESS [OPTIONAL]
    #    if total%200 == 0:
    #        print(total,'points evaluated; current score =',count/total)
    print(count,total)
```

```
        score = count/total

        return score
```

```
In [18]: # Organize array by digit
         trainarray_byclass = trainarray[trainarray[:,-1].argsort()]
         valarray_byclass = valarray[valarray[:,-1].argsort()]

         train_data = trainarray_byclass[:,:-1]
         train_labels = trainarray_byclass[:,-1]

         val_data = valarray_byclass[:,:-1]
         val_labels = valarray_byclass[:,-1]

         normalized_traindata = normalize_images(train_data)
         normalized_valdata = normalize_images(val_data)
```

```
In [ ]: samples = [100,200,500,1000,2000,5000,10000,30000,50000]
```

```
In [ ]: # Train on subsets of full training data set
        scores = []
        for number in samples:
            trainarraysubset = trainarray[:number]

            # Organize array by digit
            trainarray_byclass = trainarraysubset[trainarraysubset[:,-1].argsort()]
            valarray_byclass = valarray[valarray[:,-1].argsort()]

            # Separate data and labels
            train_data = trainarray_byclass[:,:-1]
            train_labels = trainarray_byclass[:,-1]
            val_data = valarray_byclass[:,:-1]
            val_labels = valarray_byclass[:,-1]

            # Normalize training and validation data
            normalized_train_data = normalize_images(train_data)
            normalized_val_data = normalize_images(val_data)

            print(number,"training samples: ")
            score = main(normalized_train_data,train_labels,normalized_val_data,val
            scores.append(score)
            print(score)
```

```
In [ ]: errors = np.ones(len(scores))-np.array(scores)

        fig = plt.figure(figsize=(15,15))
        plt.semilogx(samples,error)
```

```python
        plt.xlabel("# Training Points")
        plt.ylabel("Test Error")
        plt.savefig("LDA_errors.jpg")
        plt.show()
```

In [ ]:

```python
print(errors)
```

In [ ]: