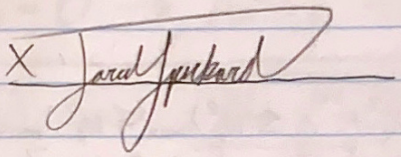
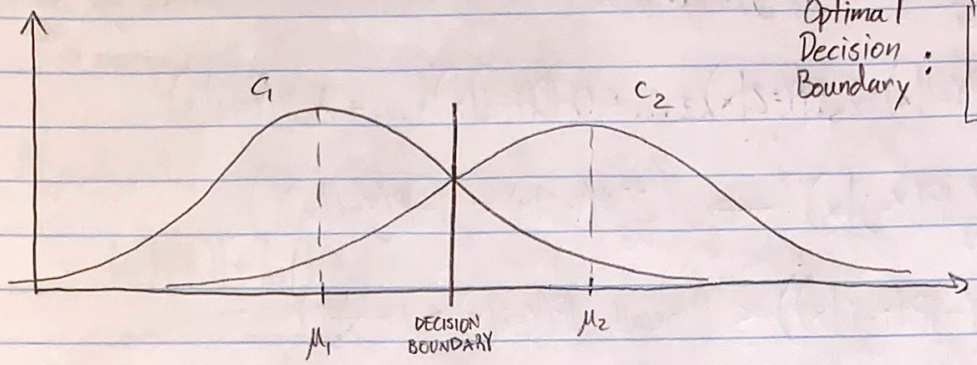
CS189 HW 3

(1) I certify that all solutions are entirely my own and that I have not looked at anyone else's solution. I have given credit to all external sources I consulted.

X 

Code Appendix

(2) 1) For 2 Gaussian Distributions w/ the same standard deviations, means μ_1 and μ_2, and $\mathbb{P}(C_1) = \mathbb{P}(C_2) = 1/2$, we can visualize the distributions as follows:



Optimal Decision Boundary: $x = \frac{\mu_1 + \mu_2}{2}$

Decision Rule: $r(x) = \begin{cases} C_1 \text{ if } x \leq \frac{\mu_1+\mu_2}{2} \\ C_2 \text{ otherwise} \end{cases}$

2) $\mathbb{P}(\text{misclassify as } C_1 | C_2)\mathbb{P}(C_2) + \mathbb{P}(\text{misclassify as } C_2 | C_1)\mathbb{P}(C_1)$

$= 1/2\left(\mathbb{P}(\text{misclassify as } C_1 | C_2) + \mathbb{P}(\text{misclassify as } C_2 | C_1)\right)$ ← by symmetry

$= 1/2 \cdot 2\left(\mathbb{P}(\text{misclassify as } C_2 | C_1)\right)$

$= \mathbb{P}(\text{misclassify as } C_2 | C_1) = \mathbb{P}(x > \bar{\mu} \mid x \in C_1)$ define $\bar{\mu} = \frac{\mu_1+\mu_2}{2}$

$$\mathbb{P}(x > \bar{\mu} \mid x \in C_1) = \int_{\bar{\mu}}^{\infty} f_{C_1}(x) = \int_{\bar{\mu}}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\|x-\mu_1\|^2}{2\sigma^2}} dx$$

$z = \frac{\|x-\mu_1\|}{\sigma}$, $dz = \frac{1}{\sigma} dx$

$$= \boxed{\frac{1}{\sqrt{2\pi}} \int_a^{\infty} e^{-\frac{z^2}{2}} dz}$$

$$a = \frac{\|\bar{\mu} - \mu_1\|}{\sigma} = \frac{\frac{\mu_1+\mu_2}{2} - \mu_1}{\sigma} = \frac{\mu_1 + \mu_2 - 2\mu_1}{2\sigma} = \frac{\mu_2 - \mu_1}{2\sigma}$$

$$a = \frac{\mu_2 - \mu_1}{2\sigma}$$

✓

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

def getPDF(mu, sigma):
    """returns X, Y, and the pdf of the Gaussian distribution with mean mu and covariance sigma"""
    eigs = np.linalg.eig(sigma)[0]
    std_devs = [math.sqrt(eig) for eig in eigs]
    x = np.linspace(mu[0]-3*std_devs[0], mu[0]+3*std_devs[0], 100)
    y = np.linspace(mu[1]-3*std_devs[1], mu[1]+3*std_devs[1], 100)
    X,Y = np.meshgrid(x, y)
    pos = np.dstack((X,Y))
    rv = multivariate_normal(mean=mu, cov=sigma)
    return X, Y, rv.pdf(pos)

def getDifInPDF(mu1, mu2, sigma1, sigma2):
    """returns X, Y, and the differnece between the pdfs of the Gaussian distributions with means mu1, mu2 and covariances sigma1, sigma2"""
    eigs1 = np.linalg.eig(sigma1)[0]
    eigs2 = np.linalg.eig(sigma2)[0]
    std_devs1 = [math.sqrt(eig) for eig in eigs1]
    std_devs2 = [math.sqrt(eig) for eig in eigs2]
    min_x = min(mu1[0]-3*std_devs1[0], mu2[0]-3*std_devs2[0])
    max_x = max(mu1[0]+3*std_devs1[0], mu2[0]+3*std_devs2[0])
    min_y = min(mu1[1]-3*std_devs1[1], mu2[1]-3*std_devs2[1])
    max_y = max(mu1[1]+3*std_devs1[1], mu2[1]+3*std_devs2[1])
    x = np.linspace(min_x, max_x, 100)
    y = np.linspace(min_y, max_y, 100)
    X,Y = np.meshgrid(x, y)
    pos = np.dstack((X,Y))
    rv1 = multivariate_normal(mean=mu1, cov=sigma1)
    rv2 = multivariate_normal(mean=mu2, cov=sigma2)
    return X, Y, rv1.pdf(pos)-rv2.pdf(pos)

def plot1(mu, sigma, part_num):
    """Plots the isocontour of the pdf of a Gaussian distribution with mean mu and covariance sigma.
    Used for parts 1 and 2"""
    X, Y, pdf = getPDF(mu, sigma)
    fig, ax = plt.subplots()
    ax.set_aspect(1)
    cs = ax.contour(X, Y, pdf)
    ax.clabel(cs, inline=1, fontsize=15)
    plt.title("Question 3 - Part " + str(part_num))
    #plt.contourf(X, Y, rv.pdf(pos), 10)
    plt.show()

def plot2(mu1, mu2, sigma1, sigma2, part_num) :
    """Plots the isocontour of the pdf of a Gaussian distribution (mu1, sigma1) - the pdf of the Gaussian distr (mu2, sigma2).
    X ~ N(m1,s1), Y ~ N(m2, s2), X-Y ~ N(m1-m2,s1+s2)
    Used for parts 3-5"""
    X, Y, dif = getDifInPDF(mu1, mu2, sigma1, sigma2)
    fig, ax = plt.subplots()
    cs = ax.contour(X, Y, dif)
    ax.set_aspect(1)
    ax.clabel(cs, inline=1, fontsize=15)
    plt.title("Question 3 - Part " + str(part_num))
    #plt.contourf(X, Y, rv.pdf(pos), 10)
    plt.show()
```

```python
# Part 1 /////////////////////////////////////////////////////////////////////
def part1():
    mu = np.array([1,1])
    sigma = np.array([[1,0],[0,2]])
    plot1(mu, sigma, 1)
# Part 2 /////////////////////////////////////////////////////////////////////
def part2():
    mu = np.array([-1,2])
    sigma = np.array([[2,1],[1,4]])
    plot1(mu, sigma, 2)
# Part 3 /////////////////////////////////////////////////////////////////////
def part3():
    mu1 = np.array([0,2])
    mu2 = np.array([2,0])
    sigma1 = np.array([[2,1],[1,1]])
    sigma2 = np.array([[2,1],[1,1]])
    plot2(mu1,mu2,sigma1,sigma2, 3)
# Part 4 /////////////////////////////////////////////////////////////////////
def part4():
    mu1 = np.array([0,2])
    mu2 = np.array([2,0])
    sigma1 = np.array([[2,1],[1,1]])
    sigma2 = np.array([[2,1],[1,4]])
    plot2(mu1,mu2,sigma1,sigma2, 4)
# Part 5 /////////////////////////////////////////////////////////////////////
def part5():
    mu1 = np.array([1,1])
    mu2 = np.array([-1,-1])
    sigma1 = np.array([[2,0],[0,1]])
    sigma2 = np.array([[2,1],[1,2]])
    plot2(mu1,mu2,sigma1,sigma2, 5)


"""Uncomment one of the following lines of code in order to run the code for the associated part"""

part1()
part2()
part3()
part4()
part5()
```

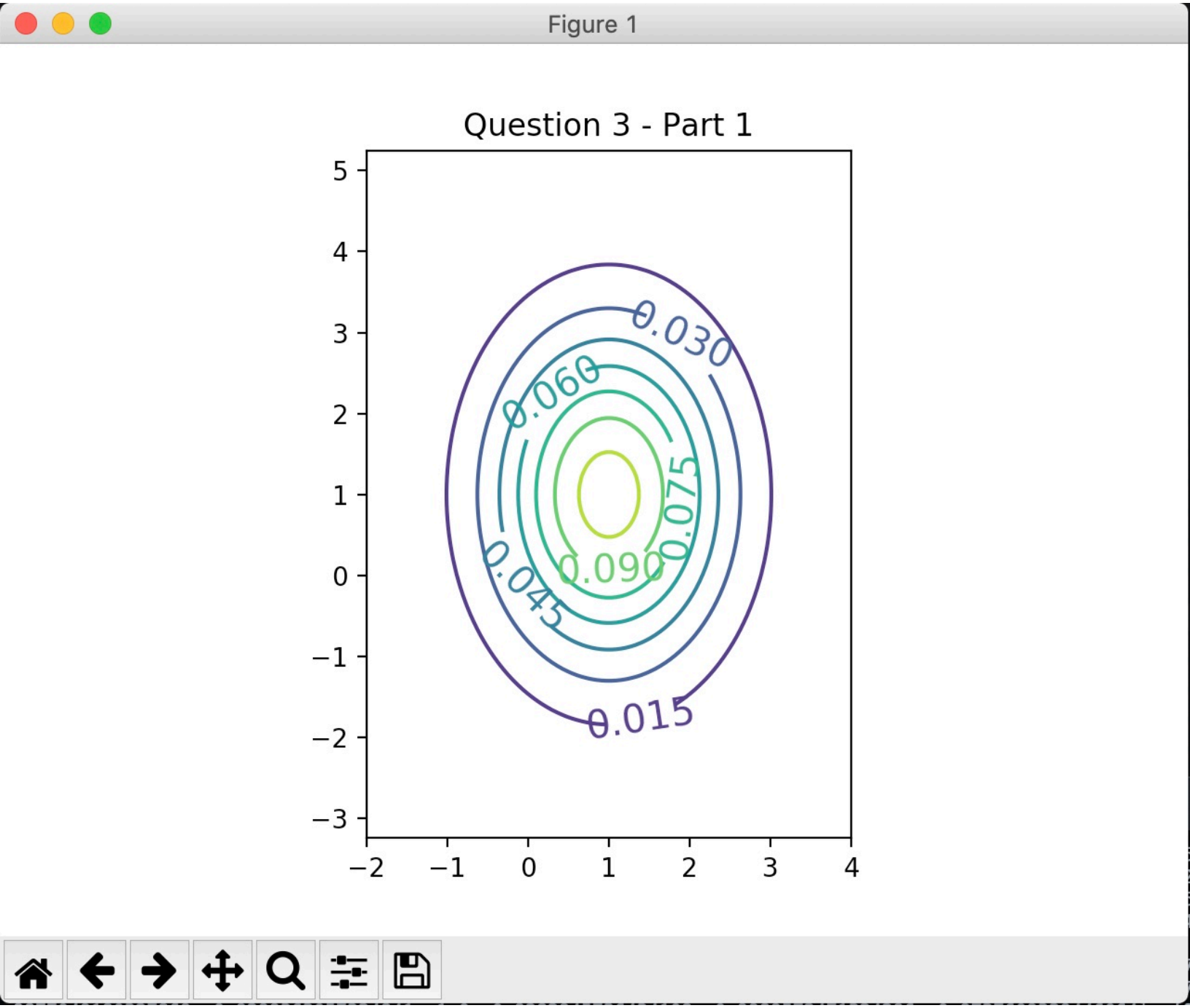


Figure 1
Question 3 - Part 1
5
4
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
4
0.030
0.060
0.075
0.090
0.045
0.015

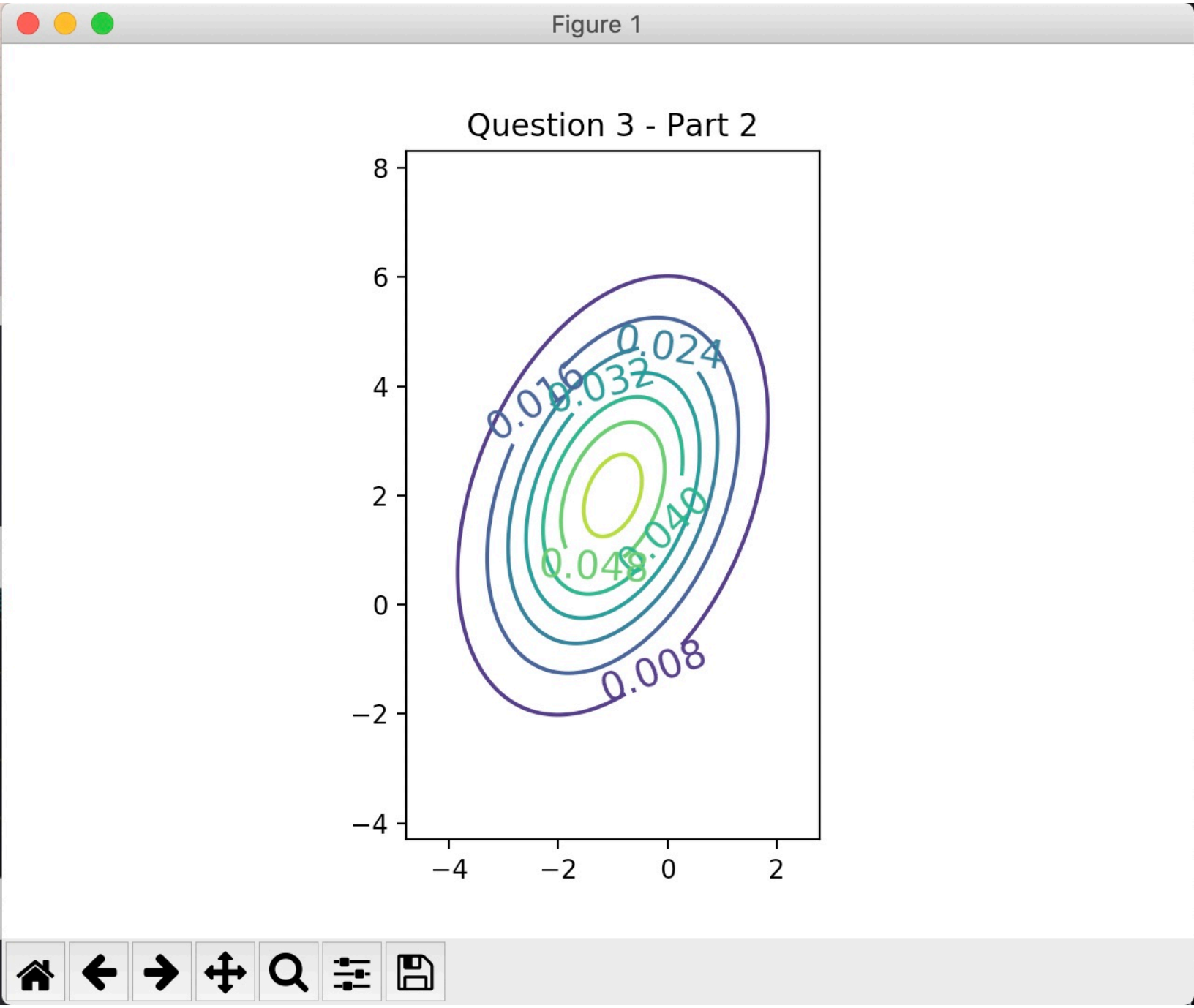


Figure 1
Question 3 - Part 2
8
6
4
2
0
−2
−4
−4
−2
0
2
0.024
0.032
0.040
0.048
0.008

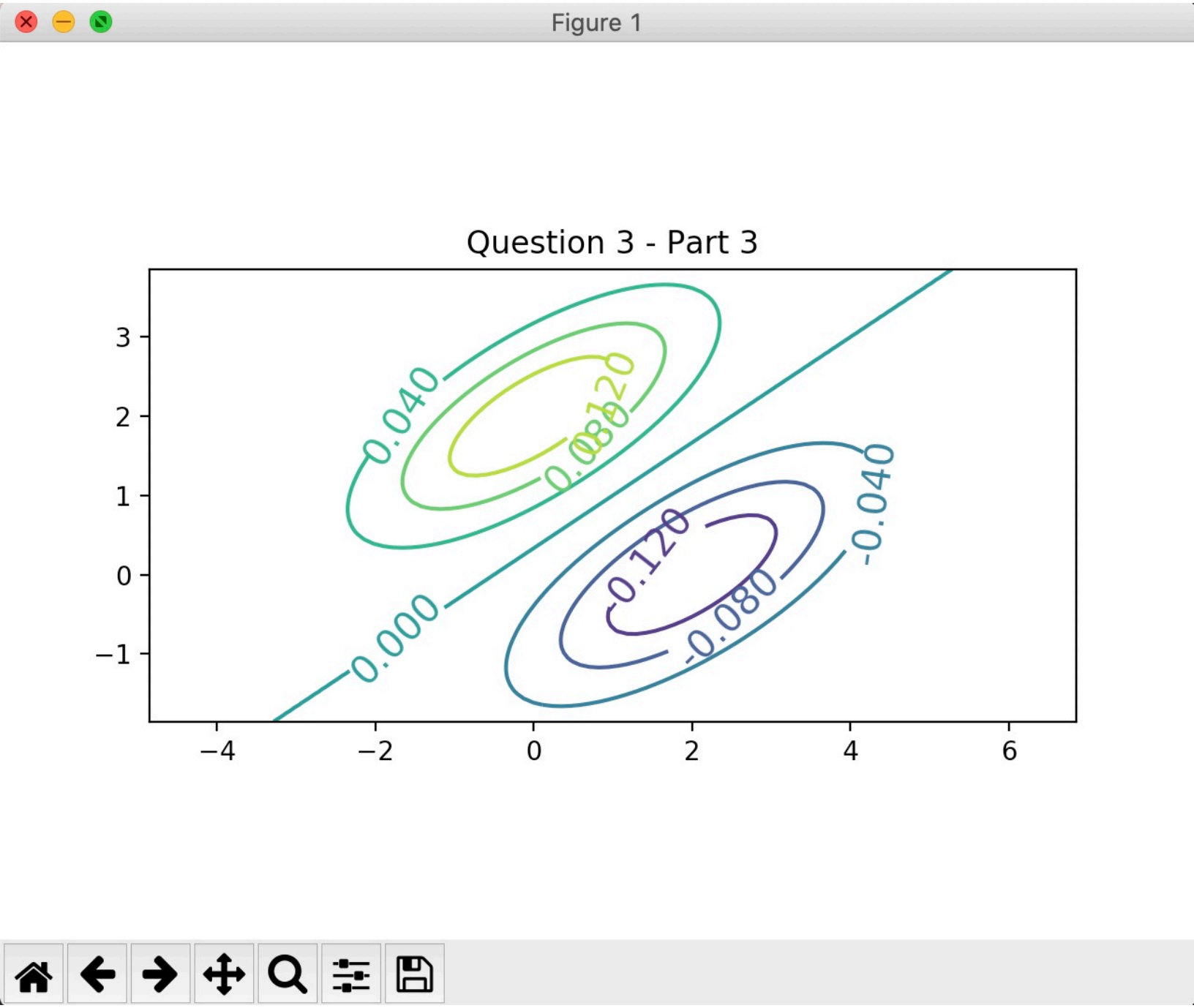
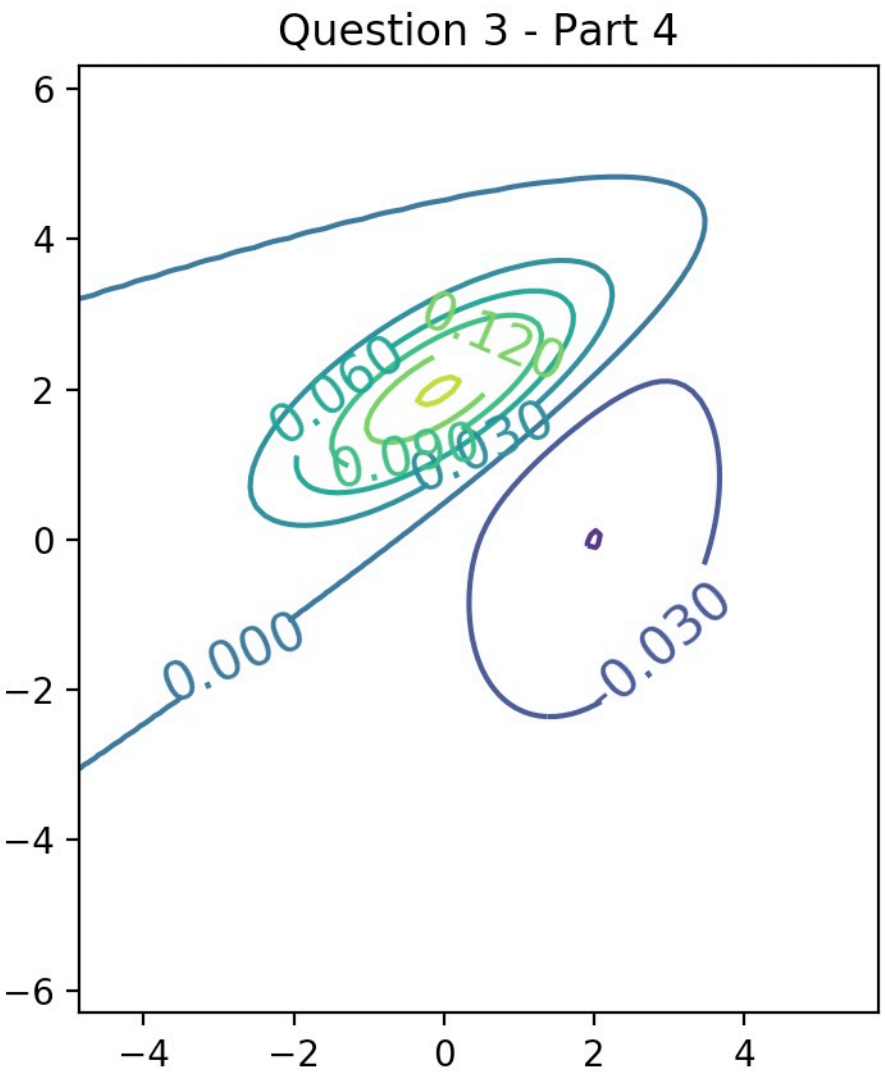


Figure 1
Question 3 - Part 3
0.040
0.080
0.120
0.000
-0.040
-0.080
-0.120
3
2
1
0
−1
−4
−2
0
2
4
6

Figure 1


Question 3 - Part 4
6
4
2
0
−2
−4
−6
−4
−2
0
2
4
0.060
0.120
0.090
0.030
0.000
0.030

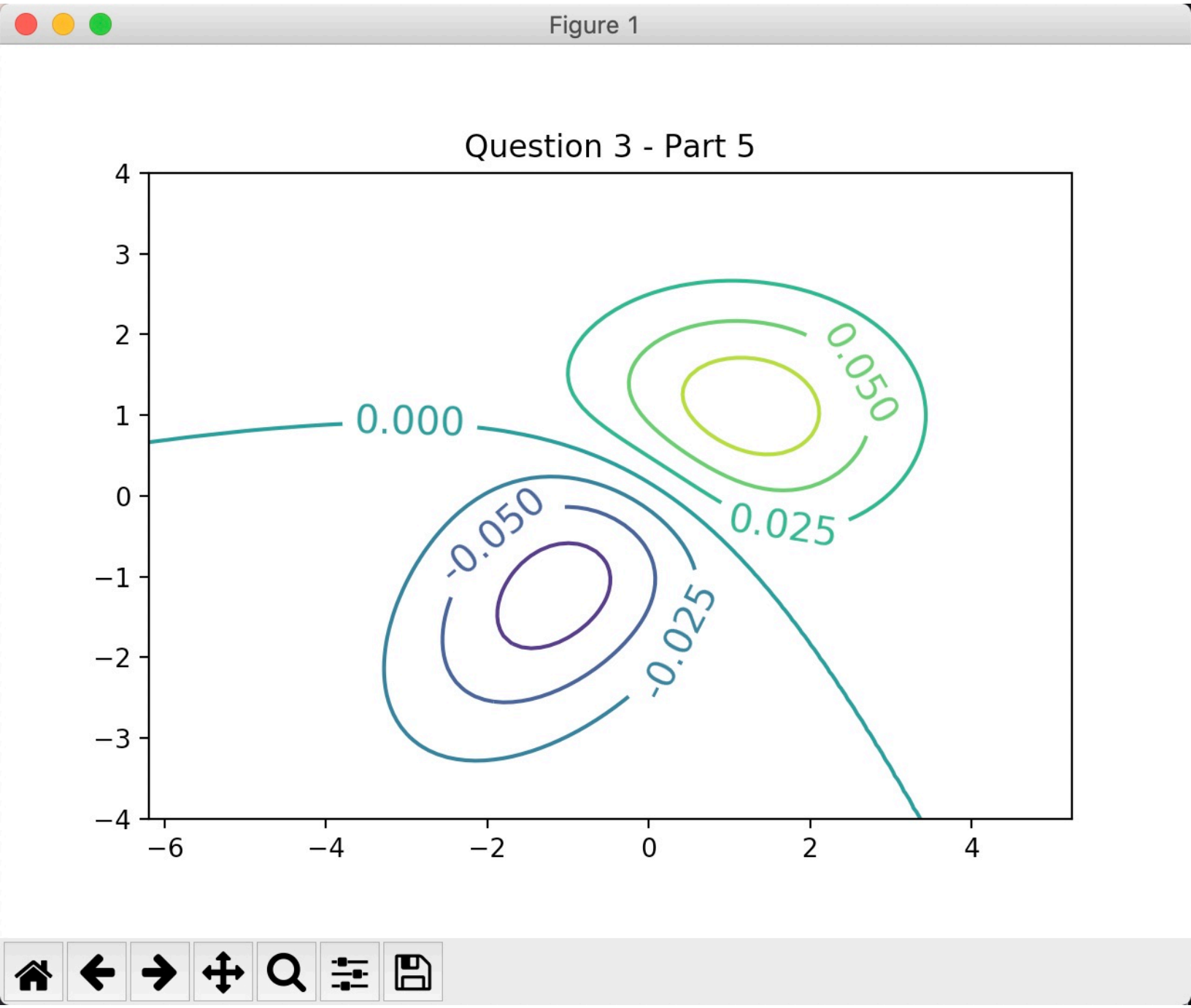


Figure 1
Question 3 - Part 5
0.000
0.050
0.025
-0.050
-0.025
4
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2
4

```
np.random.seed(123456) # initialize seed so all results are reproducible

X1 = np.random.normal(3,3, 100) #100 samples for X1
early_X2 = np.random.normal(4,2,100) #100 samples to be used to calculate X2
X2 = [0.5*X1[i] + early_X2[i] for i in range(len(X1))]
```

```python
# Part a ////////////////////////////////////////////////////////////////////////
mean = np.array([np.mean(X1), np.mean(X2)])
print("Mean: ", mean)
print()

# Part b ////////////////////////////////////////////////////////////////////////
covariance_matrix = np.cov([X1,X2])
print("Covariance Matrix:\n", covariance_matrix)
print()

# Part c ////////////////////////////////////////////////////////////////////////
eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
print("Eigenvalues: ", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
print()

# Part d ////////////////////////////////////////////////////////////////////////
def partD():
    fig, ax = plt.subplots()
    ax.scatter(X1,X2)
    ax.set_aspect(1)
    alpha1 = math.sqrt(eigenvalues[0])/math.sqrt(eigenvectors[0][0]**2 + eigenvectors[1][0]**2)
    alpha2 = math.sqrt(eigenvalues[1])/math.sqrt(eigenvectors[0][1]**2 + eigenvectors[1][1]**2)
    plt.arrow(mean[0], mean[1], alpha1*(eigenvectors[0][0]), alpha1*(eigenvectors[1][0]))
    plt.arrow(mean[0], mean[1], alpha2*(eigenvectors[0][1]), alpha2*(eigenvectors[1][1]))
    plt.xlabel("X1")
    plt.ylabel("X2")
    plt.title("Distribution of 100 Random Sample Points")
    plt.xlim(-15,15)
    plt.ylim(-15,15)
    plt.show()

"Uncomment the following line to run part D"
partD()

# Part e ////////////////////////////////////////////////////////////////////////
def partE():
    uT = eigenvectors.T
    X_rot1 = []
    X_rot2 = []
    for i in range(len(X1)):
        point = np.array([X1[i],X2[i]])
        holder = uT*(point-mean)
        X_rot1.append(holder[0])
        X_rot2.append(holder[1])
    fig, ax = plt.subplots()
    ax.scatter(X_rot1,X_rot2)
    ax.set_aspect(1)
    plt.title("100 Random Sample Points, Centered and Rotated by UT")
    plt.xlim(-15,15)
    plt.ylim(-15,15)
    plt.xlabel("Eigenvector 1")
    plt.ylabel("Eigenvector 2")
    plt.show()

"Uncomment the following line to run part E"
partE()
```

```
Mean:  [2.36040398 5.42818329]

Covariance Matrix:
 [[9.04352807 4.44724276]
 [4.44724276 6.96969976]]

Eigenvalues:  [12.57313985  3.44008798]
Eigenvectors:
 [[ 0.78328427 -0.6216637 ]
 [ 0.6216637   0.78328427]]
```

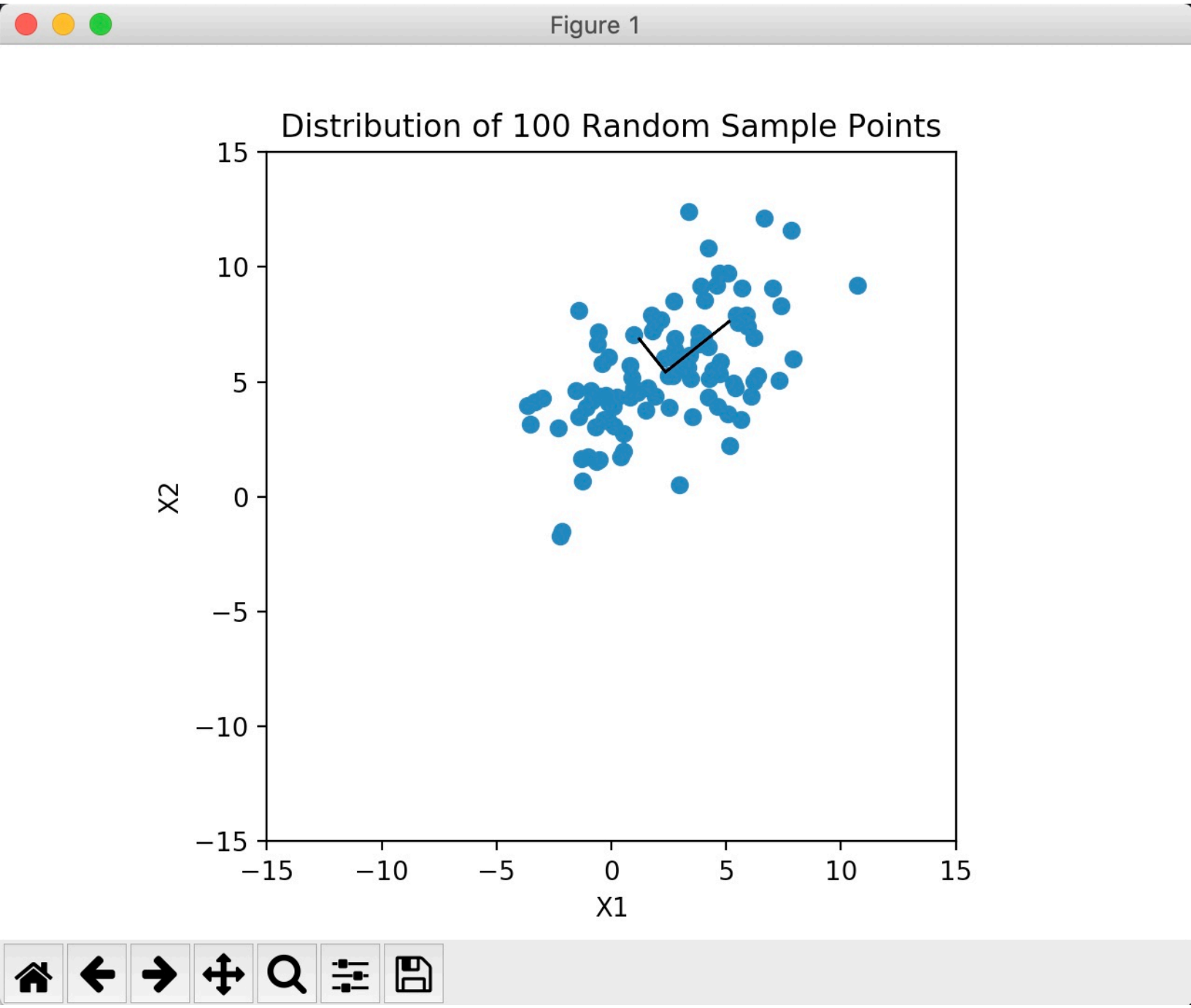


Figure 1
Distribution of 100 Random Sample Points
15
10
5
0
−5
−10
−15
X2
−15
−10
−5
0
5
10
15
X1

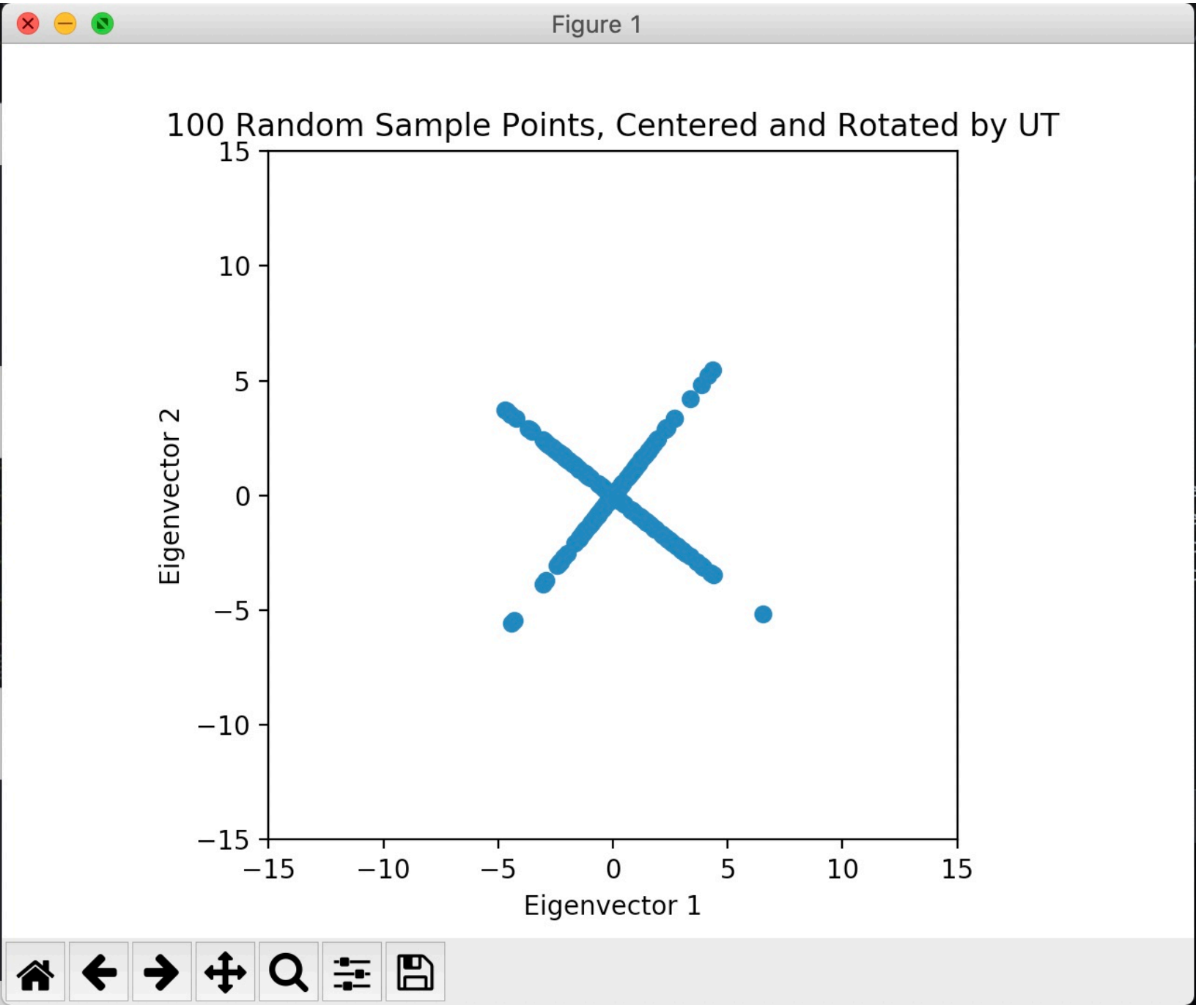


Figure 1
100 Random Sample Points, Centered and Rotated by UT
15
10
5
0
−5
−10
−15
Eigenvector 2
−15
−10
−5
0
5
10
15
Eigenvector 1

⑤ 1) Suppose we choose class i:

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

$$= \lambda_s \mathbb{P}(Y=1|x) + \lambda_s \mathbb{P}(Y=2|x) + \ldots + 0\cdot\mathbb{P}(Y=i|x) + \ldots + \lambda_s \mathbb{P}(Y=c|x)$$

$$= \lambda_s \sum_{i\neq j} \mathbb{P}(Y=j|x) \longrightarrow \left(\sum_{i\neq j}\mathbb{P}(Y=j|x) = 1-\mathbb{P}(Y=i|x)\right)$$

$$= \lambda_s\left(1-\mathbb{P}(Y=i|x)\right) \quad \left(\text{Recall } \mathbb{P}(Y=i|x) \geq 1-\frac{\lambda_r}{\lambda_s}\right)$$

$$\leq \lambda_s\left(1-\left(1-\frac{\lambda_r}{\lambda_s}\right)\right) = \lambda_s\left(\frac{\lambda_r}{\lambda_s}\right) = \lambda_r$$

Thus $\boxed{R(r(x)=i|x) \leq \lambda_r}$

Suppose we choose class $c+1$: $\boxed{R(r(x)=c+1|x) = \lambda_r}$

Therefore this policy has a maximum risk of λ_r

Other policies could go one of two ways, either they favor selecting i more, or $c+1$ more.

→ If they favor i more, then their risk would tend more towards λ_s, which is greater than λ_r

→ If they favor $c+1$ more, they have less of an opportunity to have a risk less than λ_r

Therefore, this is the optimal policy.

---

2) If $\lambda_r = 0$, then $\mathbb{P}(Y=i|x) \geq 1-\frac{0}{\lambda_s}$ and we are guaranteed to either choose i correctly (loss of 0) or choose doubt (also loss of 0)

If $\lambda_r > \lambda_s$, you are guaranteed to always be better of selecting a class $\in \{1,\ldots,c\}$, as the risk will be at most λ_s, which is less than λ_r.
This is consistent with our intuition since guessing incorrectly (λ_s) is better than doubt (λ_r)

⑥ a) $\mathcal{L}(\mu, \Sigma; X_1, \ldots, X_n) = f(X_1) f(X_2) \cdots f(X_n)$

*As stated in lecture 8, independent features allow us to

- The log-likelihood (ℓ) is the natural log of the likelihood
  → maximizing $\ell \iff$ maximizing $\mathcal{L}$

$$\ell(\mu, \Sigma; X_1, \ldots, X_n) = \ln\left(\mathcal{L}(\mu, \Sigma; X_1, \ldots, X_n)\right) = \ln\left(f(X_1) f(X_2) \cdots f(X_n)\right)$$

$$= \ln\left(f(X_1)\right) + \ln\left(f(X_2)\right) + \ldots + \ln\left(f(X_n)\right) = \sum_{i=1}^{n} \ln\left(f(X_i)\right)$$

$$f(X_i) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(\frac{-1}{2}(X_i - \mu)^T \Sigma^{-1} (X_i - \mu)\right)$$

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

- To find the most likely μ, set $\nabla_\mu \ell = 0$

$$\nabla_\mu \ell = \sum_{i=1}^{n} \frac{-1}{2} \cdot 2 \cdot \Sigma^{-1}(X_i - \mu) = \sum_{i=1}^{n} \Sigma^{-1}(\mu - X_i) = 0 \quad (\Sigma \text{ is PD, thus we can invert})$$

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

$$\sum_{i=1}^{n} (\mu - X_i) = 0 \rightarrow n\mu = \sum_{i=1}^{n} X_i \rightarrow \boxed{\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} X_i}$$

- To find the most likely σ_i, set $\frac{\partial \ell}{\partial \sigma_i} = 0$

(continues onto next page)

$$\ell = \sum_{j=1}^{n} \left(\frac{-d}{2} \ln(2\pi) - \frac{1}{2} \ln\left(\prod_{k=1}^{d} \sigma_k^2 \right) - \frac{1}{2} (x_j - \mu)^T \Sigma^{-1} (x_j - \mu) \right)$$

$$\frac{\partial \ell}{\partial \sigma_i} = \sum_{j=1}^{n} \frac{\partial}{\partial \sigma_i} \left(-\frac{1}{2} \cdot 2 \sum_{k=1}^{d} \ln \sigma_k - \frac{1}{2} \sum_{k=1}^{d} (x_j - \mu)_k^2 \frac{1}{\sigma_k^2} \right)$$

$$= \sum_{j=1}^{n} \left(\frac{-1}{\sigma_i} + \frac{1}{2} \cdot (-2) (x_j - \mu)_i^2 \frac{1}{\sigma_i^3} \right)$$

$$= \frac{-n}{\sigma_i} + \frac{1}{\sigma_i^3} \sum_{j=1}^{n} (x_j - \mu)_i^2 = 0$$

$$\frac{n}{\sigma_i} = \frac{1}{\sigma_i^3} \sum_{j=1}^{n} (x_j - \mu)_i^2$$

$$\sigma_i^2 = \frac{1}{n} \sum (x_j - \mu)_i^2$$

$$\boxed{\hat{\sigma}_i = \sqrt{\frac{1}{n} \sum_{j=1}^{n} (x_j - \hat{\mu})_i^2}}$$

$(x_j - \hat{\mu})_i$ is the i^{th} element of $(x_j - \hat{\mu})$

b) As stated in a, we can represent the likelihood as:

$$\mathcal{L}(A\mu, \Sigma; X_1, \ldots, X_n) = f(X_1)f(X_2)\ldots f(X_n)$$

And since maximizing the log likelihood is equivalent to maximizing the likelihood:

$$\ell(A\mu, \Sigma; X_1, \ldots, X_n) = \ln\left(\prod_{i=1}^{n} f(x_i)\right) = \sum_{i=1}^{n} \ln(f(x_i))$$

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

When calculating $\nabla_\mu \ell$, only this portion "matters"

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

$$= \nabla_\mu\left(\sum_{i=1}^{n} -\frac{1}{2}\left(\cancel{x_i^T\Sigma^{-1}x_i}^{\text{no } \mu \text{ term}} - 2\mu^T A^T\Sigma^{-1}x_i + \mu^T A^T\Sigma^{-1}A\mu\right)\right)$$

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

$$= \sum_{i=1}^{n} A^T\Sigma^{-1}x_i - \frac{n}{2}\cdot 2(A^T\Sigma^{-1}A)\mu = 0$$

$$n(A^T\Sigma^{-1}A)\mu = A^T\Sigma^{-1}\sum_{i=1}^{n} x_i$$

$$\cancel{A^T}\cancel{A^{-T}}\, n\Sigma^{-1}A\mu = \cancel{A^T}\cancel{A^{-T}}\,\Sigma^{-1}\sum_{i=1}^{n} x_i$$

$$\cancel{\Sigma}\cancel{\Sigma^{-1}}\, nA\mu = \cancel{\Sigma}\cancel{\Sigma^{-1}}\sum_{i=1}^{n} x_i$$

$$A\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{(as expected)}$$

$$\cancel{A^{-1}}\cancel{A}\,\mu = A^{-1}\frac{1}{n}\sum_{i=1}^{n} x_i$$

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

(7) a) The covariance matrix is singular when:
- → the variance of any R_i is zero
- → there are less than d linearly independent sample points

---

b) When $\hat{\Sigma}$ is uninvertible, we need to ensure that all of our sample points are linearly independent. To do this, we can add a bit of Gaussian noise (scaled by a small number that is perhaps a magnitude or two smaller than the parameters of the distribution). This keeps our data (and covariance matrix) relatively unchanged, but solves the issue of singularity.

---

c) The eigenvectors of Σ maximize the pdf of $f(x)$. This is because these vectors are in the direction of maximum spread. That's what the covariance matrix is all about, baby!

```python
# Part 1 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
def getGaussDistr():
    """Returns the Mean vectors and Covariance matrices for the mnist data set.
    Means[i] is the mean vector for the class of digit i.
    Covs[i] is the Covariance vector for the class of digit i."""
    mnist_data = scipy.io.loadmat("../data/mnist_data.mat")
    training_data, training_labels = shuffle(mnist_data["training_data"], mnist_data["training_labels"])
    Means = calculateMeans(training_data, training_labels)[0]
    Covs = calculateCovs(training_data, training_labels)
    return Means, Covs

def calculateMeans(training_data, training_labels):
    """Returns Means, priors.
    Means[i] is the mean vector for the class of digit i
    priors[i] is the prior for the class of digit i"""
    sum_matrix = [np.array([]) for _ in range(10)] #this will hold an array of sums of the data (to be used to calculate means)
    counts = [0 for _ in range(10)] #this will be used to hold counts of the data (also for means)
    for i in range(len(training_labels)):
        index = training_labels[i][0] #this number tells us which class this data point belongs to
        img = normalize(training_data[i]) #we set img to be the normalized data point
        if len(sum_matrix[index]) == 0: #first time encountering this class
            sum_matrix[index] = img
        else:
            sum_matrix[index] += img
        counts[index] += 1 #increment the number of times we've encountered a data point belonging to this class
    Means = [sum_matrix[i]/counts[i] for i in range(10)] #Means[i] = mean of class i
    return Means, np.array(counts)*sum(counts)

def calculateCovs(training_data, training_labels):
    """Returns Covs, where Covs[i] is the Covariance vector for the class of digit i"""
    Cov_data = [[] for _ in range(10)]
    for i in range(len(training_labels)):
        index = training_labels[i][0] #which class the data point belongs to
        img = normalize(training_data[i]) #normalize the data point
        img = addGaussianNoise(img) #adds guassian noise so the covariance matrix isn't singular
        Cov_data[index].append(img) #Cov_data[index] holds data to calculate the covariance of the index class
    Covs = []
    for i in range(len(Cov_data)):
        c = np.array(Cov_data[i])
        Covs.append(np.cov(c.T))
    return Covs
```

```python
# Part 2 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

def visualizeCovariance(digit, Covs):
    """Plots a visualization of the covariance matrix for digit (Cov[digit])"""
    cov = Covs[digit]
    plt.imshow(cov)
    plt.title("Visualization of the Covariance Matrix for the Digit " + str(digit))
    plt.show()
```

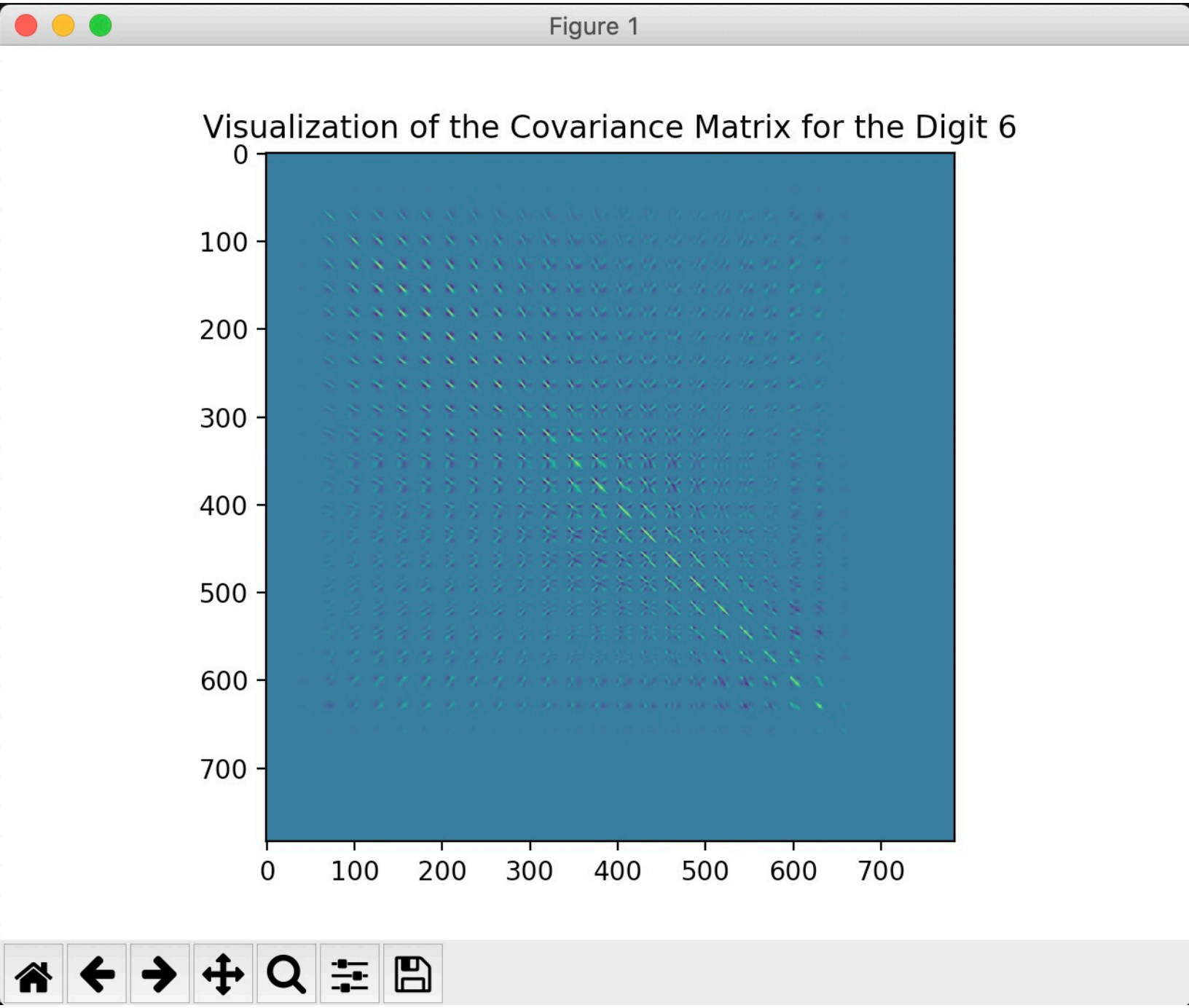


Figure 1
Visualization of the Covariance Matrix for the Digit 6
0
100
200
300
400
500
600
700
0
100
200
300
400
500
600
700

(8) 2) In general, the diagonal terms have a larger value than those off of the diagonal (especially the terms at the edge of the matrix). This implies that features are not heavily correlated, especially features on the outside of the image. This trend is consistent for all images.

---

3) c. In my case, QDA performed better. While QDA may often be in danger of overfitting, there are a lot of data points, thus allowing the model to be more accurate than LDA. The variances are also different enough that averaging them harms the model.

```python
# Part 3 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
# Part a ------------------------------------------------------------

def LDA():
    """Classify the test data points using LDA"""
    training_data, training_labels, validation_data, validation_labels = train_val_split("../data/mnist_data.mat", 10000)
    sizes = [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000]
    error_rates = []
    for size in sizes:
        Means, Cov, priors = trainLDA(training_data, training_labels, size)
        error_rates.append(getLDAErrorRate(Means, Cov, priors, validation_data, validation_labels))
    plotErrorRates(sizes, error_rates, "LDA Error Rates")

def trainLDA(training_data, training_labels, size):
    """Trains our LDA function using 'size' training points, returns Means, Covariance Matrix, and Priors"""
    t_data, t_labels = shuffle(training_data, training_labels)
    data = t_data[:size]
    labels = t_labels[:size]
    Means, priors = calculateMeans(data, labels)
    Covs = calculateCovs(data, labels) #holds individual covariances
    cov_sum = np.zeros(shape=(784,784)) #will hold the sum of all covariances
    for c in Covs:
        cov_sum += c
    Cov = .1*cov_sum
    return Means, Cov, priors

def getLDAErrorRate(Means, Cov, priors, validation_data, validation_labels):
    """Returns the error rate of our LDA model (mu[i] = Means[i], Sigma = Cov)"""
    z = [] #list of values representing Sigma^(-1)*Mu[class]
    for i in range(10):
        z.append(np.linalg.solve(Cov, Means[i]))
    predictions = getLDAPredictions(Means, Cov, priors, validation_data, validation_labels, z)
    num_correct = 0 #number of correct predicitions
    for i in range(len(validation_data)):
        if predictions[i] == validation_labels[i]:
            num_correct += 1
    return 1 - num_correct/len(validation_data)

def getLDAPredictions(Means, Cov, priors, validation_data, validation_labels, z):
    """Returns a list of predictions of our LDA model"""
    predictions = []
    for data_point in validation_data:
        max_class = 0
        max_val = -1*float('inf')
        for digit in range(10):
            val = np.dot(z[digit], data_point) - 0.5* np.dot(z[digit], Means[digit]) + np.log(priors[digit])
            if val > max_val:
                max_val = val
                max_class = digit
        predictions.append(max_class)
    return predictions
```

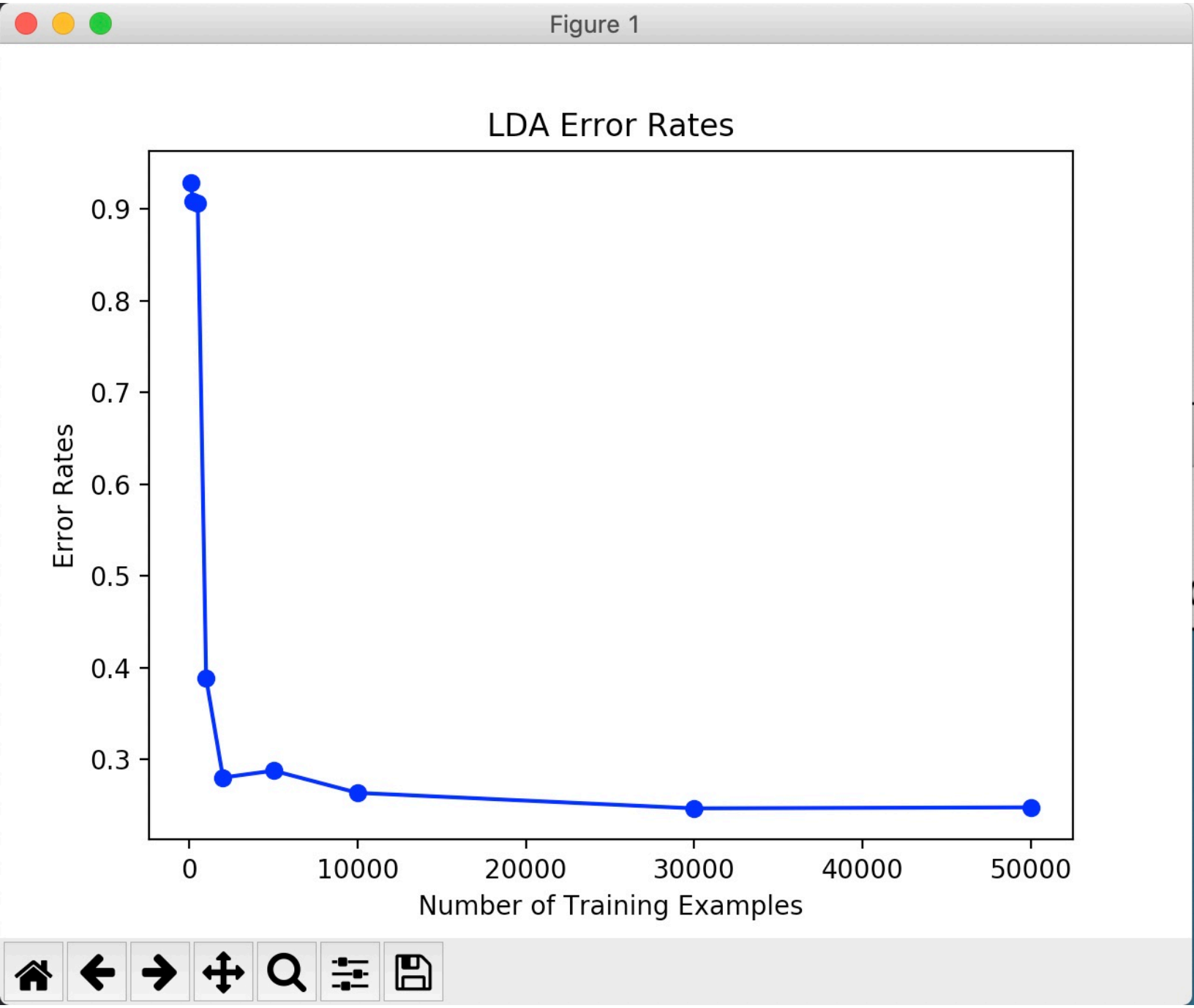


Figure 1
LDA Error Rates
Error Rates
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
10000
20000
30000
40000
50000
Number of Training Examples

```python
# Part b ------------------------------------------------------------------
def QDA():
    """Classify the test data points using QDA"""
    training_data, training_labels, validation_data, validation_labels = train_val_split("../data/mnist_data.mat", 10000)
    sizes = [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000]
    error_rates = []
    for size in sizes:
        Means, Covs, priors = trainQDA(training_data, training_labels, size)
        error_rates.append(getQDAErrorRate(Means, Covs, priors, validation_data, validation_labels))
    plotErrorRates(sizes, error_rates, "QDA Error Rates")

def trainQDA(training_data, training_labels, size):
    """Trains our QDA function using 'size' training points, returns Means, Covariance Matrices, and Priors"""
    t_data, t_labels = shuffle(training_data, training_labels)
    data = t_data[:size]
    labels = t_labels[:size]
    Means, priors = calculateMeans(data, labels)
    Covs = calculateCovs(data, labels)
    return Means, Covs, priors

def getQDAErrorRate(Means, Covs, priors, validation_data, validation_labels):
    """Returns the error rate of our QDA model (mu[i] = Means[i], Sigma = Cov)"""
    predictions = getQDAPredictions(Means, Covs, priors, validation_data)
    num_correct = 0 #number of correct predicitions
    for i in range(len(validation_data)):
        if predictions[i] == validation_labels[i]:
            num_correct += 1
    return 1 - num_correct/len(validation_data)

def getQDAPredictions(Means, Covs, priors, validation_data):
    """Returns a list of predictions of our QDA model"""
    predictions = []
    determinants = [np.linalg.slogdet(Covs[i]) for i in range(10)]
    inverses = [np.linalg.inv(Covs[i]) for i in range(10)]
    for i in range(len(validation_data)):
        data_point = validation_data[i]
        max_class = 0
        max_val = -1*float('inf')
        for digit in range(10):
            z_c = np.dot(inverses[digit], data_point-Means[digit])
            val = -0.5*np.dot(z_c, data_point-Means[digit]) - 0.5*determinants[digit][1] + np.log(priors[digit])
            if val > max_val:
                max_val = val
                max_class = digit
        predictions.append(max_class)
    return predictions
```

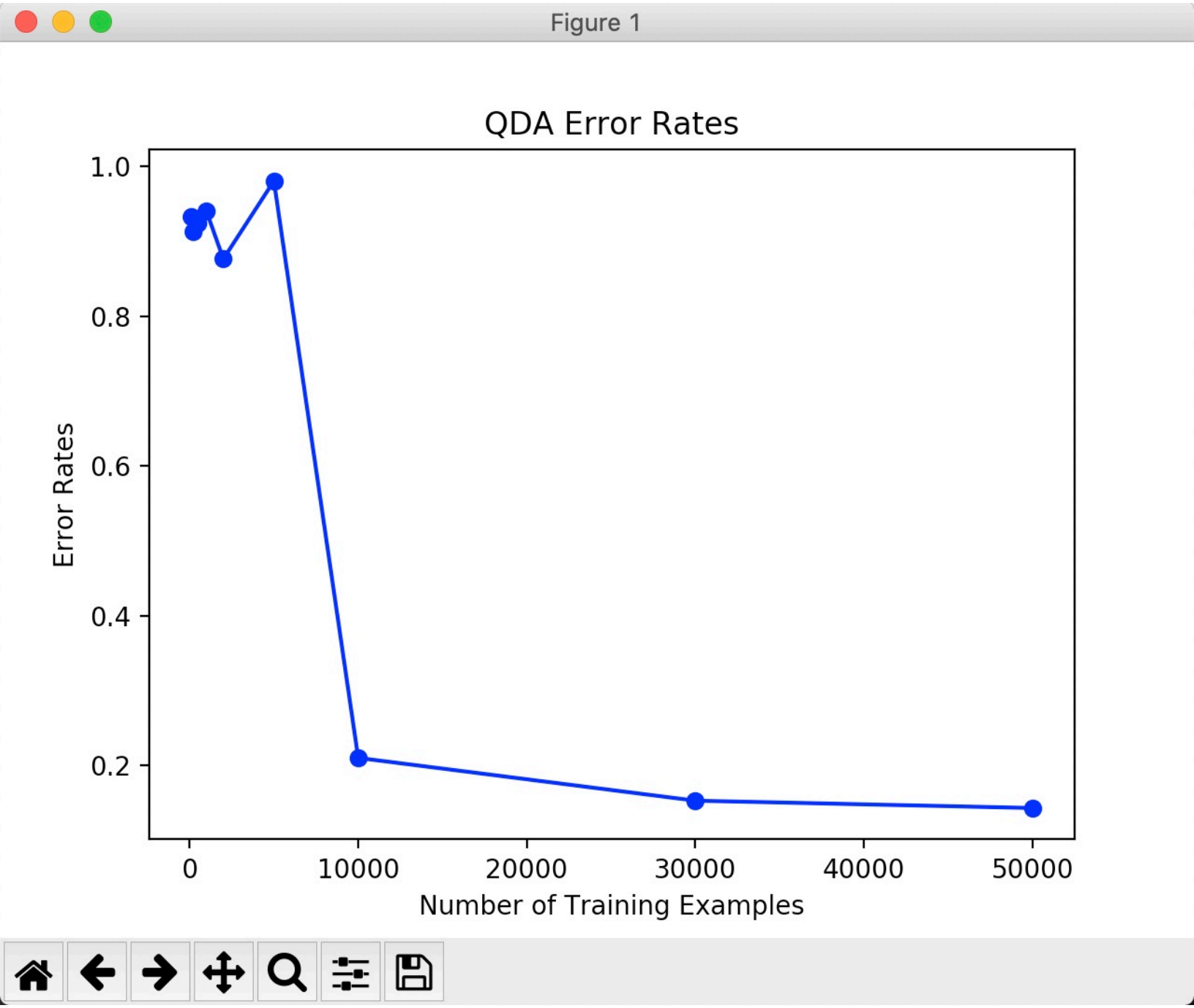


Figure 1
QDA Error Rates
1.0
0.8
0.6
0.4
0.2
Error Rates
0
10000
20000
30000
40000
50000
Number of Training Examples

```python
# Part d ------------------------------------------------------------
def testQDA():
    """trains our QDA model on 30000 points of training data and returns our predictions for the test data"""
    mnist_data = scipy.io.loadmat("../data/mnist_data.mat")
    training_data, training_labels = shuffle(mnist_data["training_data"], mnist_data["training_labels"])
    training_data = training_data[:30000]
    training_labels = training_labels[:30000]
    test_data = mnist_data["test_data"]
    Means, priors = calculateMeans(training_data, training_labels)
    Covs = calculateCovs(training_data, training_labels)
    predictions = getQDAPredictions(Means, Covs, priors, test_data)
    return predictions

def QDAIndividualErrors():
    training_data, training_labels, validation_data, validation_labels = train_val_split("../data/mnist_data.mat", 10000)
    sizes = [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000]
    error_rates = []
    for size in sizes:
        Means, Covs, priors = trainQDA(training_data, training_labels, size)
        error_rates.append(getIndividualErrorRates(Means, Covs, priors, validation_data, validation_labels))
        print(error_rates)
    error_rates = np.array(error_rates)
    plotIndividualErrorRates(error_rates.T)

def getIndividualErrorRates(Means, Covs, priors, validation_data, validation_labels):
    predictions = getQDAPredictions(Means, Covs, priors, validation_data)
    num_error = [0 for _ in range(10)] #value at each index represents number of incorrect predicitions for that class
    num_total = [0 for _ in range(10)]
    for i in range(len(validation_data)):
        digit = validation_labels[i][0]
        if not predictions[i] == digit:
            num_error[digit] += 1
        num_total[digit] += 1
    error_rates = [num_error[digit]/num_total[digit] for digit in range(10)]
    return error_rates

def plotIndividualErrorRates(error_rates):
    print("wow")
    x_axis = [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000]
    for i in range(len(error_rates)):
        plt.plot(x_axis, error_rates[i])
    plt.legend([i for i in range(10)], loc=1)
    plt.title("Error Rates for Individual Digits")
    plt.xlabel("Number of Training Samples")
    plt.ylabel("Error Rate")
    plt.show()
```

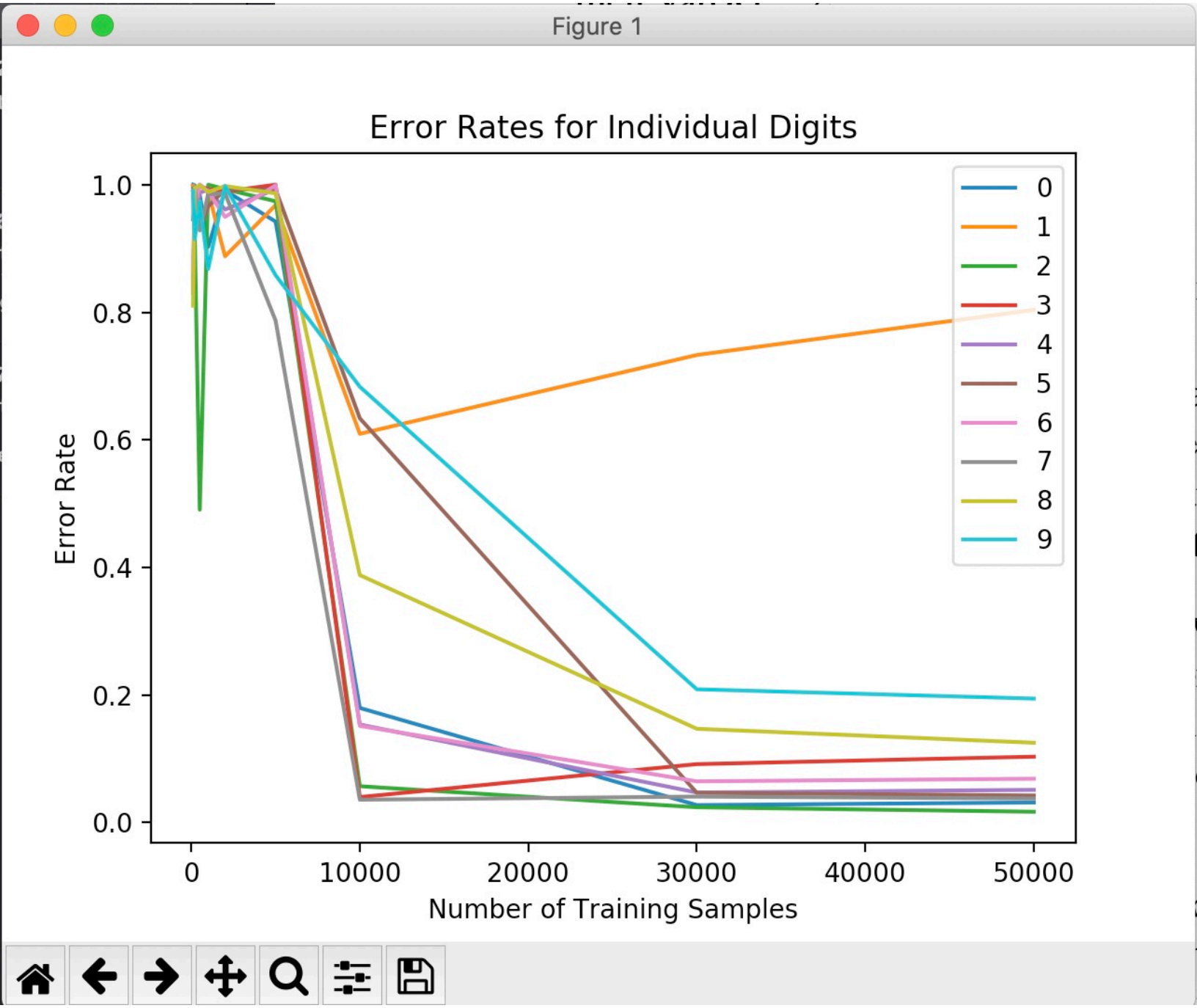


Figure 1
Error Rates for Individual Digits
Error Rate
1.0
0.8
0.6
0.4
0.2
0.0
0
10000
20000
30000
40000
50000
Number of Training Samples
0
1
2
3
4
5
6
7
8
9

```python
# Part 4 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
def spamQDA():
    """Runs QDA on spam data, returning the predictions"""
    spam_data = scipy.io.loadmat("../data/spam_data.mat")
    training_data, training_labels = shuffle(spam_data["training_data"], spam_data["training_labels"])
    test_data = spam_data["test_data"]
    Means, Covs, priors = trainSpamQDA(training_data, training_labels)
    predictions = getSpamQDAPredictions(Means, Covs, priors, test_data)
    return predictions

def trainSpamQDA(training_data, training_labels):
    Means, priors = calculateSpamMeans(training_data, training_labels)
    Covs = calculateSpamCovs(training_data, training_labels)
    return Means, Covs, priors

def calculateSpamMeans(training_data, training_labels):
    """Returns Means, priors.
    Means[0] is the mean vector for the spam class, Means[1] for ham
    priors[0] is the prior for the spam class, priors[1] for ham"""
    sum_matrix = [np.array([]), np.array([])] #this will hold an array of sums of the data (to be used to calculate means)
    counts = [0, 0] #this will be used to hold counts of the data (also for means)
    for i in range(len(training_labels)):
        index = training_labels[i][0] #this number tells us which class this data point belongs to
        img = normalize(training_data[i]) #we set img to be the normalized data point
        if len(sum_matrix[index]) == 0: #first time encountering this class
            sum_matrix[index] = img
        else:
            sum_matrix[index] += img
        counts[index] += 1 #increment the number of times we've encountered a data point belonging to this class
    Means = [sum_matrix[0]/counts[0], sum_matrix[1]/counts[1]] #Means[i] = mean of class i
    return Means, np.array(counts)*sum(counts)

def calculateSpamCovs(training_data, training_labels):
    """Returns Covs, where Covs[0] is the Covariance vector for the spam class and Covs[1] is for the ham class"""
    Cov_data = [[], []]
    for i in range(len(training_labels)):
        index = training_labels[i][0] #which class the data point belongs to
        img = normalize(training_data[i]) #normalize the data point
        img = addGaussianNoise(img) #adds guassian noise so the covariance matrix isn't singular
        Cov_data[index].append(img) #Cov_data[index] holds data to calculate the covariance of the index class
    Covs = []
    for i in range(len(Cov_data)):
        c = np.array(Cov_data[i])
        Covs.append(np.cov(c.T))
    return Covs

def getSpamQDAPredictions(Means, Covs, priors, validation_data):
    predictions = []
    determinants = [np.linalg.slogdet(Covs[0]), np.linalg.slogdet(Covs[1])]
    inverses = [np.linalg.inv(Covs[0]), np.linalg.inv(Covs[1])]
    for i in range(len(validation_data)):
        data_point = validation_data[i]
        max_class = 0
        max_val = -1*float('inf')
        for j in range(2):
            z_c = np.dot(inverses[j], data_point-Means[j])
            val = -0.5*np.dot(z_c, data_point-Means[j]) - 0.5*determinants[j][1] + np.log(priors[j])
            if val > max_val:
                max_val = val
                max_class = j
        predictions.append(max_class)
    return predictions
```

```
# Helper Functions ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

def train_val_split(directory, split):
    """returns training_data, training_labels, validation_data, validation_labels, with the validation being of size split"""
    mat = scipy.io.loadmat(directory)
    data, labels = shuffle(mat["training_data"], mat["training_labels"])
    training_data = data[split:]
    training_labels = labels[split:]
    validation_data = data[:split]
    validation_labels = labels[:split]
    return training_data, training_labels, validation_data, validation_labels

def normalize(arr):
    """returns a normalized array"""
    norm = np.linalg.norm(arr)
    if not norm == 0:
        arr = (1/norm)*arr
    return arr

def plotErrorRates(x,y,title):
    """Plots x vs y with title"""
    plt.plot(x,y,'bo-')
    plt.ylabel("Error Rates")
    plt.xlabel("Number of Training Examples")
    plt.title(title)
    plt.show()

np.random.seed(123456)

def addGaussianNoise(m):
    noise = .01*np.random.normal(0,1,len(m))
    return m + noise

# Running the Code ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

"To run the code for the ith part of question 8, make sure that the line(s) of code beneath the comment 'Part i' are uncommented"

# Part 1
#means, covs = getGaussDistr()

# Part 2
#means, covs = getGaussDistr()
#visualizeCovariance(6, Covs)

# Part 3a
#LDA()

# Part 3b
#QDA()

# Part 3d
#predictions = testQDA()
#results_to_csv(np.array(predictions))
#QDAIndividualErrors()

# Part 4
#spam_predictions = spamQDA()
#results_to_csv(np.array(spam_predictions))
```