1 Independence vs Correlation

(a) **Answer: X and Y are uncorrelated, but they are not independent.**
Prove uncorrelated:
Two random variables X, Y are uncorrelated if their covariance $E(XY) - E(X)E(Y)$ is 0.
$E(X) = 0 \cdot P(X = 0) + 1 \cdot P(X = 1) + (-1) \cdot P(X = -1)$
$= P(X = 1) - P(X = -1)$
$= P(X = 1, Y = 0) + P(X = 1, Y = 1) - P(X = -1, Y = 0) - P(X = -1, Y = 1)$
$= P(X = 1, Y = 0) - P(X = -1, Y = 0)$
$= P(X = 1|Y = 0) \cdot P(Y = 0) - P(X = -1|Y = 0) \cdot P(Y = 0)$
$= \frac{1}{2} \cdot \frac{1}{2} - \frac{1}{2} \cdot \frac{1}{2} = 0$
Similarly, $E(Y) = 0$.
Because either X or Y is zero, X and Y are never nonzero at the same time, so $E(XY) = 0$.
Therefore, we have proved that $E(XY) = E(X)E(Y) = 0$

Prove dependent:
Two random variables X, Y are independent if $P(X|Y) = P(X)$.
$P(X = 1|Y = 0) = \frac{1}{2}$
$P(X = 1) = P(X = 1, Y = 0) = P(X = 1|Y = 0) \cdot P(Y = 0) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$
Therefore, X, Y are not independent.

(b) **Answer: X, Y, Z are pairwise independent but not mutually independent.**
Prove pairwise independent:
For $P(X|Y)$ we are given Y which put some constraints on C and D, between which only C influences X. However, B randomly chooses between 0 and 1 and it is independent of C, so knowing Y won't influence P(X). Thus $P(X|Y) = P(X)$.
Similar applies to other pairs.

Prove not mutually independent:
$P(X|Y, Z) \neq P(X)$ because if given Y = 1 and Z = 1, we can deduce that B = C = D, so that X is definitely = 1 too. Thus Y, Z influence X.

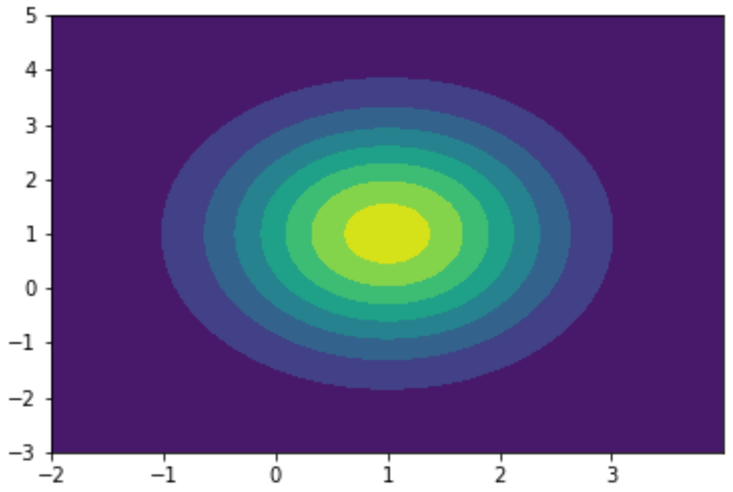
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
from scipy.stats import multivariate_normal
import math
%matplotlib inline
```

2. Isocontours of Normal Distributions

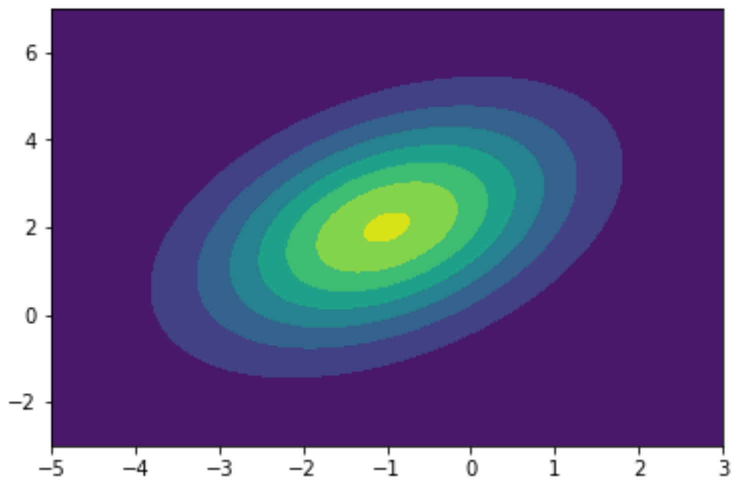
Reference: https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.multivariate_normal.html (https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.multivariate_normal.html)

In [2]:
```
def isocontours(mu, sig, x, y):
    pos = np.empty(x.shape + (2,))
    pos[:, :, 0] = x
    pos[:, :, 1] = y
    rv = multivariate_normal(mu, sig)
    fig = plt.figure()
    plot = fig.add_subplot(1.2,1.7,1)
    return plot.contourf(x, y, rv.pdf(pos))

def two_isocontours(mu1, mu2, sig1, sig2, x, y):
    pos = np.empty(x.shape + (2,))
    pos[:, :, 0] = x
    pos[:, :, 1] = y
    rv1 = multivariate_normal(mu1, sig1)
    rv2 = multivariate_normal(mu2, sig2)
    fig = plt.figure()
    plot = fig.add_subplot(1.2,1.7,1)
    return plot.contourf(x, y, rv1.pdf(pos) - rv2.pdf(pos))
```

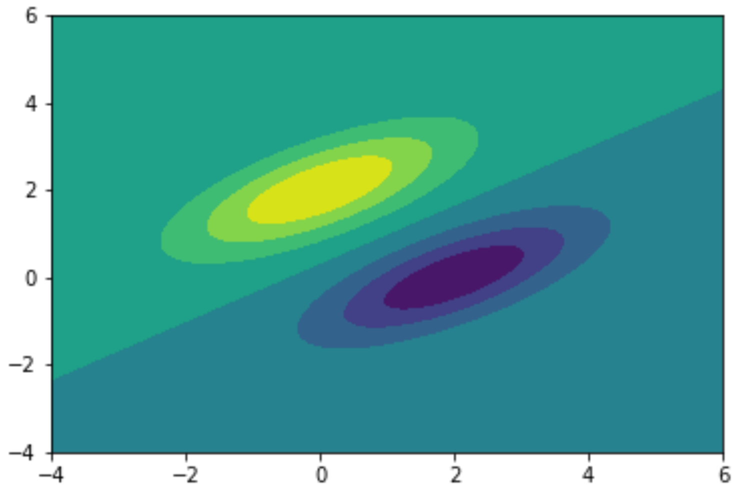
```
In [3]: # (a)
        mu = [1, 1]
        sig = [[1,0],[0,2]]
        x, y = np.mgrid[-2:4:.01, -3:5:0.01]
        a = isocontours(mu, sig, x, y)
        a

Out[3]: <matplotlib.contour.QuadContourSet at 0x107e885f8>
```



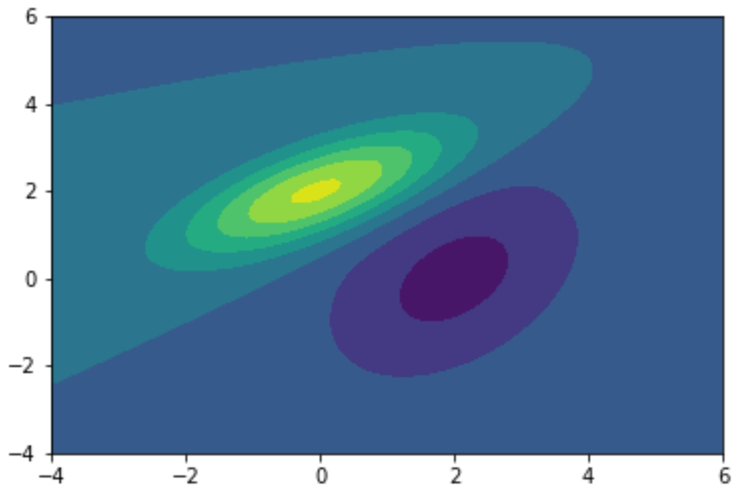
```
In [4]: # (b)
        mu = [-1, 2]
        sig = [[2,1],[1,3]]
        x, y = np.mgrid[-5:3:.01, -3:7:.01]
        b = isocontours(mu, sig, x, y)
        b

Out[4]: <matplotlib.contour.QuadContourSet at 0x10cae7e10>
```



```
In [5]: # (c)
        mu1 = [0, 2]
        mu2 = [2, 0]
        sig = [[2,1],[1,1]]
        x, y = np.mgrid[-4:6:.01, -4:6:.01]
        c = two_isocontours(mu1, mu2, sig, sig, x, y)
        c

Out[5]: <matplotlib.contour.QuadContourSet at 0x1112f0438>
```

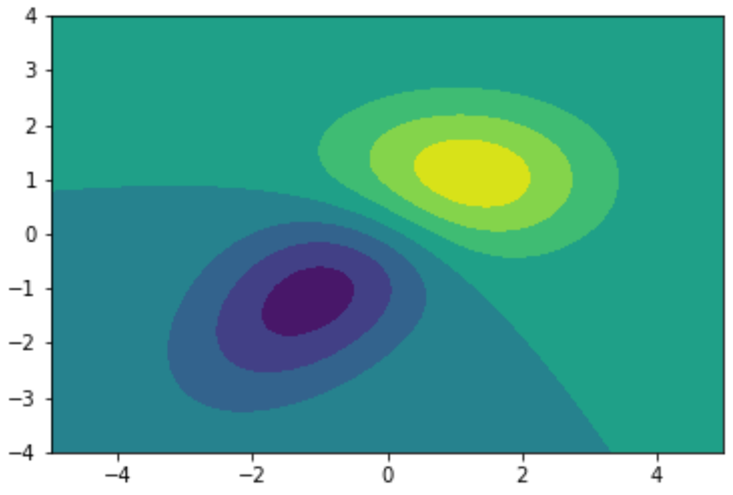


```
In [6]: # (d)
        mu1 = [0, 2]
        mu2 = [2, 0]
        sig1 = [[2,1],[1,1]]
        sig2 = [[2, 1], [1, 3]]
        x, y = np.mgrid[-4:6:.01, -4:6:.01]
        d = two_isocontours(mu1, mu2, sig1, sig2, x, y)
        d

Out[6]: <matplotlib.contour.QuadContourSet at 0x1143b0588>
```



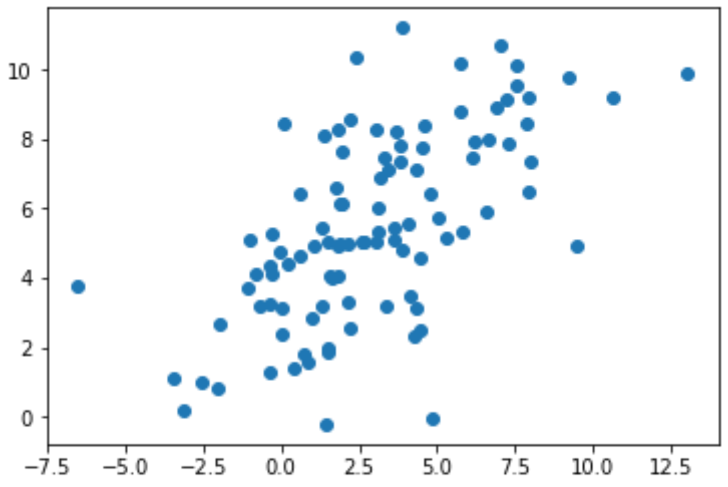
In [7]:
```
# (3)
mu1 = [1, 1]
mu2 = [-1, -1]
sig1 = [[2, 0],[0, 1]]
sig2 = [[2, 1], [1, 2]]
x, y = np.mgrid[-5:5:.01, -4:4:.01]
e = two_isocontours(mu1, mu2, sig1, sig2, x, y)
e
```

Out[7]:
```
<matplotlib.contour.QuadContourSet at 0x11611ac50>
```



3. Eigenvectors of the Gaussian Covariance Matrix

In [18]:
```
# randomly draw 100 sample points
n = 100
x1 = np.random.normal(3, 3, n)
x2 = 0.5 * x1 + np.random.normal(4, 2, n)
fig, axes = plt.subplots()
axes.scatter(x1, x2)
plt.show()
```



In [19]:
```
# (a)
x1_mean = np.mean(x1)
x2_mean = np.mean(x2)
print("x1 mean = ", x1_mean)
print("x2 mean = ", x2_mean)
```

```
x1 mean =  2.95932073974
x2 mean =  5.49994510123
```

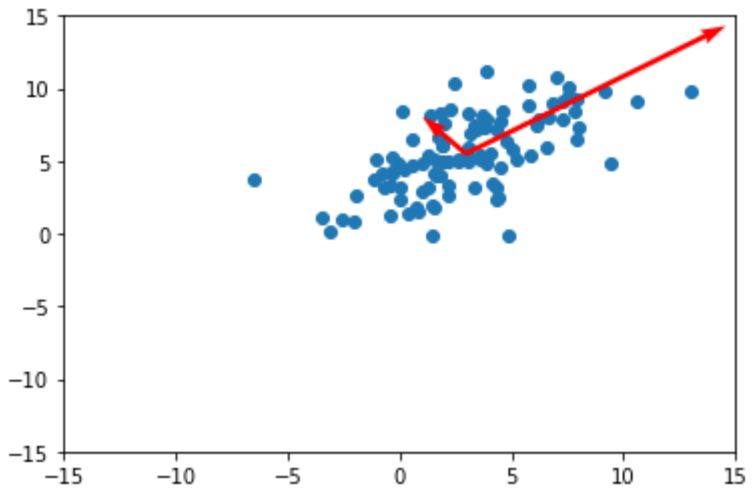
In [20]:
```
# (b)
m = np.cov(x1, x2)
print("Covariance Matrix = \n", m)
```

```
Covariance Matrix =
 [[ 10.43613528   5.47600768]
 [  5.47600768   7.34461899]]
```

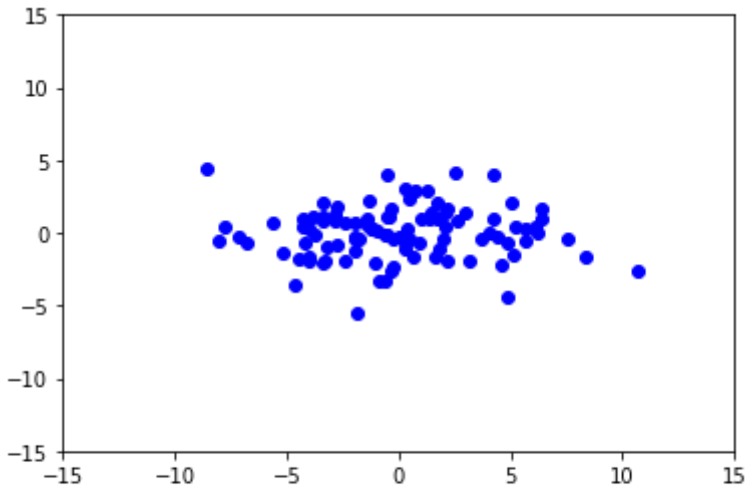
In [21]:
```
# (c)
evalue, evectors = np.linalg.eig(m)
print("Eigenvalues: \n", evalue)
print("Eigenvectors: \n", evectors)
```

```
Eigenvalues:
 [ 14.58037084   3.20038344]
Eigenvectors:
 [[ 0.79739029 -0.60346394]
 [ 0.60346394  0.79739029]]
```

In [26]:
```
# (d)
arrow1 = evalue[0] * evectors[:,0]
arrow2 = evectors[:, 1] * evalue[1]

# Plot
fig, axes = plt.subplots()
axes.scatter(x1, x2)
plt.quiver(x1_mean, x2_mean, arrow1[0], arrow1[1], angles='xy', scale_units=
plt.quiver(x1_mean, x2_mean, arrow2[0], arrow2[1], angles='xy', scale_units=
plt.axis([-15,15, -15,15])
plt.show()
```



In [27]:
```
# (e)
x1_centered = np.reshape(np.subtract(x1, x1_mean), (len(x1), 1))
x2_centered = np.reshape(np.subtract(x2, x2_mean), (len(x2), 1))
x_centered = np.hstack([x1_centered, x2_centered])
x_rotated = []
for ptr in x_centered:
    new_ptr = np.matrix(evectors).T * np.reshape(ptr, (2, 1))
    x_rotated.append([new_ptr[0, 0], new_ptr[1, 0]])

x_rotated = np.array(x_rotated)
x1 = x_rotated[:, 0]
x2 = x_rotated[:, 1]

plt.axis([-15, 15, -15, 15])
plt.plot(x1, x2, 'bo')
plt.show()
```



In [ ]:

4 Maximum Likelihood Estimation

(a) pdf for multivariate normal distribution $N(\mu, \Sigma)$:

$$p(x|\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}\Sigma^{1/2}} e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}$$

$$L(\mu,\Sigma|X) = \Pi_{i=1}^{n} P(X = x_i) = (\frac{1}{(2\pi)^{n/2}\Sigma^{1/2}})^n e^{(\frac{1}{2}\Sigma^{-1})^n \sum_{i=1}^{n}(x_i-\mu)^T(x_i-\mu)}$$

$$\log(L(\mu,\Sigma|X)) = -n\log((2\pi)^{n/2}\Sigma^{1/2}) - \frac{1}{2}\sum_{i=1}^{n}(x_i-\mu)^T\Sigma^{-1}(x_i-\mu)$$

$$= -n\log((2\pi)^{n/2}) - \frac{n}{2}\log(\Sigma) - \frac{1}{2}\sum_{i=1}^{n}(x_i-\mu)^T\Sigma^{-1}(x_i-\mu)$$

Compute $\hat{\mu}$:

$$\frac{d\log L}{d\mu} = \frac{1}{2}[2c_1(x_1-\mu) + 2c_2(x_2-\mu)...2c_n(x_n-\mu)] = \sum_{i=1}^{n} x_i - n\mu = 0$$

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

Compute $\hat{\Sigma}$:
According to the "trace trick" I found on CMU lecture slides (https://www.cs.cmu.edu/ epxing/Class/
08s/recitation/gaussian.pdf):

$$L(\mu,\Sigma|X) \propto -\frac{n}{2}\log(\Sigma) - \frac{1}{2}Trace(\Sigma^{-1}\sum_{i=1}^{n}(x_i-\mu)(x_i-\mu)^T)$$

Because $\frac{d}{dA}\log A = A^{-T}$ and $\frac{d}{dA}Tr(AB) = \frac{d}{dA}Tr(BA) = B^T$:

$$\frac{d\log L}{d\Sigma} = -\frac{n}{2}\Sigma + \frac{1}{2}\sum_{i=1}^{n}(x_i-\mu)(x_i-\mu)^T = 0$$

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

(b) MLE for $A\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$
Because A is invertible

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

5 Covariance Matirces and Decompositions

(a) **Answer: Σ is not invertible when 1. there is one or more than one deterministic-value features; 2. there is one or more than one features that are dependent on others.**
A matrix is invertible if all of its eigenvalues $\neq 0$
Case 1: ≥ 1 deterministic features
If a feature f_i always equals to certain value, $Cov(f_i, f_j)$ and $Cov(R_j, R_i)$ for j = 1, 2...n all equal 0, so there is a whole row of 0 and a whole column of 0 in the covariance matrix. This means one of the matrix's eigenvalue is 0. Geometrically, all sample points will have the same value for feature f_i, which means they are "squeezed" on a R^{n-1}-dimension space instead of R^n-dimension, and the eigenvector corresponding to f_i is 0. For example in a 2D space with features x and y, and y is constant, then all the sample points will spread on a horizontal line. If we plot eigenvector arrows on the plot, we won't see one that points in y-axis direction.

Case 2: ≥ 1 features are dependent on other features
This means some eigenvectors might be dependent on others. Geometrically, if we assume there are 2 features x and y and y is dependent on x in a way that all y values are a certain multiple of x, we will see sample points only spread on a line. The eigenvector of y will overlap with that of x although might with a different magnitude.

(b) **Approach 1**: delete all dependent features which means delete the corresponding rows and columns that concern these features

Approach 2: to make the eigenvalue not 0, we can add a really small γ constant to it.

$$Ax = x$$
$$I\gamma x = \beta x$$
$$Ax + I\gamma x = \beta x + \gamma x$$
$$(A + I\gamma)x = (\beta + \gamma)x$$

So we just add $I\gamma$ to the covariance matrix, and we can use accuracy on validation dataset to tune this γ value.

(c) **To maximize $f(x)$, choose the eigenvector that corresponds to the maximum eigenvalue; to minimize $f(x)$, choose the eigenvector that corresponds to the minimum eigenvalue.**

$$f(x) = \frac{1}{(2\pi)^{n/2}\Sigma^{1/2}} e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}$$

To maximize $f(x)$, we need to minimize $\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)$. Because $\mu = 0$, so we are minimizing $x^T\Sigma^{-1}x$
Becuase Σ is positive definite and invertible,

$$\Sigma = PDP^T$$

where P is a unitary matrix consist of orthonormal eigenvectors and D is diagonal matrix consist of eigenvalues λ_i. Therefore, $P^{-1} = P^T$ and D^{-1} is also diagonal matrix consist of σ eigenvalue's inverse.

$$\Sigma^{-1} = (PDP^T)^{-1} = (P^T)^{-1}D^{-1}P^{-1} = PD^{-1}P^T$$

Set S as a diagonal matrix consist of the square root of D^{-1}'s diagonal entries s.t. $D^{-1} = SS$

$$\Sigma^{-1} = PD^{-1}P^T = PSSP^T = PSS^TP^T$$

Set $A = PS$, then

$$\Sigma^{-1} = A^TA$$

$$x^T\Sigma^{-1}x = \|Ax\|_2^2$$

$$\|Ax\|_2^2 = x^TA^TAx = x^TPD^{-1}P^Tx$$

Set $v = Px$. Because P is unitary and $\|x\|_2^2 = 1$, we choose any x vector that is in the Euclidean basis vector form (consist of one "1" entry and rest are all "0"). And x can be transformed by P to another Euclidean basis vector. In other words, different x can single out different eigenvectors in P.

$$x^T\Sigma^{-1}x = \|Ax\|_2^2 = v^TD^{-1}v$$

The above means we can choose a v vector to single out any diagonal entries on D^{-1}. As a reminder, D^{-1}'s diagonal consist of the inverse of Σ^{-1}'s eigenvalues, $\frac{1}{\lambda_i}$. Therefore, in order to minimize $\|Ax\|_2^2$, we need $\frac{1}{\lambda_{max}}$ So we need to choose x that can single out the eigenvector that corresponds to the maximal eigenvalue of Σ^{-1}.
Minimizing $f(x)$ has opposite process, which is to find x that singles out the eigenvector that corresponds to the minimum eigenvalue.

MNIST

```
In [1]: import matplotlib.pyplot as plt
        import scipy.io as sio
        from scipy.stats import multivariate_normal
        import numpy as np
        import pandas as pd
        from sklearn.preprocessing import normalize
        %matplotlib inline
        import pdb
```

Load Data

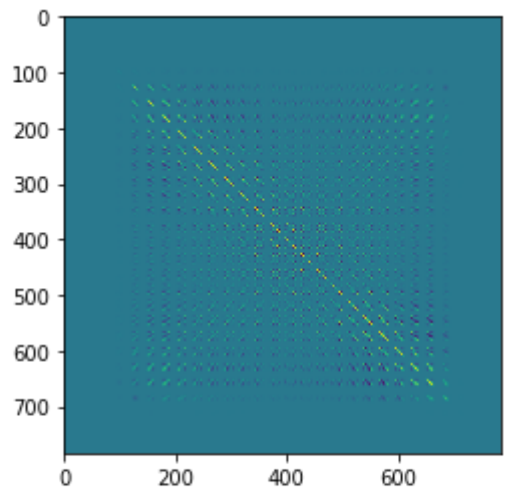
```
In [6]: trainMAT = sio.loadmat('./mnist/train.mat')
        testMAT = sio.loadmat('./mnist/test.mat')
        trainX = trainMAT["trainX"]
        trainY = trainX[:, -1:].reshape(1, len(trainX))[0]
        np.random.shuffle(trainX)
        validate_data = trainX[:10000, :]
        train_data = trainX[10000:, :]
        test_data = testMAT["testX"]
        CLASS_ = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        c = 0.0001
```

a) Fit Gaussian Distribution to each digit class using MLE

```
In [3]: def gaussian_mean_cov(data, label, if_print=False):
            df = pd.DataFrame(data)
            df = df[df[784]== label]
            features = normalize(df.values[:, :-1].astype(np.float32))
            mu = np.mean(features, axis=0)
            sigma = np.cov(features, rowvar=0)
            if if_print:
                print("Label ", label, " : ")
                print("Mu = ", mu)
                print("Sigma = ", sigma, "\n")
            return mu, sigma
```

b) Visualize covariance matrix

```
In [5]: plt.imshow(gaussian_mean_cov(train_data, 0)[1])

Out[5]: <matplotlib.image.AxesImage at 0x104163860>
```



Observation:

1. the values on one diagonal are comparably large. These values are the variances of the RVs, so it's reasonable that they are larger than any feature pair's covariance
2. There are parts of the values super small (the faded color on the graph), which means the features involved don't have much dependency on each other, or they might be always 0 (no color on this pixel)

c) LDA & QDA

```
In [15]: def lda_train(data):
             models = dict()
             x = normalize(data[:, :-1].astype(np.float32))
             sigma = np.cov(x, rowvar=0)
             for label in CLASS_:
                 mu, _ = gaussian_mean_cov(data, label)
                 m = multivariate_normal(mu, sigma + c * np.identity(sigma.shape[0]))
                 models[label] = m
             return models


         def qda_train(data):
             models = dict()
             for label in CLASS_:
                 mu, sigma = gaussian_mean_cov(data, label)
                 m = multivariate_normal(mu, sigma + c * np.identity(sigma.shape[0]))
                 models[label] = m
             return models


         def test(x, models):
             y = list()
             x = normalize(x.astype(np.float32))
             for sample in x:
                 prob = [models[label].logpdf(sample) + compute_prior(trainY, label)
                 y.append(np.argmax(prob))
             return y


         def batch_train_and_evaluate(train_data, validate_data, categories, type="LI
             errors = []
             for size in categories:
                 models = qda_train(train_data[:size, :])
                 if type == "LDA":
                     models = lda_train(train_data[:size, :])
                 prediction = test(validate_data[:, :-1], models)
                 y = validate_data[:, -1:].reshape(1, len(validate_data))[0]
                 err = 1 - np.mean(np.equal(prediction, y).astype(np.int32))
                 errors.append(err)
                 print("Error for training size {}: {}".format(size, err))

             plt.plot(categories, errors, 'ro')
             plt.axis([min(categories)-10, max(categories)+10, -0.1, max(errors)+0.1]
             plt.title("{} Error rate".format(type))
             return models

         def compute_prior(data, label):
             return np.mean(np.equal(data, label).astype(np.int32))
```

Spam

```
In [1]: import matplotlib.pyplot as plt
        import scipy.io as sio
        from scipy.stats import multivariate_normal
        import numpy as np
        import pandas as pd
        import math
        from sklearn.preprocessing import normalize
        from sklearn.feature_extraction.text import TfidfVectorizer
        %matplotlib inline
        import glob as g
        import re
        import pdb
```

Load Data

```
In [2]: spamfiles = g.glob('./spam/spam/*.txt')
        hamfiles = g.glob('./spam/ham/*.txt')
        testfiles = ['./spam/test/' + str(i) + '.txt' for i in range(10000)]
```

```
In [3]: def load_process_files(files):
            txts = list()
            for file in files:
                txt = open(file, "r", encoding='utf-8', errors='ignore').read()
                txt = txt.replace('\r\n', ' ')
                txts.append(txt)
            return txts
```

```
In [4]: spams = load_process_files(spamfiles)
        hams = load_process_files(hamfiles)
        tests = load_process_files(testfiles)
        trains = spams + hams
        all = trains + tests
```

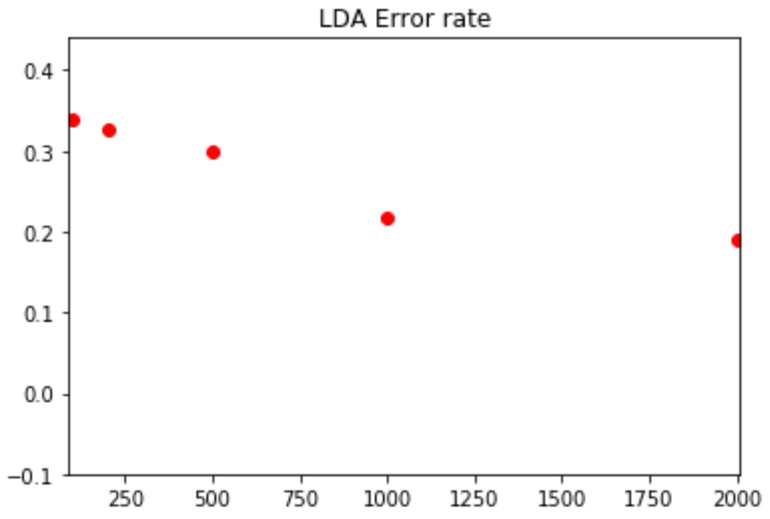
In [5]:
```
Ns = len(spams); Nh = len(hams); Nt = len(tests)
validate_n = 10000
vectorizer = TfidfVectorizer(min_df=0.05)

data = vectorizer.fit_transform(all).toarray()
train_data = data[:-10000]
test_data = data[-10000:]

spam_data = train_data[:Ns]
spam_validate = spam_data[validate_n:]

ham_data = train_data[Ns:]
ham_validate = ham_data[validate_n:]

spam_prior =  math.log(1.0 * Ns / (Ns  + Nh))
ham_prior = math.log(1.0 * Nh / (Ns  + Nh))

c = 0.0001
```

Model

In [15]:
```
def gaussian_mean_cov(data):
    mu = np.mean(data, axis=0)
    sigma = np.cov(data, rowvar=0)
    return mu, sigma


def lda_train(spam, ham, all):
    x = normalize(all.astype(np.float32))
    sigma = np.cov(x, rowvar=0)
    mu1, _ = gaussian_mean_cov(spam)
    mu2, _ = gaussian_mean_cov(ham)
    m1 = multivariate_normal(mu1, sigma + c * np.identity(sigma.shape[0]))
    m2 = multivariate_normal(mu2, sigma + c * np.identity(sigma.shape[0]))
    return m1, m2


def qda_train(spam, ham):
    mu1, sigma1 = gaussian_mean_cov(spam)
    mu2, sigma2 = gaussian_mean_cov(ham)
    m1 = multivariate_normal(mu1, sigma1 + c * np.identity(sigma1.shape[0]))
    m2 = multivariate_normal(mu2, sigma2 + c * np.identity(sigma2.shape[0]))
    return m1, m2


def test(x, spam_m, ham_m):
    y = list()
    for sample in x:
        spam_y = spam_m.logpdf(sample) + spam_prior
        ham_y = ham_m.logpdf(sample) + ham_prior
        y.append(np.argmax([ham_y, spam_y]))
    return y


def evaluate(spam_data, ham_data, spam_m, ham_m):
    spam_y = test(spam_data, spam_m, ham_m)
    ham_y = test(ham_data, spam_m, ham_m)
    correct = np.count_nonzero(spam_y) + (len(ham_y) - np.count_nonzero(ham_
    total = len(spam_y) + len(ham_y)
    return 1.0 * correct / total


def batch_train_and_evaluate(spam_data, ham_data, spam_validate, ham_validat
    errors = []
    for size in categories:
        spam_m, ham_m = qda_train(spam_data[:size, :], ham_data[:size, :])
        if type == "LDA":
            all = np.concatenate((spam_data, ham_data), axis=0)
            spam_m, ham_m = lda_train(spam_data[:size, :], ham_data[:size, :
        err = 1 - evaluate(spam_validate, ham_validate, spam_m, ham_m)
        errors.append(err)
        print("Error for training size {}: {}".format(size, err))

    plt.plot(categories, errors, 'ro')
    plt.axis([min(categories)-10, max(categories)+10, -0.1, max(errors)+0.1]
    plt.title("{} Error rate".format(type))
    return spam_m, ham_m
```

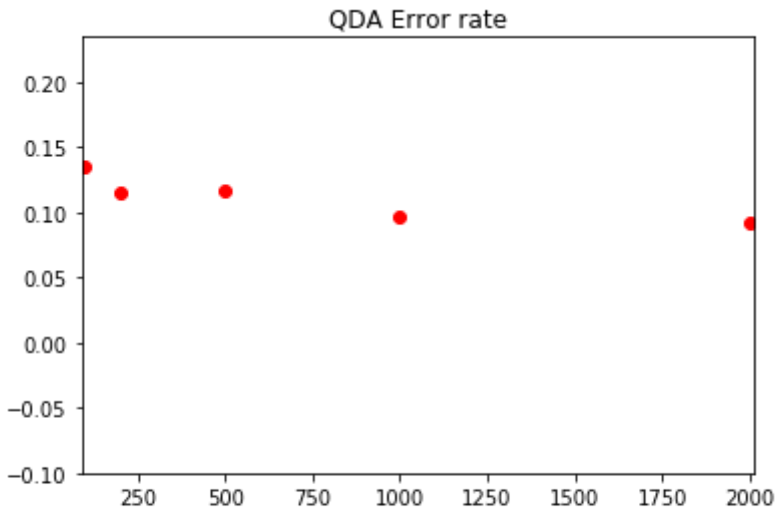
In [16]:
```
# Train
categories = [100, 200, 500, 1000, 2000]
lda_models = batch_train_and_evaluate(spam_data, ham_data, spam_validate, ha
```

```
Error for training size 100: 0.3400864397622907
Error for training size 200: 0.32685035116153427
Error for training size 500: 0.3001080497028633
Error for training size 1000: 0.217720151269584
Error for training size 2000: 0.18989735278227982
```



In [17]:
```
qda_models = batch_train_and_evaluate(spam_data, ham_data, spam_validate, ha
```

```
Error for training size 100: 0.13479200432198812
Error for training size 200: 0.11561318206374938
Error for training size 500: 0.11642355483522415
Error for training size 1000: 0.09751485683414374
Error for training size 2000: 0.09292274446245274
```



In [24]:
```
# Test
lda_y = test(test_data, lda_models[0], lda_models[1])
qda_y = test(test_data, qda_models[0], qda_models[1])
```

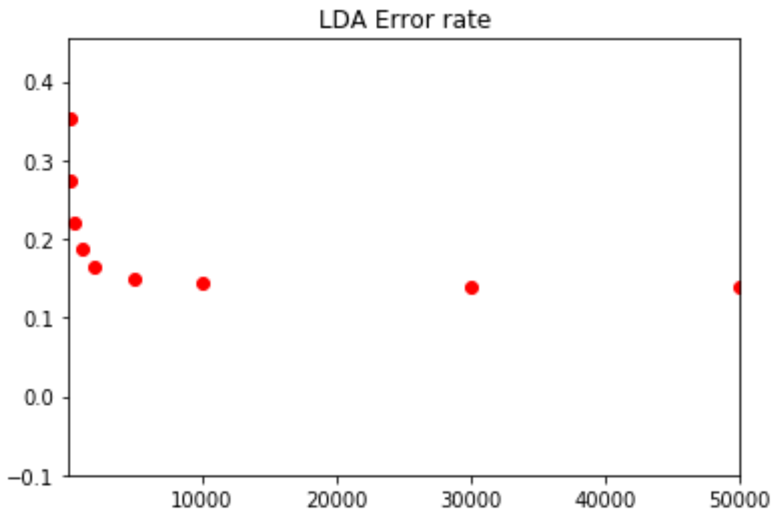
```
In [25]: df = pd.DataFrame(data = lda_y, columns=["Category"])
         df.index.name = "Id"
         df.to_csv("./spam.csv")
```

Kaggle: 0.95520

Feature Description:
Used bag of words approach plus normalization. More specifically, the features are the freqencies of each unique word in the text; after normalization, they become probabilities.

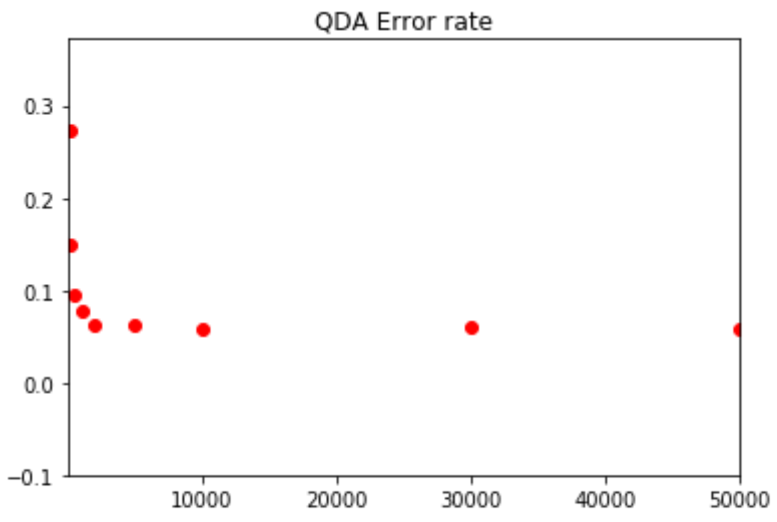
```
In [16]: # Train and Error
         categories = [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000]
         lda_models = batch_train_and_evaluate(train_data, validate_data, categories)
```

```
Error for training size 100: 0.35419999999999996
Error for training size 200: 0.2751
Error for training size 500: 0.22060000000000002
Error for training size 1000: 0.18779999999999997
Error for training size 2000: 0.16559999999999997
Error for training size 5000: 0.14970000000000006
Error for training size 10000: 0.14370000000000005
Error for training size 30000: 0.14
Error for training size 50000: 0.1392
```



In [17]:
```
qda_models = batch_train_and_evaluate(train_data, validate_data, categories,
```

```
Error for training size 100: 0.2723
Error for training size 200: 0.14900000000000002
Error for training size 500: 0.09509999999999996
Error for training size 1000: 0.07779999999999998
Error for training size 2000: 0.06340000000000001
Error for training size 5000: 0.062000000000000055
Error for training size 10000: 0.05900000000000005
Error for training size 30000: 0.05930000000000002
Error for training size 50000: 0.05810000000000004
```



In [9]:
```
# Test
lda_y = test(test_data, lda_models)
qda_y = test(test_data, qda_models)
```

c) LDA vs QDA

QDA is better. Because we have 10 classes in this problem, but LDA uses the same variance for all their distributions, which isn't reasonable because different class random variable might have very different variance.

In [21]:
```
df = pd.DataFrame(data = qda_y, columns=["Category"])
df.index.name = "Id"
df.to_csv("./mnist.csv")
```

Kaggle: 0.95640