Name: Hye Soo Choi **Student ID: 23274190**

CS 189: Introduction to Machine Learning

Homework 3

Due: March 3, 2016 at 11:59pm

Problem 1: Independence vs. Correlation.

(a) The joint probability density table of (X, Y) is drawn as below.

$X \backslash Y$	-1	0	1
-1	0	1/4	0
0	1/4	0	1/4
1	0	1/4	0

Therefore,

$$E[XY] = 0,$$

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

Since $E[XY] = E[X]E[Y]$, X and Y are uncorrelated. X and Y are not independent because

$$0 = P\{X = 0, Y = 0\} \neq P\{X = 0\}P\{Y = 0\} = \frac{1}{2} \cdot \frac{1}{2}.$$

(b) X, Y, Z are pairwise independent. This is because

$$P\{X = 0\} = P\{X = 1\} = P\{Y = 0\} = P\{Y = 1\} = P\{Z = 0\} = P\{Z = 1\} = \frac{1}{2},$$

and, no matter what value X might have, $Y|X$ always takes a value $\{0, 1\}$ with equal probability since B_3, which is independent of X, takes a value of $\{0, 1\}$ with equal probability. Therefore the conditional distribution of Y given X is the same as the original distribution of Y. Therefore X and Y are independent. By symmetry, we can easily prove that Y and Z, Z and X are pairwise independent as well.
Since it is always true that

$$X \oplus Y \oplus Z = (B_1 \oplus B_2) \oplus (B_2 \oplus B_3) \oplus (B_3 \oplus B_1) = (B_1 \oplus B_1) \oplus (B_2 \oplus B_2) \oplus (B_3 \oplus B_3) = 0 \oplus 0 \oplus 0 = 0,$$

X,Y,Z are not mutually independent. To be more specific,

$$0 = P\{X = 0, Y = 0, Z = 1\} \neq P\{X = 0\}P\{Y = 0\}P\{Z = 1\} = \frac{1}{8}.$$

Problem 4: Covariance Matrixes and Decompositions.

(a) The inverse of Σ_X will not exist if and only if(TFAE)

- Σ_X has determinant zero,
- Σ_X has at least one eigenvalue of zero,
- there exists nonzero $y \in R^N$ such that $y^\top \Sigma_X y = 0$,
- there exists nonzero $y \in R^N$ such that $E[(y^\top (X - \mu))^2] = 0$,
- there exists nonzero $y \in R^N$ such that $y^\top (X - \mu) = 0$ almost surely,
- there exists nonzero $y \in R^N$ such that $y^\top X$ is some constant almost surely,
- there exists some random variable X_i which can be expressed as a linear combination of other X_j's.

We can remove all the X_i's which are expressed as a linear combination of other X_j's and preserve only the smallest number of X_j's that span all X_i's. By doing so, we can transform X into X' whose $\Sigma_{X'}$ is invertible, without losing any information: By a linear combination, we are able to restore the removed elements X_i's always.

(b) Let's denote the spectral decomposition of Σ^{-1} as $UDU^\top$, where $D = \text{diag}(\lambda_i)$ is a diagonal matrix along with the eigenvalues of Σ^{-1} and U is a matrix whose columns are corresponding normalized eigenvectors of length 1. Write $D^{\frac{1}{2}}$ as $\text{diag}(\lambda_i^{\frac{1}{2}})$, then

$$x^\top \Sigma^{-1} x = x^\top U D^{\frac{1}{2}} D^{\frac{1}{2}} U^\top x = ||D^{\frac{1}{2}} U^\top x||_2^2.$$

It follows that $A = D^{\frac{1}{2}} U$.

(c) When we transform it to $||Ax||_2^2$, $x^\top \Sigma^{-1} x$ have intuitive meaning of a squared distance from origin after rotating x around origin, with the rotation matrix $U^\top$ and either stretching or contracting the rotated vector by size of eigenvalues. note that the rotation transforms all eigenvector onto a standard axis. By multiplying a diagonal matrix $D^{\frac{1}{2}}$, a vector is stretched or contracted along standard axis .
(d) Observe that

$$\min_{x:||x||_2=1} ||Ax||_2 = \min_{x:||x||_2=1} ||D^{\frac{1}{2}} U^\top x||_2 = \min_{x:||x||_2=1} ||D^{\frac{1}{2}} x||_2$$

$$\max_{x:||x||_2=1} ||Ax||_2 = \max_{x:||x||_2=1} ||D^{\frac{1}{2}} U^\top x||_2 = \max_{x:||x||_2=1} ||D^{\frac{1}{2}} x||_2.$$

Since $D^{\frac{1}{2}}$ is a diagonal matrix, the minimum of $||Ax||_2^2$ is just the square of the minimum diagonal values of $D^{\frac{1}{2}}$, that is, the minimum eigenvalue of Σ^{-1}. Similarly, the maximum of $||Ax||_2^2$ is just the square of the maximum diagonal values of $D^{\frac{1}{2}}$, that is, the maximum eigenvalue of Σ^{-1}. To maximize $f(x)$, we should minimize $x^\top \Sigma x$ and therefore we should choose the eigenvector that matches with the smallest eigenvalues of Σ^{-1}. This is because, to minimize $x^\top \Sigma x$, it should be that $U^\top x = e_i$, where λ_i is the smallest eigenvalue. Equivalently, $x = U e_i$, the eigenvector that matches with λ_i.

If X_i's are pairwise independent, then covariance matrix Σ becomes a diagonal matrix whose diagonal elements are the variance of X_i's, and U is an N dimensional identity matrix. This implies that an eigenvalue λ_i of Σ^{-1} is equal to a inverse of $Var(X_i)$, for $1 \leq i \leq N$. Thus, the minimum of $||Ax||_2^2$ is the minimum of $\frac{1}{Var(X_i)}, 1 \leq i \leq N$, and likewise the maximum of $||Ax||_2^2$ is the maximum of $\frac{1}{Var(X_i)}, 1 \leq i \leq N$. To maximize $f(x)$, we should choose an elementary vector $e_{i'}$, where $i' = \text{argmax}_i Var(X_i)$.

HW3

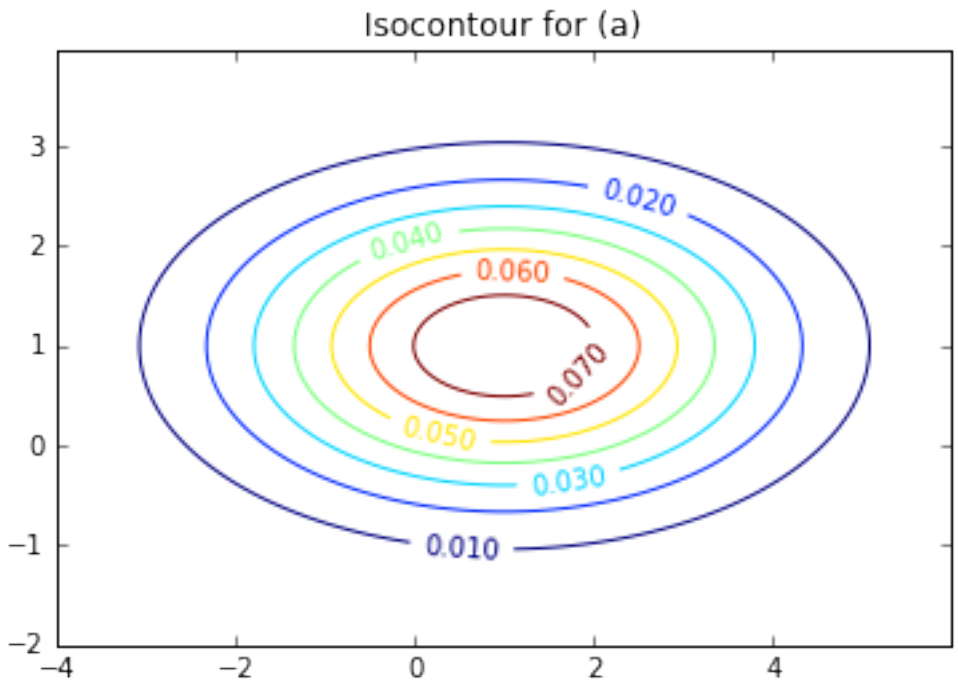
March 4, 2016

1 Problem 2: Isocontours of Normal Distributions

In [30]:
```
#import packages

import matplotlib
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
%matplotlib inline
```

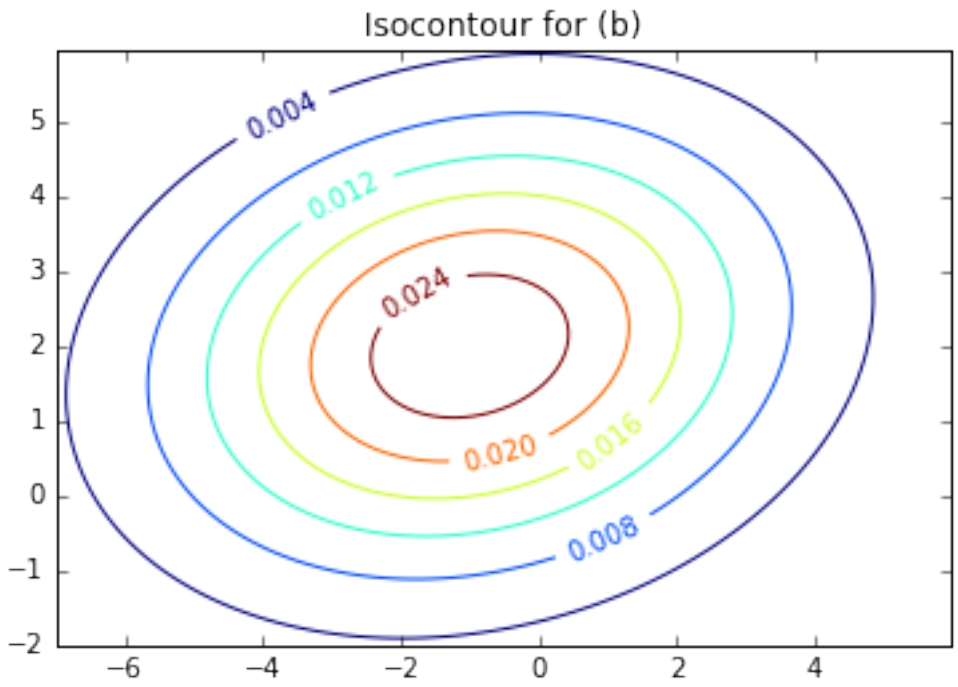
a) For Gaussian probability density function $f(\mu; \Sigma)$, where $\mu = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$, the isocontour is drawn as follows.

In [37]:
```
delta = 0.025
x = np.arange(-4.0, 6.0, delta)
y = np.arange(-2.0, 4.0, delta)
X, Y = np.meshgrid(x, y)
Z = mlab.bivariate_normal(X, Y, 2.0, 1.0, 1.0, 1.0)

plt.figure()
CS = plt.contour(X, Y, Z)
plt.clabel(CS, inline=1, fontsize=10)
plt.title('Isocontour for (a)')
plt.show()
```



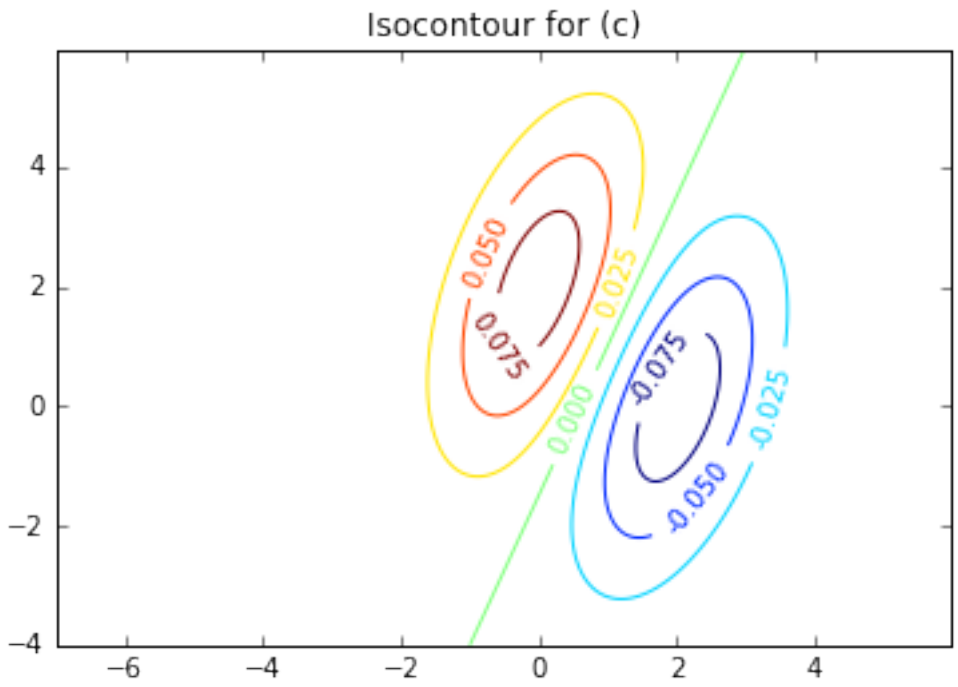
b) For Gaussian probability density function $f(\mu; \Sigma)$, where $\mu = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$, the isocontour is drawn as follows.

```
In [41]: delta = 0.025
         x = np.arange(-7.0, 6.0, delta)
         y = np.arange(-2.0, 6.0, delta)
         X, Y = np.meshgrid(x, y)
         Z = mlab.bivariate_normal(X, Y, 3.0, 2.0, -1.0, 2.0, 1.0)

         plt.figure()
         CS = plt.contour(X, Y, Z)
         plt.clabel(CS, inline=1, fontsize=10)
         plt.title('Isocontour for (b)')
         plt.show()
```



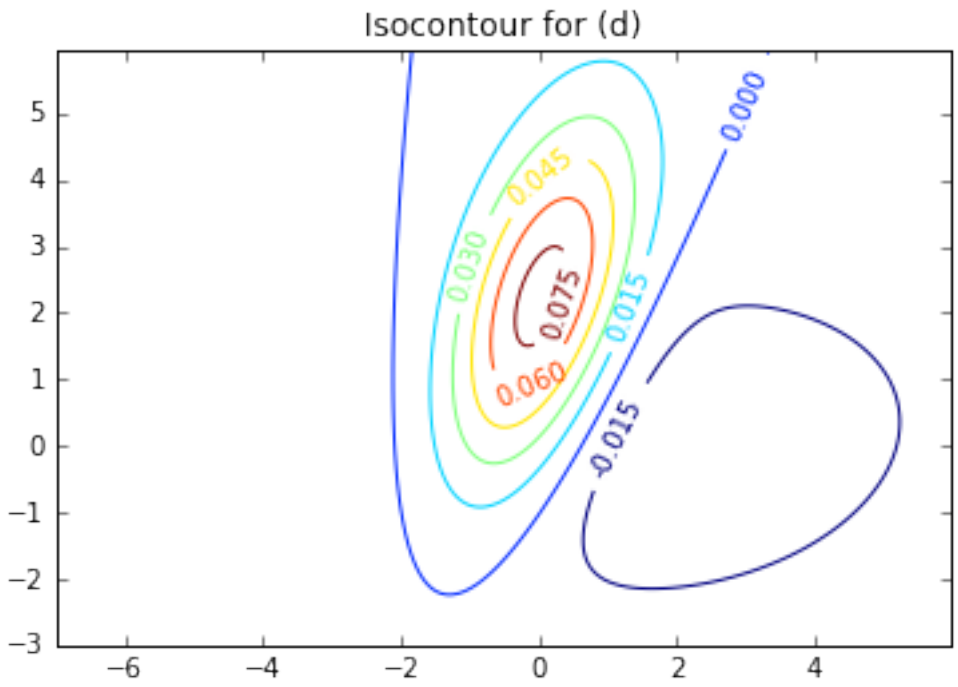
c) If we have two Gaussian probability density functions $f(\mu_1; \Sigma_1)$, where $\mu_1 = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$, and $f(\mu_2; \Sigma_2)$, where $\mu_2 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$, then the isocontour of their difference $f(\mu_1, \Sigma_1) - f(\mu_2, \Sigma_2)$ is drawn as follows.

```
In [45]: delta = 0.025
         x = np.arange(-7.0, 6.0, delta)
         y = np.arange(-4.0, 6.0, delta)
         X, Y = np.meshgrid(x, y)
         Z1 = mlab.bivariate_normal(X, Y, 1.0, 2.0, 0.0, 2.0, 1.0)
         Z2 = mlab.bivariate_normal(X, Y, 1.0, 2.0, 2.0, 0.0, 1.0)
         Z = Z1- Z2

         plt.figure()
         CS = plt.contour(X, Y, Z)
         plt.clabel(CS, inline=1, fontsize=10)
         plt.title('Isocontour for (c)')
         plt.show()
```



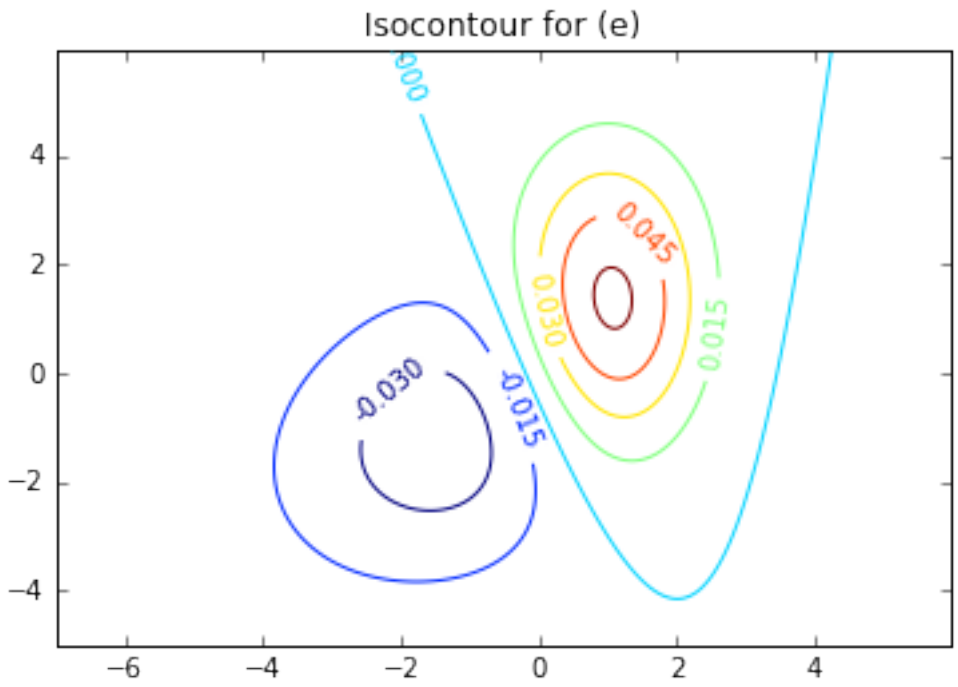
d) If we have two Gaussian probability density functions $f(\mu_1; \Sigma_1)$, where $\mu_1 = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$, and $f(\mu_2; \Sigma_2)$, where $\mu_2 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$, then the isocontour of their difference $f(\mu_1, \Sigma_1) - f(\mu_2, \Sigma_2)$ is drawn as follows.

```
In [47]: delta = 0.025
         x = np.arange(-7.0, 6.0, delta)
         y = np.arange(-3.0, 6.0, delta)
         X, Y = np.meshgrid(x, y)
         Z1 = mlab.bivariate_normal(X, Y, 1.0, 2.0, 0.0, 2.0, 1.0)
         Z2 = mlab.bivariate_normal(X, Y, 3.0, 2.0, 2.0, 0.0, 1.0)
         Z = Z1- Z2

         plt.figure()
         CS = plt.contour(X, Y, Z)
         plt.clabel(CS, inline=1, fontsize=10)
         plt.title('Isocontour for (d)')
         plt.show()
```



e) If we have two Gaussian probability density functions $f(\mu_1; \Sigma_1)$, where $\mu_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$, and $f(\mu_2; \Sigma_2)$, where $\mu_2 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$, then the isocontour of their difference $f(\mu_1, \Sigma_1) - f(\mu_2, \Sigma_2)$ is drawn as follows.

```
In [49]: delta = 0.025
         x = np.arange(-7.0, 6.0, delta)
         y = np.arange(-5.0, 6.0, delta)
         X, Y = np.meshgrid(x, y)
         Z1 = mlab.bivariate_normal(X, Y, 1.0, 2.0, 1.0, 1.0, 0.0)
         Z2 = mlab.bivariate_normal(X, Y, 2.0, 2.0, -1.0, -1.0, 1.0)
         Z = Z1- Z2

         plt.figure()
         CS = plt.contour(X, Y, Z)
         plt.clabel(CS, inline=1, fontsize=10)
         plt.title('Isocontour for (e)')
         plt.show()
```



2 Problem 3: Visualizing Eigenvectors of Gaussian Covariance Matrix

We have two one-dimensional random variables $X_1 \sim N(3;9)$ and $X_2 \sim \frac{1}{2}X_1 + N(4;4)$, where $N(\mu;\sigma^2)$ is a Gaussian distribution with mean μ and variance σ^2. In software, draw $N = 100$ random samples of X_1 and of X_2.

```
In [95]: import numpy as np
         import matplotlib.pyplot as plt
         %matplotlib inline
```

```
---------------------------------------------------------------------------

ImportError                               Traceback (most recent call last)

<ipython-input-95-abd20ae3475c> in <module>()
      1 import numpy as np
      2 import matplotlib.pyplot as plt
----> 3 import matplotlib.axes.Axes as ma
      4 get_ipython().magic('matplotlib inline')


ImportError: No module named 'matplotlib.axes.Axes'
```

```
In [127]: np.random.seed(10)
          X1 = np.random.normal(3,3, 100)
```

```
X2 = X1 + np.random.normal(4,2,100)

#compute the mean and covariance of the sampled data.

mean1 = np.mean(X1)
mean2 = np.mean(X2)
Z = np.vstack((X1,X2))
cov = np.cov(Z)
```

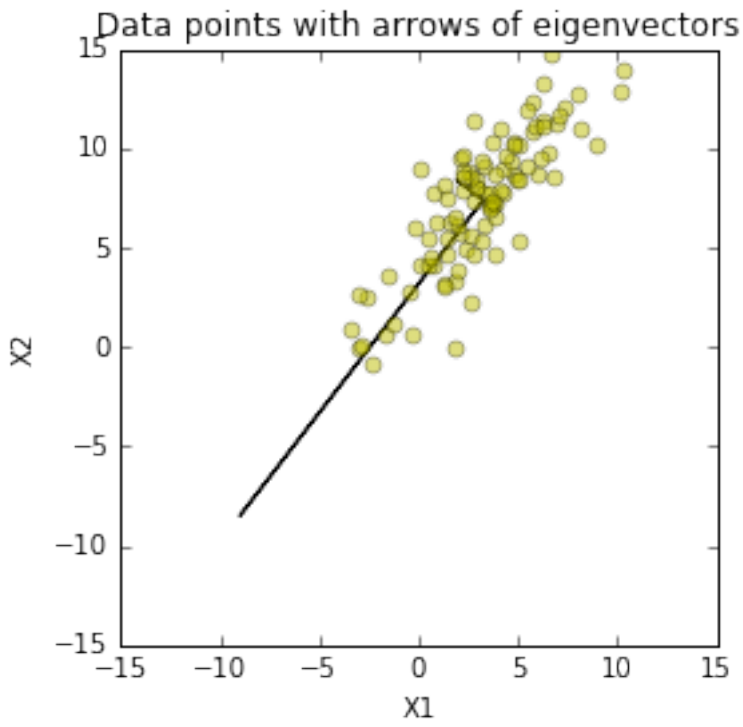
In [130]:
```
#compute the eigenvectors and eigenvalues of this covariance matrix.
eigen = np.linalg.eig(cov)
eigvalues = eigen[0]
eigvectors = eigen[1]
print(eigvalues)
eigvectors
```

```
[  1.65887353  20.01851492]
```

Out[130]:
```
array([[-0.79202303, -0.61049122],
       [ 0.61049122, -0.79202303]])
```

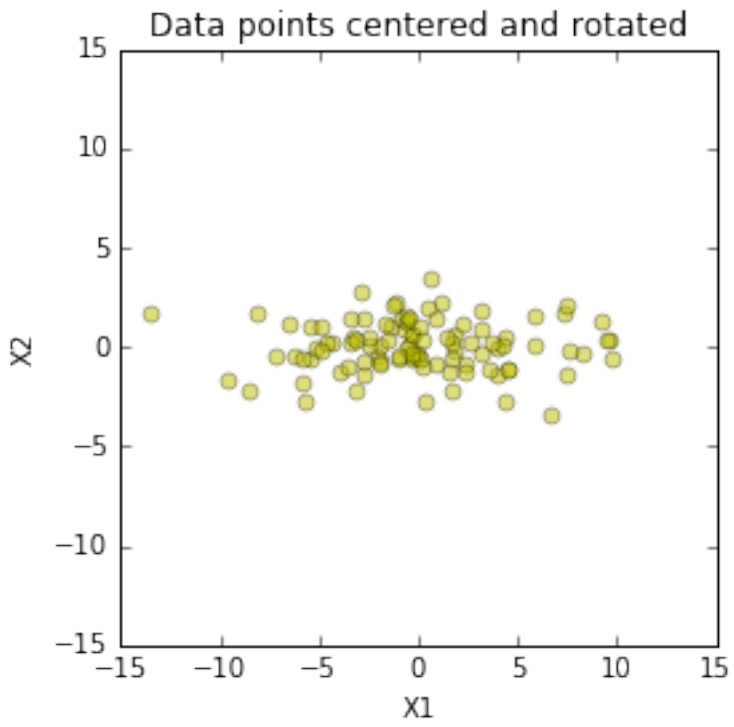
Therefore, covariance matrix is $\begin{bmatrix} 8.50150318 & 8.87731074 \\ 8.87731074 & 13.17588527 \end{bmatrix}$ and the mean vector is $[3.2382499888106211, 7.3766443562036113]^\top$. The eigenvalues of the covariance matrix are 1.65887353 and 20.01851492, and the corresponding eigenvectors are (-0.79202303, 0.61049122) and (-0.61049122, -0.79202303) respectively.

In [134]:
```
plt.figure()
plt.plot(X1,X2,'yo', alpha = 0.5)
plt.axis([-15,15,-15,15])
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Data points with arrows of eigenvectors')
plt.arrow(mean1,mean2, eigvalues[0] * eigvectors[0,0], eigvalues[0]*eigvectors[1,0], color =
plt.arrow(mean1,mean2, eigvalues[1] * eigvectors[0,1], eigvalues[1]*eigvectors[1,1], color =
plt.axes().set_aspect('equal')
plt.show()
```



In [137]:
```
# rotation matrix
U = eigvectors
U = U[...,[1,0]]
print(U)
# center data points by subtracting the mean and then rotate each point.
Z_cr = np.dot(U.T, (Z.T - np.array([mean1, mean2])).T)

plt.figure()
plt.plot(Z_cr[0,],Z_cr[1,],'yo', alpha = 0.5)
plt.axis([-15,15,-15,15])
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Data points centered and rotated')
plt.axes().set_aspect('equal')
plt.show()
```

```
[[-0.61049122 -0.79202303]
 [-0.79202303  0.61049122]]
```



3 Problem 5: Gaussian Classiffiers for Digits

In this problem we will build Gaussian classiffiers for digits in MNIST. More specifically, we will model each digit class as a Gaussian distribution and make our decisions on the basis of posterior probabilities. This is a generative method for classifying images where we are modelling the class conditional probabilities as normal distributions.

In [2]:
```
#import packages

import scipy.io as sio
import numpy as np
```

In [ ]:
```
#load data

test_digit = sio.loadmat('./data/digit_dataset/test.mat')
train_digit = sio.loadmat('./data/digit_dataset/train.mat')
```

In [304]:
```
#extract test feature vectors

  test_img = test_digit['test_images']
```

Out[304]: (10000, 784)

In [140]:
```
#extract train feature vectors

  train_img = np.ravel(np.transpose(train_digit['train_images'])).reshape(60000,784)
```

Out[140]: (60000, 784)

3.0.1 a-1) Normalize contrast

To normalize contrast on images, we divided each feature vecotr by the maximum value of its elements.

In [141]:
```
#calculate the maximum of elements in each row vector.

train_max = train_img.max(1)
```

In [142]:
```
#normalize each train feature vector by its maximum elements.

train_img = np.divide(train_img, train_max[:, None])
```

In [312]:
```
#normalize each test feature vector by its maximum elements.

test_max = test_img.max(1)
test_img = np.divide(test_img, test_max[:, None])
```

3.0.2 a-2) The Maximum Likehood Estimates(MLE) for mean and covariance matrix

Say we have i.i.d observations $X_1, \ldots, X_n$. The maximum likelihood estimates for the mean and covariance matrix of a Gaussian distribution is as follows:

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

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

Here we want to find the maximum likelihood estimates for the mean and covariance matrix for each class.

In [152]:
```
#extract train labels

train_digit_lb = train_digit['train_labels'][...,0]
```

In [154]:
```
#classify train feature vectors by its train labels.

index_class0 = train_digit_lb == 0
index_class1 = train_digit_lb == 1
index_class2 = train_digit_lb == 2
index_class3 = train_digit_lb == 3
index_class4 = train_digit_lb == 4
index_class5 = train_digit_lb == 5
index_class6 = train_digit_lb == 6
index_class7 = train_digit_lb == 7
index_class8 = train_digit_lb == 8
index_class9 = train_digit_lb == 9
train_img0 = train_img[index_class0,]
train_img1 = train_img[index_class1,]
train_img2 = train_img[index_class2,]
train_img3 = train_img[index_class3,]
train_img4 = train_img[index_class4,]
train_img5 = train_img[index_class5,]
train_img6 = train_img[index_class6,]
train_img7 = train_img[index_class7,]
train_img8 = train_img[index_class8,]
train_img9 = train_img[index_class9,]
```

In [155]:
```
# calculate MLE for the mean and covariance matrix for each class.

mean0 = np.sum(train_img0, axis=0)/index_class0.sum()
cov0 = np.dot(np.transpose(train_img0 - mean0), train_img0 - mean0)/ index_class0.sum()
mean1 = np.sum(train_img1, axis=0)/index_class1.sum()
cov1 = np.dot(np.transpose(train_img1 - mean1), train_img1 - mean1)/ index_class1.sum()
mean2 = np.sum(train_img2, axis=0)/index_class2.sum()
cov2 = np.dot(np.transpose(train_img2 - mean2), train_img2 - mean2)/ index_class2.sum()
mean3 = np.sum(train_img3, axis=0)/index_class3.sum()
cov3 = np.dot(np.transpose(train_img3 - mean3), train_img3 - mean3)/ index_class3.sum()
mean4 = np.sum(train_img4, axis=0)/index_class4.sum()
cov4 = np.dot(np.transpose(train_img4 - mean4), train_img4 - mean4)/ index_class4.sum()
mean5 = np.sum(train_img5, axis=0)/index_class5.sum()
cov5 = np.dot(np.transpose(train_img5 - mean5), train_img5 - mean5)/ index_class5.sum()
mean6 = np.sum(train_img6, axis=0)/index_class6.sum()
cov6 = np.dot(np.transpose(train_img6 - mean6), train_img6 - mean6)/ index_class6.sum()
mean7 = np.sum(train_img7, axis=0)/index_class7.sum()
cov7 = np.dot(np.transpose(train_img7 - mean7), train_img7 - mean7)/ index_class7.sum()
mean8 = np.sum(train_img8, axis=0)/index_class8.sum()
cov8 = np.dot(np.transpose(train_img8 - mean8), train_img8 - mean8)/ index_class8.sum()
mean9 = np.sum(train_img9, axis=0)/index_class9.sum()
cov9 = np.dot(np.transpose(train_img9 - mean9), train_img9 - mean9)/ index_class9.sum()
```

3.0.3 b) Prior distribution

To get prior distribution, we calculate a ratio of the number of training data labelled as each class to the total number of training data.

In [185]:
```
prior = np.array([index_class0.sum(),index_class1.sum(),index_class2.sum(),index_class3.sum()
                  index_class5.sum(),index_class6.sum(),index_class7.sum(), index_class8.sum(), index_
prior = prior/prior.sum()
```

In [186]:
```
prior
```

Out[186]:
```
array([ 0.09871667,  0.11236667,  0.0993    ,  0.10218333,  0.09736667,
        0.09035   ,  0.09863333,  0.10441667,  0.09751667,  0.09915   ])
```

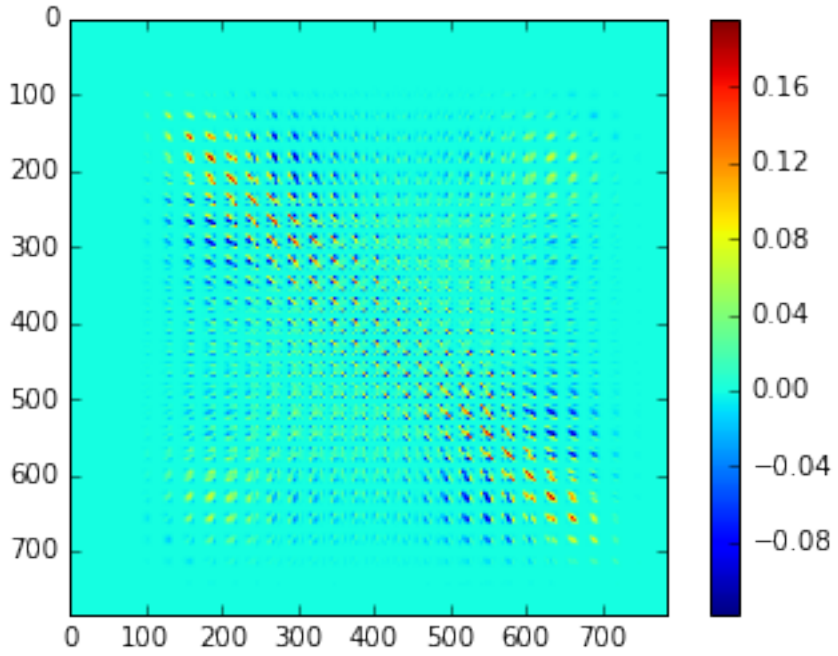
3.0.4 c) Visualize the covariance matrix

We visualize the covariance matrix for particular classes: 0,1,2. All the images that visualize the covariance matrices, as one might expect, are symmetric and have bigger value on the diagonal. This symbolizes that a feature has the largest covariance with itself, and covariance is symmetric. Also, many of two different features exhibit covariance near 0, which means that the two features are nearly indepent (since we are assuming the multivariate gaussian).

In [18]:
```
#import package necessary for drawing plot.

import matplotlib.pyplot as plt
```

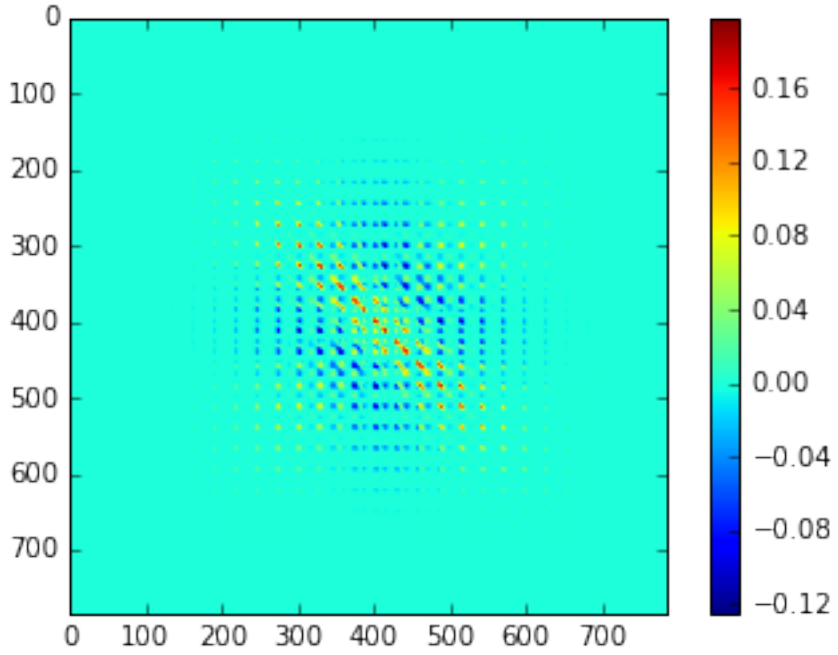
In [156]:
```
%matplotlib inline
imgplot = plt.imshow(cov0)
plt.colorbar()
```

Out[156]:
```
<matplotlib.colorbar.Colorbar at 0x10dc2d5f8>
```

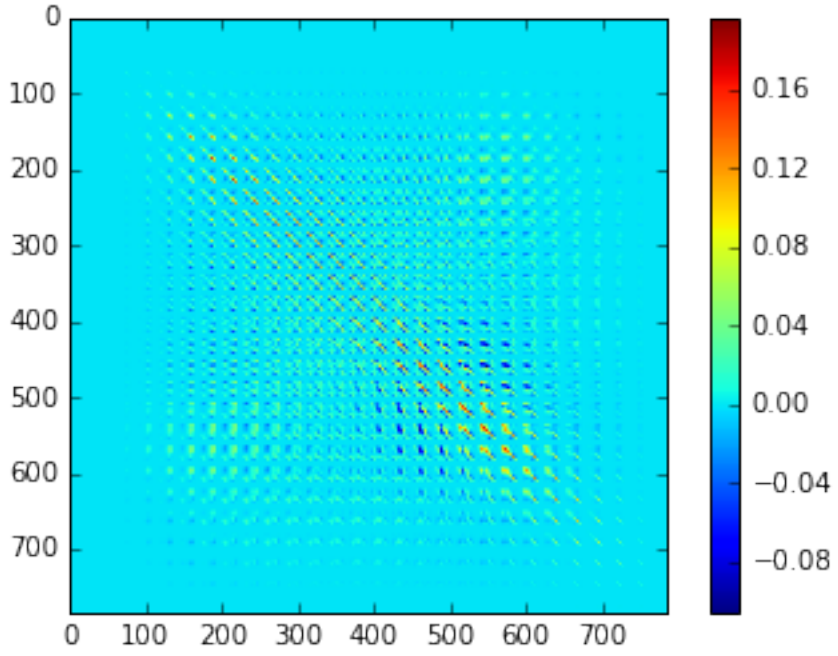


In [157]:
```
imgplot = plt.imshow(cov1)
plt.colorbar()
```

Out[157]: <matplotlib.colorbar.Colorbar at 0x10ddbe320>



```
In [158]: imgplot = plt.imshow(cov2)
          plt.colorbar()

Out[158]: <matplotlib.colorbar.Colorbar at 0x17eee1fd0>
```



3.0.5 d) Classify digits in the test set on the basis of posterior probabilities using two different approaches:

LDA Define $\Sigma_{overall}$ to be the average of the covariance matrices of all the classes. We will use this matrix as an estimate of the covariance of all the classes. This amounts to modelling class conditionals as Gaussians ($N(\mu_i; \Sigma_{overall})$) with different means and the same covariance matrix. Using this form of class conditional probabilities, classify the images in the test set into one of the 10 classes assuming 0-1 loss and compute the error rate and plot it over the following number of randomly chosen training data points [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 60000]. Note that this is not exactly

QDA We can also model class conditionals as $N(\mu_i; \Sigma_i)$, where Σ_i is the estimated covariance matrix for the ith class. Classify images in the test set using this form of the conditional probability (assuming 0-1 loss) and compute the error rate and plot it over the following number of randomly chosen training data points [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 60000].

```
In [19]: import math

In [213]: def approximate_inv(M):
              temp = M
              while np.linalg.det(temp) == 0:
                      temp = temp + 0.001 * np.identity(784)
              return np.linalg.inv(temp)

In [187]: np.random.seed(1)
          choice = np.random.choice(np.arange(60000), 10000, replace = False)
```

```
valid_img = train_img[choice, :]
valid_lb = train_digit_lb[choice]
nonchoice = np.setdiff1d(np.arange(60000),choice)
nonchoice.shape
```

Out[187]: (50000,)

We implement QDA/LDA here. In case when a covriance matrix has a zero determinant, or, is not positive definite, we add the same constant multiple of 784 by 784 identity matrix to covariance matrices for every class. We try to find as small constant as possible, to preserve the covariance matrix of original training features as much as possible.

```
In [272]: def gaussian_classifier(n):
              # n: number of training data that will be used
              # return: a list of two functions each of that, given a feature vector x, returns a predi
              #         either QDA or LDA which trained by n chosen training data.
              np.random.seed(0)
              choice = np.random.choice(np.arange(50000), n, replace = False)
              train_temp = train_img[nonchoice, : ][choice, :]
              lb_temp = train_digit_lb[nonchoice][choice]
              index_temp0 = lb_temp == 0
              index_temp1 = lb_temp == 1
              index_temp2 = lb_temp == 2
              index_temp3 = lb_temp == 3
              index_temp4 = lb_temp == 4
              index_temp5 = lb_temp == 5
              index_temp6 = lb_temp == 6
              index_temp7 = lb_temp == 7
              index_temp8 = lb_temp == 8
              index_temp9 = lb_temp == 9
              img_temp0 = train_temp[index_temp0,]
              img_temp1 = train_temp[index_temp1,]
              img_temp2 = train_temp[index_temp2,]
              img_temp3 = train_temp[index_temp3,]
              img_temp4 = train_temp[index_temp4,]
              img_temp5 = train_temp[index_temp5,]
              img_temp6 = train_temp[index_temp6,]
              img_temp7 = train_temp[index_temp7,]
              img_temp8 = train_temp[index_temp8,]
              img_temp9 = train_temp[index_temp9,]
              mean0 = np.sum(img_temp0, axis=0)/index_temp0.sum()
              cov0 = np.dot(np.transpose(img_temp0 - mean0), img_temp0 - mean0)/ index_temp0.sum()
              mean1 = np.sum(img_temp1, axis=0)/index_temp1.sum()
              cov1 = np.dot(np.transpose(img_temp1 - mean1), img_temp1 - mean1)/ index_temp1.sum()
              mean2 = np.sum(img_temp2, axis=0)/index_temp2.sum()
              cov2 = np.dot(np.transpose(img_temp2 - mean2), img_temp2 - mean2)/ index_temp2.sum()
              mean3 = np.sum(img_temp3, axis=0)/index_temp3.sum()
              cov3 = np.dot(np.transpose(img_temp3 - mean3), img_temp3 - mean3)/ index_temp3.sum()
              mean4 = np.sum(img_temp4, axis=0)/index_temp4.sum()
              cov4 = np.dot(np.transpose(img_temp4 - mean4), img_temp4 - mean4)/ index_temp4.sum()
              mean5 = np.sum(img_temp5, axis=0)/index_temp5.sum()
              cov5 = np.dot(np.transpose(img_temp5 - mean5), img_temp5 - mean5)/ index_temp5.sum()
              mean6 = np.sum(img_temp6, axis=0)/index_temp6.sum()
              cov6 = np.dot(np.transpose(img_temp6 - mean6), img_temp6 - mean6)/ index_temp6.sum()
              mean7 = np.sum(img_temp7, axis=0)/index_temp7.sum()
```

```
cov7 = np.dot(np.transpose(img_temp7 - mean7), img_temp7 - mean7)/ index_temp7.sum()
mean8 = np.sum(img_temp8, axis=0)/index_temp8.sum()
cov8 = np.dot(np.transpose(img_temp8 - mean8), img_temp8 - mean8)/ index_temp8.sum()
mean9 = np.sum(img_temp9, axis=0)/index_temp9.sum()
cov9 = np.dot(np.transpose(img_temp9 - mean9), img_temp9 - mean9)/ index_temp9.sum()
cov_overall = (cov0 + cov1 + cov2 + cov3+ cov4 + cov5 + cov6 + cov7 + cov8 + cov9)/10
prior = np.array([index_temp0.sum(),index_temp1.sum(),index_temp2.sum(),index_temp3.sum()
     index_temp5.sum(),index_temp6.sum(),index_temp7.sum(), index_temp8.sum(), index_temp

inv = approximate_inv(cov_overall)



det = np.linalg.det(cov0) * np.linalg.det(cov1) * np.linalg.det(cov2) * np.linalg.det(cov

while det == 0:
    cov0 = cov0 + 0.001 * np.identity(784)
    cov1 = cov1 + 0.001 * np.identity(784)
    cov2 = cov2 + 0.001 * np.identity(784)
    cov3 = cov3 + 0.001 * np.identity(784)
    cov4 = cov4 + 0.001 * np.identity(784)
    cov5 = cov5 + 0.001 * np.identity(784)
    cov6 = cov6 + 0.001 * np.identity(784)
    cov7 = cov7 + 0.001 * np.identity(784)
    cov8 = cov8 + 0.001 * np.identity(784)
    cov9 = cov9 + 0.001 * np.identity(784)
    det = np.linalg.det(cov0) * np.linalg.det(cov1) * np.linalg.det(cov2) * np.linalg.det

inv0 = np.linalg.inv(cov0)
inv1 = np.linalg.inv(cov1)
inv2 = np.linalg.inv(cov2)
inv3 = np.linalg.inv(cov3)
inv4 = np.linalg.inv(cov4)
inv5 = np.linalg.inv(cov5)
inv6 = np.linalg.inv(cov6)
inv7 = np.linalg.inv(cov7)
inv8 = np.linalg.inv(cov8)
inv9 = np.linalg.inv(cov9)
det0 = np.linalg.det(inv0)
det1 = np.linalg.det(inv1)
det2 = np.linalg.det(inv2)
det3 = np.linalg.det(inv3)
det4 = np.linalg.det(inv4)
det5 = np.linalg.det(inv5)
det6 = np.linalg.det(inv6)
det7 = np.linalg.det(inv7)
det8 = np.linalg.det(inv8)
det9 = np.linalg.det(inv9)

def overall(x):

    prob0 = - 0.5 * np.dot(np.dot(x-mean0,inv), np.transpose(x-mean0)) + math.log(prior[0]
    prob1 = - 0.5 * np.dot(np.dot(x-mean1,inv), np.transpose(x-mean1)) + math.log(prior[1]
    prob2 = - 0.5 * np.dot(np.dot(x-mean2,inv), np.transpose(x-mean2)) + math.log(prior[2]
```

```
            prob3 = - 0.5 * np.dot(np.dot(x-mean3,inv), np.transpose(x-mean3)) + math.log(prior[3]
            prob4 = - 0.5 * np.dot(np.dot(x-mean4,inv), np.transpose(x-mean4)) + math.log(prior[4]
            prob5 = - 0.5 * np.dot(np.dot(x-mean5,inv), np.transpose(x-mean5)) + math.log(prior[5]
            prob6 = - 0.5 * np.dot(np.dot(x-mean6,inv), np.transpose(x-mean6)) + math.log(prior[6]
            prob7 = - 0.5 * np.dot(np.dot(x-mean7,inv), np.transpose(x-mean7)) + math.log(prior[7]
            prob8 = - 0.5 * np.dot(np.dot(x-mean8,inv), np.transpose(x-mean8)) + math.log(prior[8]
            prob9 = - 0.5 * np.dot(np.dot(x-mean9,inv), np.transpose(x-mean9)) + math.log(prior[9]

            prob = np.array([prob0, prob1, prob2, prob3, prob4, prob5, prob6, prob7, prob8, prob9]
            return np.argmax(prob)

        def qda(x):

            prob0 = 0.5 * math.log(det0) - 0.5 * np.dot(np.dot(x-mean0,inv0), np.transpose(x-mean0
            prob1 = 0.5 * math.log(det1) - 0.5 * np.dot(np.dot(x-mean1,inv1), np.transpose(x-mean1
            prob2 = 0.5 * math.log(det2) - 0.5 * np.dot(np.dot(x-mean2,inv2), np.transpose(x-mean2
            prob3 = 0.5 * math.log(det3) - 0.5 * np.dot(np.dot(x-mean3,inv3), np.transpose(x-mean3
            prob4 = 0.5 * math.log(det4) - 0.5 * np.dot(np.dot(x-mean4,inv4), np.transpose(x-mean4
            prob5 = 0.5 * math.log(det5) - 0.5 * np.dot(np.dot(x-mean5,inv5), np.transpose(x-mean5
            prob6 = 0.5 * math.log(det6) - 0.5 * np.dot(np.dot(x-mean6,inv6), np.transpose(x-mean6
            prob7 = 0.5 * math.log(det7) - 0.5 * np.dot(np.dot(x-mean7,inv7), np.transpose(x-mean7
            prob8 = 0.5 * math.log(det8) - 0.5 * np.dot(np.dot(x-mean8,inv8), np.transpose(x-mean8
            prob9 = 0.5 * math.log(det9) - 0.5 * np.dot(np.dot(x-mean9,inv9), np.transpose(x-mean9

            prob = np.array([prob0, prob1, prob2, prob3, prob4, prob5, prob6, prob7, prob8, prob9]
            return np.argmax(prob)
        return [overall, qda]
```

In [266]:
```
#predict labels on validation set by a model trained with 100 training data.

[overall100, qda100] = gaussian_classifier(100)
pred_img100 = np.apply_along_axis(overall100, 1, valid_img)
result100 = pred_img100 == valid_lb
result100.sum()
```

Out[266]: 6291

In [268]:
```
qpred_img100 = np.apply_along_axis(qda100, 1, valid_img)
qresult100 = qpred_img100 == valid_lb
qresult100.sum()
```

Out[268]: 7037

In [269]:
```
#predict labels on validation set by a model trained with 200 training data.

[overall200, qda200] = gaussian_classifier(200)
pred_img200 = np.apply_along_axis(overall200, 1, valid_img)
qpred_img200 = np.apply_along_axis(qda200, 1, valid_img)
result200 = pred_img200 == valid_lb
qresult200 = qpred_img200 == valid_lb
print(result200.sum())
print(qresult200.sum())

#predict labels on validation set by a model trained with 500 training data.
```

```
        [overall500, qda500] = gaussian_classifier(500)
        pred_img500 = np.apply_along_axis(overall500, 1, valid_img)
        qpred_img500 = np.apply_along_axis(qda500, 1, valid_img)
        result500 = pred_img500 == valid_lb
        qresult500 = qpred_img500 == valid_lb
        print(result500.sum())
        print(qresult500.sum())
```

```
[ 0.09   0.105  0.105  0.08   0.065  0.11   0.135  0.09   0.12   0.1  ]
[5.2388320033509897e+31, 1.5298618973490495e+35, 1.3606194697551303e+30, 5.2216474519147938e+32, 6.4793
7239
7900
[ 0.088  0.128  0.082  0.102  0.088  0.096  0.12   0.104  0.092  0.1  ]
[1.5641378502722663e+31, 3.0078162075045203e+36, 1.0168210879959122e+30, 4.0318380929693488e+31, 4.9595
7614
8265
```

In [271]:
```
#predict labels on validation set by a model trained with 1000 training data.

        [overall1000, qda1000] = gaussian_classifier(1000)
        pred_img1000 = np.apply_along_axis(overall1000, 1, valid_img)
        qpred_img1000 = np.apply_along_axis(qda1000, 1, valid_img)
        result1000 = pred_img1000 == valid_lb
        qresult1000 = qpred_img1000 == valid_lb
        print(result1000.sum())
        print(qresult1000.sum())

        #predict labels on validation set by a model trained with 2000 training data.

        [overall2000, qda2000] = gaussian_classifier(2000)
        pred_img2000 = np.apply_along_axis(overall2000, 1, valid_img)
        qpred_img2000 = np.apply_along_axis(qda2000, 1, valid_img)
        result2000 = pred_img2000 == valid_lb
        qresult2000 = qpred_img2000 == valid_lb
        print(result2000.sum())
        print(qresult2000.sum())
```

```
[ 0.086  0.113  0.094  0.113  0.091  0.105  0.111  0.099  0.091  0.097]
7891
8358
[ 0.0895  0.1125  0.095   0.1085  0.0925  0.1025  0.1015  0.1025  0.0915
  0.104 ]
8078
8368
```

In [273]:
```
#predict labels on validation set by a model trained with 5000 training data.

        [overall5000, qda5000] = gaussian_classifier(5000)
        pred_img5000 = np.apply_along_axis(overall5000, 1, valid_img)
        qpred_img5000 = np.apply_along_axis(qda5000, 1, valid_img)
        result5000 = pred_img5000 == valid_lb
        qresult5000 = qpred_img5000 == valid_lb
        print(result5000.sum())
```

```
print(qresult5000.sum())

#predict labels on validation set by a model trained with 10000 training data.

[overall10000, qda10000] = gaussian_classifier(10000)
pred_img10000 = np.apply_along_axis(overall10000, 1, valid_img)
qpred_img10000 = np.apply_along_axis(qda10000, 1, valid_img)
result10000 = pred_img10000 == valid_lb
qresult10000 = qpred_img10000 == valid_lb
print(result10000.sum())
print(qresult10000.sum())
```

```
8192
8419
8189
8422
```

In [276]:
```
#predict labels on validation set by a model trained with 3000 training data.

[overall30000, qda30000] = gaussian_classifier(30000)
pred_img30000 = np.apply_along_axis(overall30000, 1, valid_img)
qpred_img30000 = np.apply_along_axis(qda30000, 1, valid_img)
result30000 = pred_img30000 == valid_lb
qresult30000 = qpred_img30000 == valid_lb
print(result30000.sum())
print(qresult30000.sum())
```

```
8190
8425
```

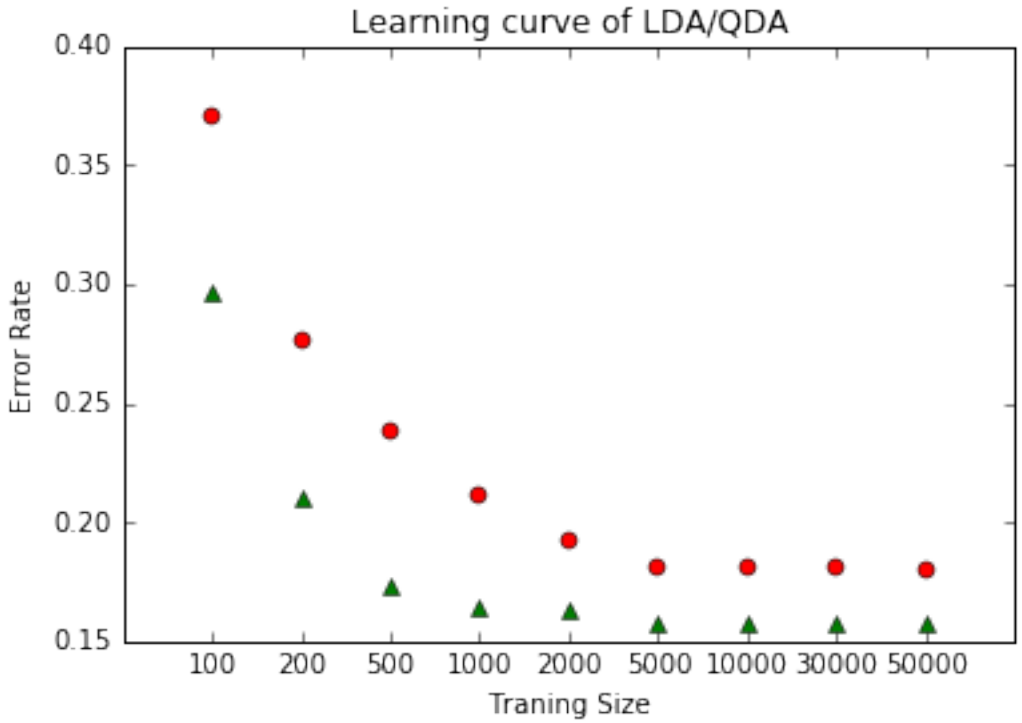
In [277]:
```
#predict labels on validation set by a model trained with 5000 training data.

[overall50000, qda50000] = gaussian_classifier(50000)
pred_img50000 = np.apply_along_axis(overall50000, 1, valid_img)
qpred_img50000 = np.apply_along_axis(qda50000, 1, valid_img)
result50000 = pred_img50000 == valid_lb
qresult50000 = qpred_img50000 == valid_lb
print(result50000.sum())
print(qresult50000.sum())
```

```
8198
8425
```

In [289]:
```
# error rates of LDA trained with different number of data.

error_rate = 1 - np.array([result100.sum(), result200.sum(), result500.sum(), result1000.sum()
                           result5000.sum(), result10000.sum(), result30000.sum(), result5000

# error rates of QDA trained with different number of data.

qerror_rate = 1 - np.array([qresult100.sum(), qresult200.sum(), qresult500.sum(), qresult1000
                            qresult2000.sum(), qresult5000.sum(), qresult10000.sum(),
                            qresult30000.sum(), qresult50000.sum()])/10000
```

i, ii) Learning curves for LDA/QDA

```
In [301]: import matplotlib.pyplot as plt
          xpos = np.arange(len(error_rate))+ 1
          plt.plot(xpos, error_rate, 'ro',xpos, qerror_rate, 'g^')

          xticklb = ['','100','200','500','1000','2000','5000','10000','30000','50000','']
          plt.axis = [0,11,0.15,0.40]
          plt.title('Learning curve of LDA/QDA')
          plt.xlabel('Traning Size')
          plt.ylabel('Error Rate')
          plt.xticks(np.arange(len(error_rate)+2), xticklb)
          plt.show()
```



i, ii, iii) QDA VS LDA Because a quadratic term in posterior probabilities for every class will be the same as $x\Sigma_{overall}x^\top$, this quadratic term will be cancelled out when comparing the posterior probabilities for different classes and finding the best probable class given a feauture vector x. Therefore, LDA gives a linear decision boundary. In contrast, since in QDA we use different covariance matrices for each class, the quadratic terms in the posterior probabilities will not be called. For this reason, QDA exhibits a quadratic form of decision boundary. Because we made an additional assumption that all covariance matrices are the same in LDA, LDA may not be able to reflect the reality of given data. From the learning curves, we can see that QDA results in lower error rates compared to LDA. Also, the more the number of traing samples are, the lower the error rate is in general.

iv) Kaggle Score We train the best classiffier, QDA with 50000 training samples, using train.mat and classify the images in test.mat. This achieved a kaggle score of 0.92960.

```
In [313]: qpred_test = np.apply_along_axis(qda50000, 1, test_img)

In [314]: qpred_txt = np.asarray([[i+1, qpred_test[i]] for i in np.arange(10000)])
          np.savetxt('qpred_test.csv',qpred_txt,fmt = '%1.u' , delimiter = ',', header = 'Id,Category',
```

3.1 e) Apply the Gaussian classifier to Spam Data

```
In [21]: data_spam = sio.loadmat('./data/spam_dataset/spam_data.mat')

In [22]: data_spam

Out[22]: {'__globals__': [],
          '__header__': b'MATLAB 5.0 MAT-file Platform: posix, Created on: Tue Jan 20 22:50:03 2015',
          '__version__': '1.0',
          'test_data': array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 ...,
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  3.,  0.,  0.]]),
          'training_data': array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 ...,
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  1., ...,  4.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  2.,  0.,  0.]]),
          'training_labels': array([[1, 1, 1, ..., 0, 0, 0]])}

In [63]: sptrain_ft = data_spam['training_data']
         sptrain_lb = np.transpose(data_spam['training_labels'])[...,0]
         sptest = data_spam['test_data']

In [64]: sp_sum = sptrain_ft.sum(1) + 0.000001
         sptrain_ft.shape
         #sp_sum[sp_sum>0, ].shape

Out[64]: (5172, 32)
```

3.1.1 Normalize the contrast

We normalize feature vectors, which is the count vectors, by dividing each vector by its L_1 norm, i.e., the total sum of its elements. This will result in vectors that represent the proportion of each word in documents.

```
In [65]: sptrain_ft = (sptrain_ft.T/ sp_sum).T

In [66]: def spapproximate_inv(M):
             temp = M
             while np.linalg.det(temp) == 0:
                     temp = temp + 0.0000000001 * np.identity(32)
             return np.linalg.inv(temp)


         spindex0 = sptrain_lb == 0
         spindex1 = sptrain_lb == 1
```

```
sp_ft0 = sptrain_ft[spindex0,]
sp_ft1 = sptrain_ft[spindex1,]

spmean0 = np.sum(sp_ft0, axis=0)/spindex0.sum()
spcov0 = np.dot(np.transpose(sp_ft0 - spmean0), sp_ft0 - spmean0)/ spindex0.sum()
spmean1 = np.sum(sp_ft1, axis=0)/spindex1.sum()
spcov1 = np.dot(np.transpose(sp_ft1 - spmean1),sp_ft1 - spmean1)/ spindex1.sum()

spcov_overall = (spcov0 + spcov1 )/2

spprior = np.array([spindex0.sum(),spindex1.sum()])/ 5172

spinv = spapproximate_inv(spcov_overall)



spdet = np.linalg.det(spcov0) * np.linalg.det(spcov1)
while spdet == 0:
    spcov0 = spcov0 + 0.0000000001 * np.identity(32)
    spcov1 = spcov1 + 0.0000000001 * np.identity(32)
    spdet = np.linalg.det(spcov0) * np.linalg.det(spcov1)
spinv0 = np.linalg.inv(spcov0)
spinv1 = np.linalg.inv(spcov1)
spdet0 = np.linalg.det(spinv0)
spdet1 = np.linalg.det(spinv1)

def sp_lda(x):

    prob0 = - 0.5 * np.dot(np.dot(x-spmean0,spinv), np.transpose(x-spmean0)) + math.log(spprio
    prob1 = - 0.5 * np.dot(np.dot(x-spmean1,spinv), np.transpose(x-spmean1)) + math.log(spprio


    prob = np.array([prob0, prob1])
    return np.argmax(prob)

def sp_qda(x):

    prob0 = 0.5 * math.log(spdet0) - 0.5 * np.dot(np.dot(x-spmean0,spinv0), np.transpose(x-spm
    prob1 = 0.5 * math.log(spdet1) - 0.5 * np.dot(np.dot(x-spmean1,spinv1), np.transpose(x-spm

    prob = np.array([prob0, prob1])
    return np.argmax(prob)
```

In [67]:
```
sppred_lda = np.apply_along_axis(sp_lda, 1, sptrain_ft )
```

In [71]:
```
sppred_qda = np.apply_along_axis(sp_qda, 1, sptrain_ft )
(sppred_lda == sptrain_lb).sum()
```

Out[71]: 4234

In [69]:
```
sptest_sum = sptest.sum(1)+ 0.000001
sptest = np.divide(sptest, sptest_sum[:, None])
sppred = np.apply_along_axis(sp_qda, 1, sptest)
```

In [70]:
```
sppred_test = np.asarray([[i+1, sppred[i]] for i in np.arange(5857)])
np.savetxt('sppred.csv', sppred_test, fmt = '%1.u' , delimiter = ',', header = 'Id,Category',c
```

Kaggle score We achieved a kaggle score of 0.73387 by using QDA trained with all 5857 data.

4 Problem 6

In this problem we will try to predict the median home value in a given Census area by using linear regression. There are only 8 features for each data point.

In [1]:
```
# import packages, if necessary

import scipy.io as sio
import numpy as np
```

In [2]:
```
# import data

housing_data = sio.loadmat('./data/housing_dataset/housing_data.mat')
```

In [3]:
```
# extracting training feature and labels, and validating feature and labels.
# For features, we add another dimension to each data point, with the value of 1.

htrain_ft = np.insert(housing_data['Xtrain'], 8, 1, axis =1)
hvalid_ft = np.insert(housing_data['Xvalidate'], 8 , 1, axis = 1)
htrain_hv = housing_data['Ytrain'][..., 0]
hvalid_hv = housing_data['Yvalidate'][..., 0]
```

We consider linear regression model $Y = X^\top \beta$. It is when $\beta = (X^\top X)^{-1} X^\top$ that the linear regression model exhibits the least squaure error.

In [6]:
```
# calculate the coefficient matrix beta

beta = np.dot(np.dot(np.linalg.inv(np.dot(htrain_ft.T, htrain_ft)), htrain_ft.T), htrain_hv)
```

We test the trained model with the coefficient matrix beta on the validation set.

In [7]:
```
pred_hv = np.dot(hvalid_ft,beta)
```

4.0.1 Min and Max of predicted values

The minimum of predicted values is -56562.827543101739, and the maximum is 710798.83868835587. These are not reasonable since home value cannot be of a negative value. Compared to teh actual range of home value, the maximum is too high.

In [8]:
```
# min of predicted values

min(pred_hv)
```

Out[8]:
```
-56562.827543101739
```

In [9]:
```
# max of predicted values

max(pred_hv)
```

Out[9]:
```
710798.83868835587
```

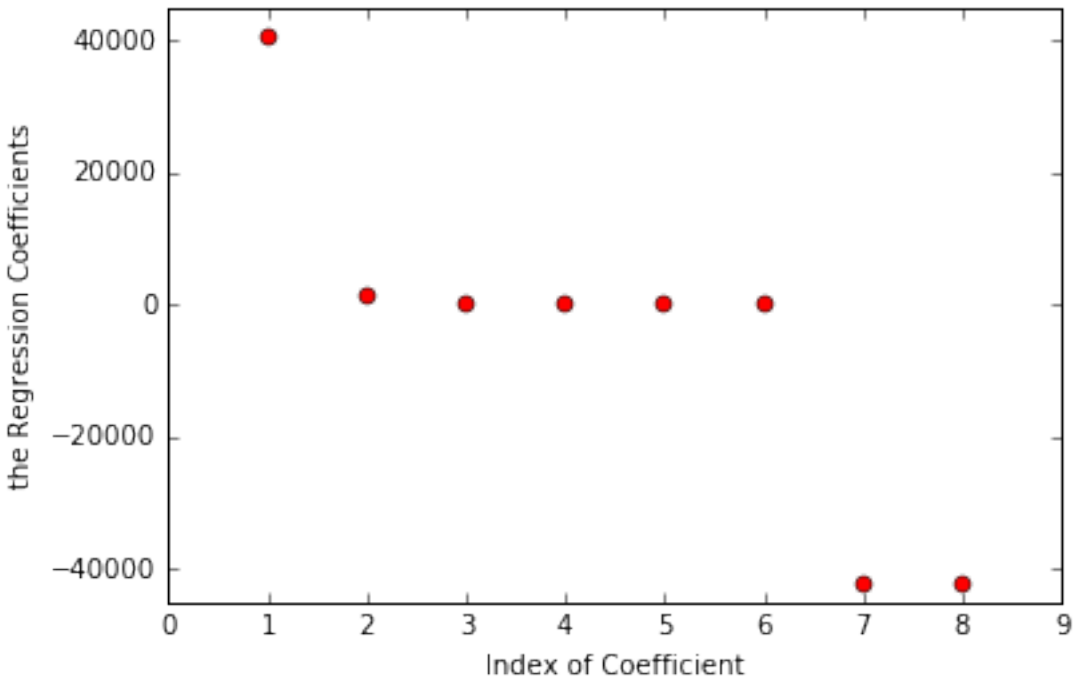
4.0.2 Residual Sum of Square (RSS)

The residual sum of squares (RSS) is 5794953797654.9834, which is huge considering our home value range.

```
In [10]: np.dot((hvalid_hv - pred_hv).T, (hvalid_hv - pred_hv))

Out[10]: 5794953797654.9834

In [11]: beta[0:8]

Out[11]: array([  4.05879986e+04,   1.19561189e+03,  -8.50145688e+00,
                  1.18352188e+02,  -3.77900280e+01,   4.30562637e+01,
                 -4.21794075e+04,  -4.24573474e+04])
```

4.0.3 Regression coefficients

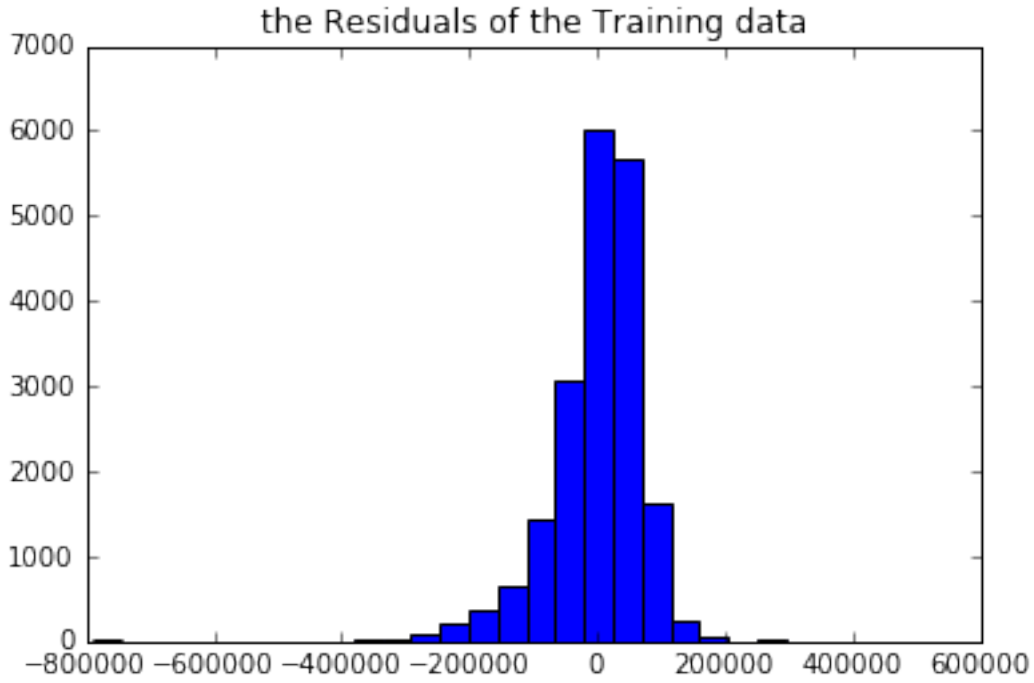
```
In [ ]: import matplotlib.pyplot as plt
        %matplotlib inline

In [15]: xpos = np.arange(len(beta[0:8]))+ 1
         plt.axis([0,9, -45000, 45000])
         plt.plot(xpos, beta[0:8], 'ro')
         plt.xlabel('Index of Coefficient')
         plt.ylabel('the Regression Coefficients')
         plt.show()
```



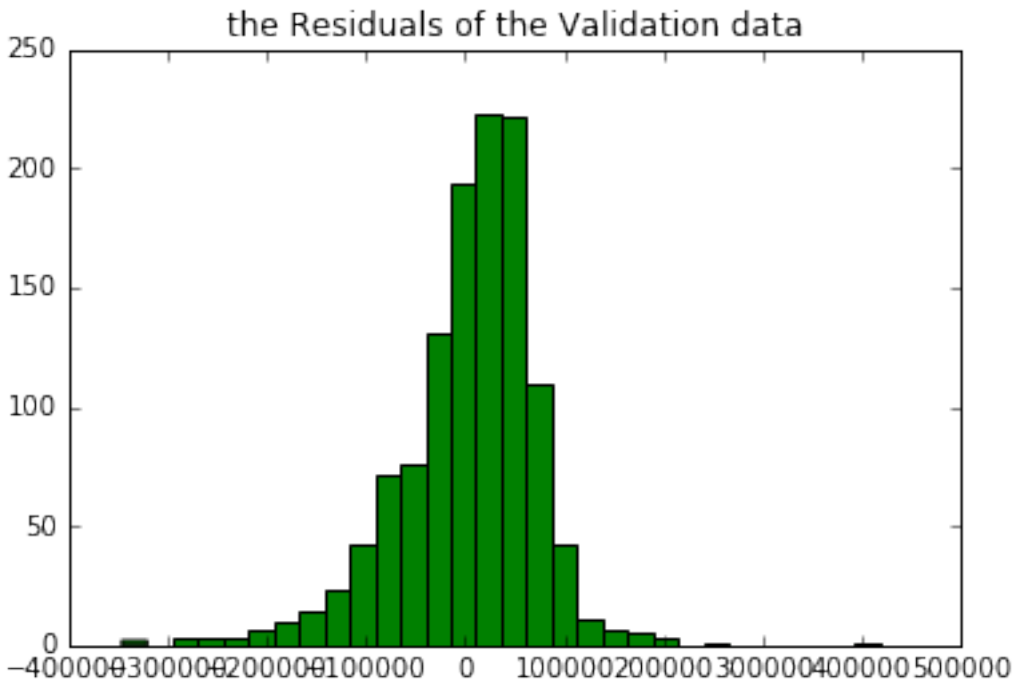
4.0.4 Histogram of the Residuals of the Training data

Histogram of the Residuals of the Training data is bell-curve shaped, symmetric around 0, and this resembles the probability density curve of a Normal distribution with mean 0.

In [25]:
```
plt.hist(np.dot(htrain_ft,beta) - htrain_hv, bins = 30)
plt.title('the Residuals of the Training data')
plt.show()
```



In [29]:
```
plt.hist(pred_hv - hvalid_hv, bins = 30, color = 'green')
plt.title('the Residuals of the Validation data')
plt.show()
```


the Residuals of the Validation data
250
200
150
100
50
0
−400000
−300000
−200000
−100000
0
100000
200000
300000
400000
500000