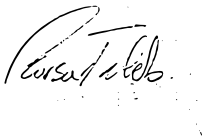
Introduction to Machine Learning

HW3

Parsa Taleb ¶

3032263716

I worked on this homework alone, and I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."



1 Gaussian Classification

a)

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

Gaussian Discriminant Analysis

Fundamental assumption: each class comes from normal distribution (Gaussian).

$X \sim \mathcal{N}\left(\mu, \sigma^2\right) : P(x) = \frac{1}{(\sqrt{2\pi}\sigma)^d} \exp\left(-\frac{|x-\mu|^2}{2\sigma^2}\right)$

$\mu \& x =$ vectors; $\sigma =$ scalar; $d =$ dimension

For each class C, suppose we estimate mean μ_{C}, variance σ_{C}^2, and prior $\pi_{\mathrm{C}} = P(Y = \mathrm{C})$

Given x, Bayes decision rule $r^*(x)$ returns class C that maximizes $P(X = x|Y = \mathrm{C})\pi_{\mathrm{C}}$

$\ln \omega$ is monotonically increasing for $\omega > 0$, so it is equivalent to maximize

$$Q_{\mathrm{C}}(x) = \ln\left((\sqrt{2\pi})^d P(x)\pi_{\mathrm{C}}\right) = -\frac{|x-\mu_{\mathrm{C}}|^2}{2\sigma_{\mathrm{C}}^2} - d\ln\sigma_{\mathrm{C}} + \ln\pi_{\mathrm{C}}$$

Quadratic Discriminant Analysis (QDA)

Suppose only 2 classes C, D. Then

$$r^*(x) = \begin{cases} C & \text{if } Q_{\mathrm{C}}(x) - Q_{\mathrm{D}}(x) > 0 \\ D & \text{otherwise} \end{cases}$$

Decision fn is quadratic in x . Bayes decision boundary is $Q_{\mathrm{C}}(x) - Q_{\mathrm{D}}(x) = 0$

To recover posterior probabilities in 2 -class case, use Bayes:

$$P(Y = \mathrm{C}|X) = \frac{P(X|Y=\mathrm{C})\pi_{\mathrm{C}}}{P(X|Y=\mathrm{C})\pi_{\mathrm{C}} + P(X|Y=\mathrm{D})\pi_{\mathrm{D}}}$$

$e^{Q_{\mathrm{C}}(x)} = (\sqrt{2\pi})^d P(x)\pi_{\mathrm{C}}$ [by definition of Q_{C}]

$$P(Y = \mathrm{C}|X = x) = \frac{e^{Q_{\mathrm{C}}(x)}}{e^{Q_{\mathrm{C}}(x)} + e^{Q_{\mathrm{D}}(x)}} = \frac{1}{1 + e^{Q_{\mathrm{D}}(x) - Q_{\mathrm{C}}(x)}}$$

$= s\left(Q_{\mathrm{C}}(x) - Q_{\mathrm{D}}(x)\right)$ ", where

$s(\gamma) = \frac{1}{1+e^{-\gamma}}$

Linear Discriminant Analysis (LDA)

$$Q_{\mathrm{C}}(x) - Q_{\mathrm{D}}(x) = \frac{\left(\mu_{\mathrm{C}} - \mu_{\mathrm{D}}\right)\cdot x}{\sigma^2} - \frac{|\mu_{\mathrm{C}}|^2 - |\mu_{\mathrm{D}}|^2}{2\sigma^2} + \ln\pi_{\mathrm{C}} - \ln\pi_{\mathrm{D}}$$

$= wx + \alpha$

We have a linear classifier. We choose C that maximizes linear discriminant function.

$$\frac{\mu_{\mathrm{C}}\cdot x}{\sigma^2} - \frac{|\mu_{\mathrm{C}}|^2}{2\sigma^2} + \ln\pi_{\mathrm{C}}$$

In 2 -class case: decision boundary is $w \cdot x + \alpha = 0$

Bayes posterior is $P(Y = \mathrm{C}|X = x) = s(w \cdot x + \alpha)$

If $\pi_{\mathrm{C}} = \pi_{\mathrm{D}} = \frac{1}{2} \Rightarrow (\mu_{\mathrm{C}} - \mu_{\mathrm{D}}) \cdot x - (\mu_{\mathrm{C}} - \mu_{\mathrm{D}}) \cdot \left(\frac{\mu_{\mathrm{C}} + \mu_{\mathrm{D}}}{2}\right) = 0$

which is the centroid method.

b)

The Bayes error is the probability of misclassification:

$P_e = P(($ misclassified as $C_1)\,|C_2)\,P(C_2) + P($ (misclassified as $C_2)\,|C_1)P(C_1)$

$X \sim \mathcal{N}(\mu, \sigma^2) : P(x) = \frac{1}{(\sqrt{2\pi}\sigma)^d} \exp\left(-\frac{|x-\mu|^2}{2\sigma^2}\right)$

Let, by centroid method, $a = \frac{\mu_2 - \mu_1}{2\sigma}$ and $d = 1$

$\Rightarrow P_e = \frac{1}{2}\int_{-\infty}^{a} P($ misclassified as $C_1)\,|C_2)(x)dx + \frac{1}{2}\int_{a}^{\infty} P($ misclassified as $C_2)\,|C_1)(x)dx$

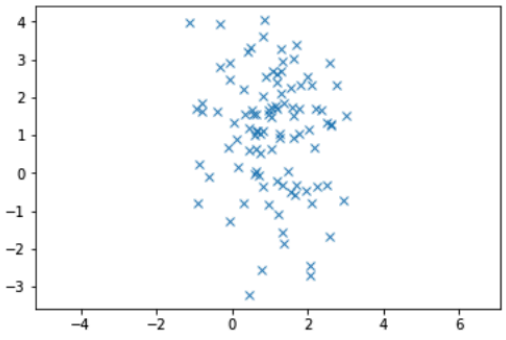
Let $z = \frac{|x-\mu_c|}{\sigma} \rightarrow dz = \frac{|dx|}{\sigma} \rightarrow dx = \sigma dz$

$\Rightarrow P_e = \frac{1}{\sqrt{2\pi}}\int_{a}^{\infty} e^{-z^2/2} dz$

2 Isocontours of Normal Distributions ¶

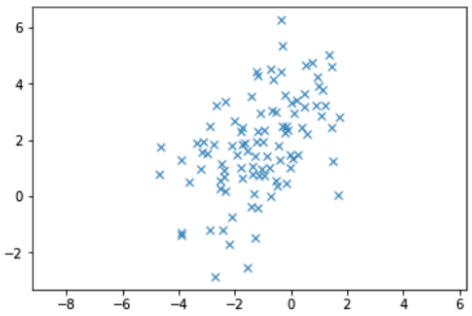
a)

```
import numpy as np
import matplotlib.pyplot as plt
miu = [1,1]
sigma = np.matrix('1 0; 0 2')
x, y = np.random.multivariate_normal(miu, sigma, 100).T
plt.plot(x, y, 'x')
plt.axis('equal')
plt.show()
```



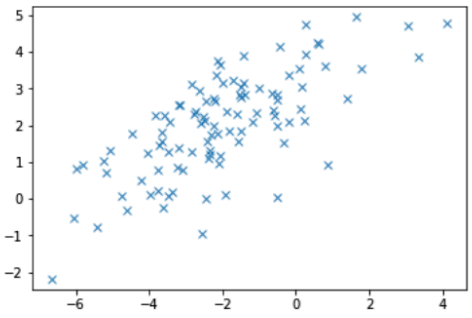
b)

In [2]:
```
miu = [-1,2]
sigma = np.matrix('2 1; 1 3')
x, y = np.random.multivariate_normal(miu, sigma, 100).T
plt.plot(x, y, 'x')
plt.axis('equal')
plt.show()
```



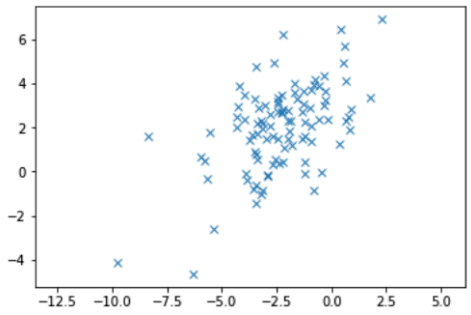
c)

In [3]:
```
miu1 = [0,2]
miu2 = [2,0]
sigma1 = np.matrix('2 1; 1 1')
sigma2 = sigma1
x1,y1 = np.random.multivariate_normal(miu1, sigma1, 100).T
x2,y2 = np.random.multivariate_normal(miu2, sigma2, 100).T
x3 = x1-x2
y3 = y1-y2
plt.plot(x3,y3, 'x')
plt.axis('equal')
plt.show()
```



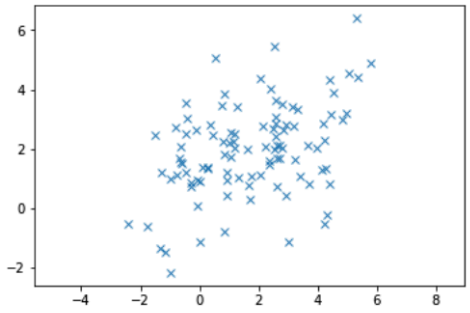
d)

In [4]:
```
miu1 = [0,2]
miu2 = [2,0]
sigma1 = np.matrix('2 1; 1 1')
sigma2 = np.matrix('2 1; 1 3')
x1,y1 = np.random.multivariate_normal(miu1, sigma1, 100).T
x2,y2 = np.random.multivariate_normal(miu2, sigma2, 100).T
x3 = x1-x2
y3 = y1-y2
plt.plot(x3,y3, 'x')
plt.axis('equal')
plt.show()
```



e)

In [5]:
```
miu1 = [1,1]
miu2 = [-1,-1]
sigma1 = np.matrix('2 0; 0 1')
sigma2 = np.matrix('2 1; 1 2')
x1,y1 = np.random.multivariate_normal(miu1, sigma1, 100).T
x2,y2 = np.random.multivariate_normal(miu2, sigma2, 100).T
x3 = x1-x2
y3 = y1-y2
plt.plot(x3,y3, 'x')
plt.axis('equal')
plt.show()
```



3 Eigenvectors of the Gaussian Covariance Matrix

(a)

In [6]:
```
import numpy as np
np.random.seed(10)
#np.random.RandomState(10)
X1= 3 + 9* np.random.randn(100,1)
X2= 0.5*X1 + (4 + 4*np.random.randn(100,1))
Z=[X1, X2]
#print(X1.T)
#print(len(Z))
#print(np.shape(Z))
Z = np.array(Z)[:,:,0]
#print(np.shape(Z))
mean_Z=np.array([np.mean(X1),np.mean(X2)])
print('The mean in R^2 of the sample is')
print(mean_Z)
```

```
The mean in R^2 of the sample is
[3.71474997 6.13416372]
```

(b)

In [7]:
```
cov_Z=np.cov(Z)
print('The 2x2 covariance Matrix of the sample is')
print(cov_Z)
```

```
The 2x2 covariance Matrix of the sample is
[[76.51352866 40.51160968]
 [40.51160968 37.07429541]]
```

(c)

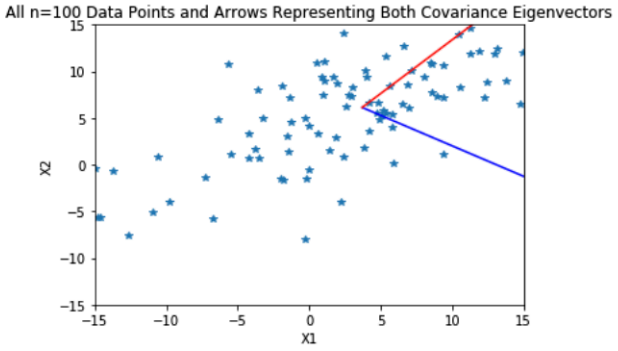
In [8]:
```
from numpy import linalg as LA
[eigv,eigvec]=LA.eig(cov_Z)
print('The eigenvectors of this covariance matrix are:')
print(eigvec)
print('The eigenvalues of this covariance matrix are:')
print(eigv)
```

```
The eigenvectors of this covariance matrix are:
[[ 0.84784076 -0.53025093]
 [ 0.53025093  0.84784076]]
The eigenvalues of this covariance matrix are:
[101.85003036  11.73779371]
```

(d)

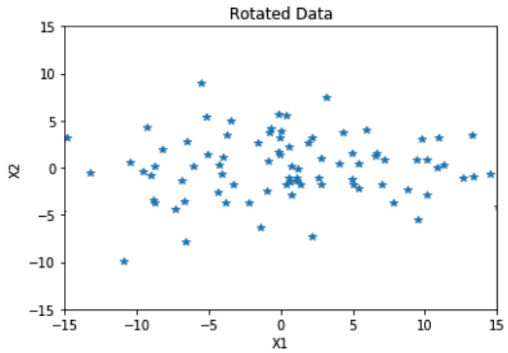
In [9]:
```
#(i)
plt.plot(X1, X2, '*')
plt.xlim([-15, 15])
plt.ylim([-15, 15])
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('All n=100 Data Points and Arrows Representing Both Covariance Eigenvectors')

#(ii)
n=mean_Z[0]+ eigv[0]*eigvec[0,:].T
n2=mean_Z[1]+ eigv[1]*eigvec[1,:].T
d=np.array([mean_Z,n]).T
d2=np.array([mean_Z,n2]).T
plt.plot(d[0,:],d[1,:],'b-')
plt.plot(d2[0,:],d2[1,:],'r-')
plt.show()
```



(e)

In [10]:
```
U=np.array([eigvec[0,:], eigvec[1,:]])
x_mu = np.array([(Z[0,:]-mean_Z[0]),(Z[1,:]-mean_Z[1])])
UT = U.T
x_rotated =np.dot(UT,x_mu)
plt.plot(x_rotated[0,:],x_rotated[1,:],'*')
plt.xlim([-15, 15])
plt.ylim([-15, 15])
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Rotated Data')
plt.show()
```



4 Classification

(a)

(1)

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

Let $r:\mathbb{R}^d\rightarrow\{1,\dots,c+1\}$ be a decision rule, we know this is a classification problem with classes labeled 1,...,c and an additional doubt category labeled c + 1.

$\lambda_r \leq \lambda_s$

$\lambda_r \geq 0$

$\lambda_s \geq 0$

$i\in\{1,2,\dots,c+1\}$

Average risk to obtain the minimum risk by is choosing class $r(x)=i$ such that:

$P(Y=i|x)\geq P(Y=j|x)$ for all j

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

$\quad = 0^{*}P\left(r(X)=i|\mathbf{x}\right)+\sum_{j=1,j!=i}^{c}\lambda_s P\left(Y=j|\mathbf{x}\right)$

Here $\lambda\left(r(x)=i|y=j\right)$ is the cost of choosing class $r(x)=i$ where the true class is $y=j$.

$\Rightarrow R\left(r(x)=i|\mathbf{x}\right)=\lambda_s\left(1-P\left(r(x)=i|\mathbf{x}\right)\right)$

If highest posterior class probability and the average risk is less than the cost of rejection, $\mathbf{x}$ is associated with the class $r(x)=i$, the policy obtains the minimum risk when $\lambda_r\leq\lambda_s$

$\Rightarrow \lambda_s\left(1-P\left(r(x)=i|\mathbf{x}\right)\right)\leq\lambda_r$

$\Rightarrow P\left(r(x)=i|\mathbf{x}\right)\geq 1-\lambda_r/\lambda_s$

(2)

Originally the additional doubt category corresponds to:

$L(r(x) = i = c + 1, y = j) = \lambda_r$

with $\lambda_r \leq \lambda_s$

where $\lambda_r \geq 0$ is the loss incurred for choosing doubt

Choosing the doubt othrewise, for instance by setting

$\lambda_r = \lambda_s \forall\ i \neq j \quad \& \quad i, j \in \{1, \ldots, c\}$

also obtains the minimum risk R, because the first term which is dependent of λ_r becomes 0 and does not contribute to risk summation:

$$\Rightarrow P(r(x) = i|\mathbf{x}) \geq 0$$

$$\begin{aligned} \Rightarrow R(r(x) = i + 1|x) &= \sum_{j=1}^{c} L(r(x) = i + 1, y = j) P(Y = j|x) \\ &= 0^* P(r(X) = i = c + 1|\mathbf{x}) + \sum_{j=1, j!=i}^{c} \lambda_s P(Y = j|\mathbf{x}) \\ &= \sum_{j=1, j!=(c+1)}^{c} \lambda_s P(Y = j|\mathbf{x}) \end{aligned}$$

(b)

If the value of λ_r is zero, there will be essentially no doubt category and the loss functions becomes:

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

Hence, intuitevly, one would expect to observe 0-1 loss function like behavior

If $\lambda_r > \lambda_s \Rightarrow \lambda_r = 0 > \lambda_s \Rightarrow \lambda_s < 0$

We end up with a negative loss function, which is counterintutive to the purpose

$textofdefiningnon - negative$ " $loss$ " $functions$.

Additionally, the risk R becomes summation of negative values, and thus R is negative, and negative risk does not make sense.

5 Maximum Likelihood Estimation

(a)

Let $x_1, \ldots, x_n \in \mathbb{R}^d$ be n samples points drawn independently from a multivariate normal distribution $\mathcal{N}(\mu, \Sigma)$

$$\sum = \begin{bmatrix} \sigma_1^2 & . & . & . & . & . & . \\ \ldots & \sigma_2^2 & & & \ldots & & \\ . & . & & & . & & \\ . & . & & & . & & \\ . & . & & & . & & \\ \ldots & \ldots & & & \sigma_n^2 & & \end{bmatrix}$$

Likelihood of generating these points is

$\mathcal{L}(\mu, \sigma; X_1, \ldots, X_n) = P(X_1) P(X_2) \cdots P(X_n)$

$\ell(\mu, \sigma; X_1, \ldots, X_n) = \ln P(X_1) + \ln P(X_2) + \ldots + \ln P(X_n)$

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

$\nabla_\mu \ell = 0, \frac{\partial \ell}{\partial \sigma} = 0$

$\nabla_\mu \ell = \sum_{i=1}^{n} \frac{X_i - \mu}{\sigma^2} = 0 \quad \Rightarrow \quad \hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} X_i$

$\frac{\partial \ell}{\partial \sigma} = \sum_{i=1}^{n} \frac{|X_i - \mu|^2 - d\sigma^2}{\sigma^3} = 0 \quad \Rightarrow \quad \hat{\sigma}_i^2 = \frac{1}{dn} \sum_{i=1}^{n} |X_i - \mu|^2$

$\hat{\sigma}_i^2 = \frac{1}{dn} \sum_{\mathrm{C}} \sum_{\{i : y_i = \mathrm{C}\}} |X_i - \hat{\mu}_{\mathrm{C}}|^2$

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

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

We can compute the upper quantities $\hat{\mu}$ and $\hat{\sigma}_i$ without knowing $\sum$ and μ.

(b)

Let's call this unknown mean $\mu' = A\mu$

Because A is invertible and we know its value,

$\rightarrow \mu = A^{-1}\mu'$

We know the value of the positive-definite covariance matrix $\sum$

$\boldsymbol{\mu} \in \mathbb{R}^d$ and positive-definite covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{d\times d}$, with density

$p(\mathbf{x};\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^d \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$

Since $\boldsymbol{\Sigma}$ is by derivation a symmetric, and by given information a positive definite matrix,

we can decompose it by the spectral theorem into

$\boldsymbol{\Sigma} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^T$, where the columns of $\mathbf{V}$ form an orthonormal basis in $\mathbb{R}^d$,

and $\boldsymbol{\Lambda}$ is a diagonal matrix with real, non-negative values. The entries of Λ dictate how elongated or shrunk the distribution is along each direction.

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

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

$\hat{\Sigma} = \frac{1}{n}\sum_{i=1}^{n}\left(X_i - A\hat{\mu}\right)\left(X_i - A\hat{\mu}\right)^\top \approx \sum(\text{known quantity})$

In the above equation, we know all the quantities

except $\hat{\mu}$ which we can compute from it. Alternative derivation:

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{n_k}\sum_{t_i=k}\mathbf{x}_i$$

$$\hat{\boldsymbol{\Sigma}}_k = \frac{1}{n_k}\sum_{t_i=k}\left(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k\right)\left(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k\right)^T$$

$$\hat{\mu_k}, \hat{\sigma_k}^2 = \arg\max_{\mu_k,\sigma_k^2} P\left(x_1, x_2, \ldots, x_{n_k}|\mu_k,\sigma_k^2\right)$$

$$= \arg\max_{\mu_k,\sigma_k^2} \ln\left(P\left(x_1, x_2, \ldots, x_{n_k}|\mu_k,\sigma_k^2\right)\right)$$

$$= \arg\max_{\mu_k,\sigma_k^2} \sum_{i=1}^{n_k} \ln\left(P\left(x_i|\mu_k,\sigma_k^2\right)\right)$$

$$= \arg\max_{\mu_k,\sigma_k^2} \sum_{i=1}^{n_k} -\frac{\left(x_i-\mu_k\right)^2}{2\sigma_k^2} - \ln(\sigma_k) - \frac{1}{2}\ln(2\pi)$$

$$= \arg\min_{\mu_k,\sigma_k^2} \sum_{i=1}^{n_k} \frac{\left(x_i-\mu_k\right)^2}{2\sigma_k^2} + \ln(\sigma_k)$$

$$\min_{\mu_k,\sigma_k^2} \sum_{i=1}^{n_k} \frac{\left(x_i-\mu_k\right)^2}{2\sigma_k^2} + \ln(\sigma_k) = \min_{\sigma_k^2} \min_{\mu_k} \sum_{i=1}^{n_k} \frac{\left(x_i-\mu_k\right)^2}{2\sigma_k^2} + \ln(\sigma_k)$$

$$\frac{\partial}{\partial\mu_k}\left(\sum_{i=1}^{n_k} \frac{\left(x_i-\mu_k\right)^2}{2\sigma_k^2} + \ln(\sigma_k)\right) = \sum_{i=1}^{n_k} \frac{-\left(x_i-\mu_k\right)}{\sigma_k^2} = 0 \Longrightarrow \hat{\mu}_k = \frac{1}{n_k}\sum_{i=1}^{n_k} x_i$$

6 Covariance Matrices and Decompositions

a)

The covariance matrix $\text{Var}(R) \in \mathbb{R}^{d \times d}$ for a random variable $R \in \mathbb{R}^d$ with mean μ is

$\text{Var}(R) = \text{Cov}(R, R) = \mathbb{E}\left[(R-\mu)(R-\mu)^\top\right]$

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

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

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

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

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

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

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

When $n < p$ (the number of observations is less than the number of random variables)

the empirical estimate of the covariance matrix becomes singular,

i.e. it cannot be inverted to compute the precision matrix. Recall

$\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$

Consider the $n \times p$ matrix of sample data, y. he rank of y is at most min(n,p).

the rank of $\sum$ should not be larger than the rank y

If $n < p \rightarrow \text{rank}(y) < p$ in which case rank $(\sum) < p$

In addition, clearly the covariance matrix need to be full-rank composed of linearly independent vectors

for it to have an inverse. It must have no zero eigenvalue and a non-zero determinant.

Moreover, in case of diagonal covariance matrix,

note that as long as no σ is 0, $\mathbf{\Sigma}$ is invertible.

$$\sum = \begin{bmatrix} \sigma_1^2 & . & . & . & . & . & . \\ \ldots & \sigma_2^2 & & & \ldots & & \\ . & . & & & . & & \\ . & . & & & . & & \\ . & . & & & . & & \\ \ldots & \ldots & & & \sigma_n^2 & & \end{bmatrix}$$

b)

J.H. Friedman,Regularized discriminant analysis, J. Amer. Statist. Assoc. 84(405) (1989), pp. 165–175.

Friedman has presented an alternative approach to covariance-matrix regularization in small-sample, high-dimensional supervised classification problems with K classes, where the covariance matrices are not assumed to be equal. To estimate the kth class covariance matrix, he has proposed

$$\hat{\boldsymbol{\Sigma}}_k(\lambda,\gamma) = (1-\gamma)\hat{\boldsymbol{\Sigma}}_k(\lambda) + \frac{\gamma}{p}\left[\mathrm{tr}\left\{\hat{\boldsymbol{\Sigma}}_k(\lambda)\right\}\mathbf{I}_p\right.$$

$$\hat{\boldsymbol{\Sigma}}_k(\lambda) = \frac{[(1-\lambda)\mathbf{S}_k+\lambda\mathbf{S}]}{[(1-\lambda)n_k+\lambda N]}$$

$$\mathbf{S}_k = \sum_{i=1}^{n_k}\left(\mathbf{x}_{ik}-\bar{\mathbf{x}}_k\right)\left(\mathbf{x}_{ik}-\bar{\mathbf{x}}_k\right)', \mathbf{S} = \sum_{k=1}^{K}\mathbf{S}_k, 0 \le \lambda, \gamma \le 1$$

For $\lambda = 1$, $\hat{\boldsymbol{\Sigma}}_k(\lambda)$ reduces to $\hat{\boldsymbol{\Sigma}}$, the maximum-likelihood estimator of $\boldsymbol{\Sigma}$, with

$$\hat{\boldsymbol{\Sigma}} = \frac{N-K}{N}\mathbf{S}_p$$

$$\mathbf{S}_{\mathrm{RDA}} \equiv (1-\gamma)\hat{\boldsymbol{\Sigma}} + \frac{\gamma}{p}[\mathrm{tr}(\hat{\boldsymbol{\Sigma}})]\mathbf{I}_p$$

$$\hat{G}_k(\mathbf{x}) = \left(\mathbf{x}-\bar{\mathbf{x}}_k\right)'\mathbf{S}_p^{-1}\left(\mathbf{x}-\bar{\mathbf{x}}_k\right) - 2\ln\hat{\pi}_k$$

$$\bar{\mathbf{x}}_k = \frac{1}{n_k}\sum_{i=1}^{n_k}\mathbf{x}_{ik}$$

$$\mathbf{S}_p = \frac{1}{N-K}\sum_{k=1}^{K}\sum_{i=1}^{n_k}\left(\mathbf{x}_{ik}-\bar{\mathbf{x}}_k\right)\left(\mathbf{x}_{ik}-\bar{\mathbf{x}}_k\right)'$$

c)

We have a normal distribution $\mathcal{N}(0, \Sigma)$ with mean $\mu = 0$

We are considering all vectors of length 1; i.e., any vector x for which $|x| = 1$

The data are generated by a multivariate Gaussian distribution:

$x_i \overset{\text{iid}}{\sim} \mathcal{N}(\mu = 0, \Sigma)$

Then the maximum likelihood estimate of the covariance matrix Σ is

$\hat{\Sigma} = \frac{1}{n} \sum_{i=1}^{n} \left(x_i - \bar{x}\right) \left(x_i - \bar{x}\right)^\top = \frac{1}{n} X^\top X$

where $\bar{x}$ is the sample average and the matrix X is assumed to be zero-mean as before. The eigenvectors of $\hat{\Sigma}$ and $X^\top X$ are the same since they are positive scalar multiples of each other.

The contours of the multivariate Gaussian density form ellipsoids. The direction of largest variance (i.e. the first principal component) is the eigenvector corresponding to the smallest eigenvalue of Σ^{-1}, which is the largest eigenvalue of Σ. We do not know Σ in general, so we use $\hat{\Sigma}$ in its place. Thus the principal component is an eigenvector corresponding to the largest eigenvalue of $\hat{\Sigma}$. As mentioned earlier, this matrix, has the same eigenvalues and eigenvectors as $X^\top X$, so we arrive at the same solution.

Ordinary least squares minimizes the vertical distance between the fitted line and the data points:

$\|y - Xu\|^2$

We show that PCA can be interpreted as minimizing the perpendicular distance between the principal component subspace and the data points, so in this sense it is doing the same thing as total least squares.

The orthogonal projection of a vector x onto the subspace spanned by a unit vector u equals u scaled by the scalar projection of x onto u :

$P_u x = \left(uu^\top\right) x = \left(x^\top u\right) u$

Suppose we want to minimize the total reconstruction error:

$\epsilon_{PCA\_Err}(u) = \sum_{i=1}^{n} \|x_i - P_u x_i\|^2$

$= \sum_{i=1}^{n} \left(\|x_i\|^2 - \|P_u x_i\|^2\right)$

$= \sum_{i=1}^{n} \|x_i\|^2 - \sum_{i=1}^{n} \left\|\left(x_i^\top u\right) u\right\|^2$

$= \sum_{i=1}^{n} \|x_i\|^2 - \sum_{i=1}^{n} \left(x_i^\top u\right)^2$

where it holds by the Pythagorean theorem

$\|x - P_u x\|^2 + \|P_u x\|^2 = \|x\|^2$

since $x - P_u x \perp P_u x$. Then since the first term $\sum_i \|x_i\|^2$ is constant with respect to u, we have

$\arg\min_u \epsilon_{PCA-Err}(u) = \arg\min_u \text{constant} - \epsilon_{PCA\_Var}(u) = \arg\max_u \epsilon_{PCA-Var}(u)$

Hence minimizing reconstruction error is equivalent to maximizing projected variance.

7 Gaussian Classifiers for Digits and Spam

(a)

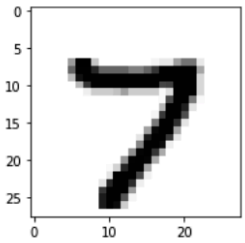
In [16]:
```
data_mnist = comp("mnist", val_size=100)
data_mnist.MLE()
```

```
loaded mnist dataset!
Class is initialized!
Dataset splitted!
0
1
2
3
4
5
6
7
8
9
```

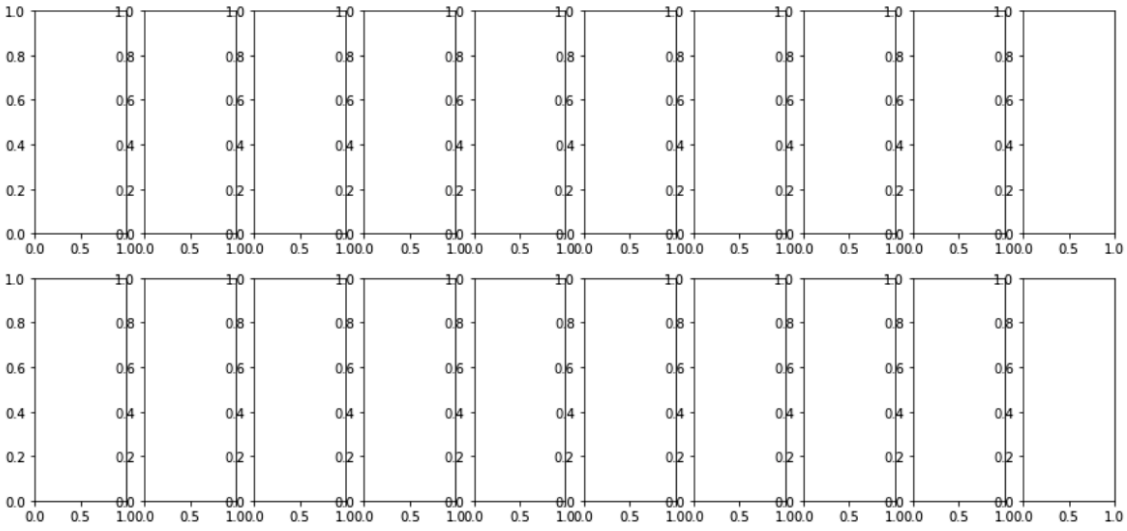
(b)

In [17]:
```
data_mnist.LDA()
```

```
Dataset splitted!
```



```
Error rate= 4.556000
```



(c)

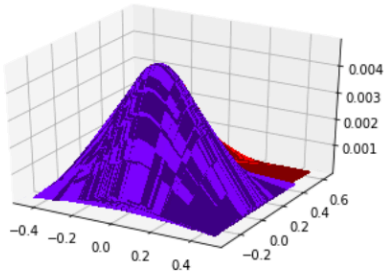
In [18]:
```
data_mnist.QDA()
```

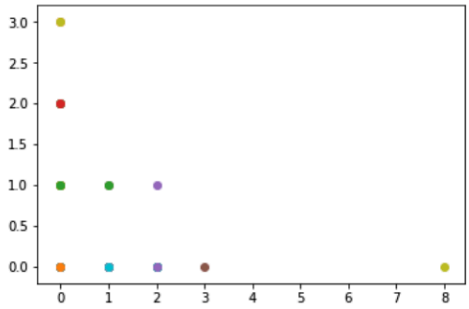
```
QDA
```

(d)

In [19]:
```
data_spam = comp("spam", val_size=0.2)
trainerror, testerror = data_spam.fit()
data_spam.plot_bivariate_gaussians()
data_spam.plot_proj_2D()
data_spam.QDA()
#data_spam.predict_kaggle(5000, C=spam_C)
#data_mnist.predict_kaggle(10000, C=mnist_C)
```

```
loaded spam dataset!
Class is initialized!
0
1
train error is:
0    0.290023
dtype: float64
```






```
QDA
```

In [14]:

```
from __future__ import division
import os
from scipy import io
import numpy as np
import matplotlib.pyplot as plt
import sys, itertools
import pandas as pd
import matplotlib.cm as cm
from mpl_toolkits.mplot3d import Axes3D
import random
import math
import matplotlib.pyplot as plt
%pylab inline
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
from scipy.stats import norm
from operator import itemgetter
from itertools import groupby
from numpy import linalg as LA
#LA.norm(a, 2)

print('Finished loading libraries')

class comp(object):

    def __init__(self, dataset_name, val_size, seed=1):
        self.dataset_name = dataset_name
        self.val_size = val_size
        self.data = None
        self.seed = seed
        self.random = np.random.RandomState(seed)
        self.labelcol = 2
        self.load_data(dataset_name)
        print("Class is initialized!")

    def split(self, data, labels, val_size):
        num_items = len(data)
        assert num_items == len(labels)
        assert val_size >= 0
        if val_size < 1.0:
            val_size = int(num_items * val_size)
        train_size = num_items - val_size
        idx = self.random.permutation(num_items)
        data_train = data[idx][:train_size]
        label_train = labels[idx][:train_size]
        data_val = data[idx][train_size:]
        label_val = labels[idx][train_size:]
        print("Dataset splitted!")
        return data_train, data_val, label_train, label_val

    def load_data(self,dataset_name):
        self.data = io.loadmat("data/%s_data.mat" % dataset_name)
        print("\nloaded %s dataset!" % dataset_name)

    def train(self, X_train, Y_train, C=1.0):
        model = svm.SVC(kernel='linear', C=C)
        model.fit(X_train, Y_train)
        return model

    def train_all_sizes(self, input_train_sizes):
        print(self.data)
        training_data = self.data["training_data"]
        training_labels = self.data['training_labels']
        X_train, X_val, Y_train, Y_val = self.split(training_data,
                                                    training_labels,
                                                    self.val_size)

        print(X_train.shape, X_val.shape, self.val_size)
        train_sizes, train_accuracies, val_accuracies = [], [], []
        end_training = False
        for train_size in input_train_sizes:
            if train_size > X_train.shape[0]:
                train_size = X_train.shape[0]
                end_training = True
```

```
            model = self.train(X_train[:train_size], Y_train[:train_size])
            Y_train_prediction = model.predict(X_train[:train_size])
            train_accuracy = metrics.accuracy_score(Y_train[:train_size], Y_train_prediction)
            train_sizes.append(train_size)
            train_accuracies.append(train_accuracy)
            Y_val_prediction = model.predict(X_val)
            val_accuracy = metrics.accuracy_score(Y_val, Y_val_prediction)
            val_accuracies.append(val_accuracy)
            print("train_size", train_size,
                  "train accuracy", train_accuracy,
                  "val accuracy", val_accuracy)
            if end_training:
                break
        return train_sizes, train_accuracies, val_accuracies

    def predict_kaggle(self, train_size, C=1.0):
        num_examples = self.data['training_data'].shape[0]
        val_size = num_examples - train_size
        training_data, _, training_labels, _ = self.split(self.data['training_data'],
                                                          self.data['training_labels'],
                                                          val_size)
        test_data = self.data['test_data']
        model = self.train(training_data, training_labels, C=C)
        test_labels = model.predict(test_data)

        filename = os.path.join(os.path.split(__file__)[0], '%s_solution.csv' % self.dataset_name)
        f = open(filename, 'w')
        f.write("Id,Category\n")
        for i, y in enumerate(test_labels):
            f.write(str(i + 1) + ',' + str(y) + '\n')
        f.close()


    def plot_result(self, train_sizes, train_accuracies, val_accuracies, title, filename):
        plt.figure()
        plt.title(title)
        plt.xlabel("Number of Examples")
        plt.ylabel("Accuracy")
        plt.plot(train_sizes, train_accuracies, label="Training Set")
        plt.plot(train_sizes, val_accuracies, label="Validation Set")
        plt.legend()
        plt.tight_layout()
        #plt.savefig(os.path.join(os.path.split(__file__)[0], filename))


    def fit(self):
        # Function estimates the LDA parameters
        def estimate_params(data):
            # group data by label column
            #grouped = data.groupby(self.data.ix[:,self.labelcol])

            # calculate means for each class
            means = {}
            for c in self.classes:
                #means[c] = np.array(self.drop_col(self.classwise[c], self.labelcol).mean(axis = 0))
                means[c] = np.array(data.mean(axis = 0))

            # calculate the overall mean of all the data
            #print("here")
            #print(data.shape)
            #overall_mean = np.array(self.drop_col(data, self.labelcol).mean(axis = 0))
            overall_mean = np.array(data.mean(axis = 0))
            #print(overall_mean.shape)
            #overall_mean = overall_mean[0:(len(overall_mean)-1),]
            # calculate between class covariance matrix
            # S_B = \sigma{N_i (m_i - m) (m_i - m).T}
            #S_B = np.zeros((data.shape[1] - 1, data.shape[1] - 1))
            S_B = np.zeros((data.shape[1], data.shape[1]))
            #print(S_B.shape)
            for c in means.keys():
                #print(len(self.classwise[c]))
                #print(means[c])
                #print(len(overall_mean))
                meansC = means[c]
                #print(meansC.shape)
                #meansC = meansC[0:(len(meansC)-1),]
                #print(meansC.shape)
                #print(np.outer((meansC - overall_mean), (meansC - overall_mean)))
                S_B += np.multiply(len(self.classwise[c]),np.outer((meansC - overall_mean), (meansC - overall_mean)))
            #print(S_B)
            # calculate within class covariance matrix
            # S_W = \sigma{S_i}
            # S_i = \sigma{(x - m_i) (x - m_i).T}
```

```
            S_W = np.zeros(S_B.shape)
            for c in self.classes:
                #tmp = np.subtract(self.drop_col(self.classwise[c], self.labelcol).T, np.expand_dims(means[c], axis=
1))
                tmp = np.subtract(data.T, np.expand_dims(means[c], axis=1))
                S_W = np.add(np.dot(tmp, tmp.T), S_W)

            # objective : find eigenvalue, eigenvector pairs for inv(S_W).S_B
            mat = np.dot(np.linalg.pinv(S_W), S_B)
            eigvals, eigvecs = np.linalg.eig(mat)
            eiglist = [(eigvals[i], eigvecs[:, i]) for i in range(len(eigvals))]

            # sort the eigvals in decreasing order
            eiglist = sorted(eiglist, key = lambda x : x[0], reverse = True)

            # take the first num_dims eigvectors
            w = np.array([eiglist[i][1] for i in range(2)])

            self.w = w
            self.means = means
            return



        tl = pd.DataFrame.from_dict(self.data['training_labels'])
        #print(tl.shape)
        tl = tl.iloc[:,0]

        dataPD = pd.DataFrame(self.data['training_data'])

        grouped = dataPD.groupby(tl)
        self.classes = [c for c in grouped.groups.keys()] #[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        self.classwise = {}
        for c in self.classes:
            #print(c)
            self.classwise[c] = grouped.get_group(c)

        traindata = dataPD#self.data["training_data"]
        testdata = pd.DataFrame(self.data['test_data'])#self.data["test_data"]


        # estimate the LDA parameters
        estimate_params(traindata)
        # perform classification on test set

        self.gaussian_modeling()
        # append the training and test error rates for this iteration
        #print(type(self.calculate_score_gaussian(traindata)))
        #print(type(float(traindata.shape[0])))
        #print(type(traindata.shape[0]))
        trainerror = self.calculate_score_gaussian(traindata) / float(traindata.shape[0])
        print("train error is:")
        print(trainerror)
        #testerror = self.calculate_score_gaussian_test(testdata) / float(testdata.shape[0])
        testerror=0
        #print("test error is:")
        #print(testerror)

        return trainerror, testerror

    def drop_col(self, data, col):
        return data.drop(data.columns[[col]], axis = 1)

    def gaussian_modeling(self):
        self.priors = {}
        self.gaussian_means = {}
        self.gaussian_cov = {}

        for c in self.means.keys():
            print(c)
            #inputs = self.drop_col(self.classwise[c], self.labelcol)
            inputs = pd.DataFrame(self.data['training_data'])
            #if c == 0:
            #    self.w = self.w[0:,:-1]
            #print(self.w.shape)
            #print(inputs.shape)
            proj = np.dot(self.w, inputs.T).T
            #print(proj)
            #print(inputs.shape)
```

```
        #print(len(tl))
        labels = pd.DataFrame(labels)

        errors = np.sum(labels != tl)
        #print(type(errors))
        return errors

    def pdf(self, point, mean, cov):
        cons = 1./((2*np.pi)**(len(point)/2.)*np.linalg.det(cov)**(-0.5))
        try:
            return cons*np.exp(-np.dot(np.dot((point-mean),np.linalg.inv(cov)),(point-mean).T)/2.)
        except:
            return np.exp(0)

    def plot_bivariate_gaussians(self):
        classes = list(self.means.keys())
        colors = cm.rainbow(np.linspace(0, 1, len(classes)))
        plotlabels = {classes[c] : colors[c] for c in range(len(classes))}

        fig = plt.figure()
        ax3D = fig.add_subplot(111, projection='3d')
        for c in self.means.keys():
            data = np.random.multivariate_normal(self.gaussian_means[c],
                                                 self.gaussian_cov[c], size=100)
            pdf = np.zeros(data.shape[0])
            cons = 1./((2*np.pi)**(data.shape[1]/2.)*np.linalg.det(self.gaussian_cov[c])**(-0.5))
            X, Y = np.meshgrid(data.T[0], data.T[1])
            def pdf(point):
                try:
                    return cons*np.exp(-np.dot(np.dot((point-self.gaussian_means[c]),np.linalg.inv(self.gaussian_cov[c
]),(point-self.gaussian_means[c]).T)/2.)
                except:
                    return np.exp(0)

            zs = np.array([pdf(np.array(ponit)) for ponit in zip(np.ravel(X), np.ravel(Y))])
            Z = zs.reshape(X.shape)
            surf = ax3D.plot_surface(X, Y, Z, rstride=1, cstride=1,
                                     color=plotlabels[c], linewidth=0,
                                     antialiased=False)
        plt.show()

    def plot_proj_1D(self):
        data =  pd.DataFrame(self.data['training_data'])
        classes = list(self.means.keys())
        colors = cm.rainbow(np.linspace(0, 1, len(classes)))
        plotlabels = {classes[c] : colors[c] for c in range(len(classes))}

        fig = plt.figure()
        for i, row in data.iterrows():
            proj = np.dot(self.w, row)
            plt.scatter(proj, np.random.normal(0,1,1)+0)
        plt.show()

    def plot_proj_2D(self):
        data =  pd.DataFrame(self.data['training_data'])
        classes = list(self.means.keys())
        colors = cm.rainbow(np.linspace(0, 1, len(classes)))
        plotlabels = {classes[c] : colors[c] for c in range(len(classes))}

        fig = plt.figure()
        for i, row in data.iterrows():
            proj = np.dot(self.w, row)
            plt.scatter(proj[0], proj[1])
        plt.show()



    def LDA(self):
#digits = datasets.load_digits()
#X_train, X_test, Y_train, Y_test = train_test_split(digits.data, digits.target, test_size=0.4, random_state=4)
        training_data = self.data['training_data']
        training_labels = self.data['training_labels']
        X_train, X_test, Y_train, Y_test = self.split(training_data,training_labels,self.val_size)
        #X_train.shape
        from sklearn.naive_bayes import GaussianNB
        helper = GaussianNB()
        helper.fit(X_train, Y_train)
        classes = helper.classes_
        priors = helper.class_prior
```

```
priors = helper.class_prior_
#print(priors)
        posteriors=[]
        def displaychar(image):
            plt.imshow(np.reshape(image, (8,8)), cmap=plt.cm.gray)
            plt.axis('off')
        def get_examples_for_class(class_id):
            examples = []
            for i, example in enumerate(X_train):
                if Y_train[i]==class_id:
                    examples.append(example)

            examples = np.matrix(examples)
            return examples
        for klass in classes:
            examples = get_examples_for_class(klass)
            mean = np.array(examples.mean(0))[0]
            cov = np.cov(examples.T)
            p_x = multivariate_normal(mean=mean, cov=cov)
            posteriors.append(p_x)

#choose a random point from the test data
#print(X_test)
#print(X_test.shape)
        x = X_test[4,0:] #random.choice (X_test[0:,0])
#print(x)

        bayes_probs = []
        for klass in classes:
    #print(posteriors[klass])
            prob = [klass,(priors[klass] * np.dot(posteriors[klass],norm.pdf(x)))]
            bayes_probs.append(prob)
#print(bayes_probs)

        prediction = max(bayes_probs, key= lambda a: a[1].any())
#print(digits.target_names[prediction[0]])


        plt.figure(1, figsize=(3, 3))
#print(x)
        plt.imshow(x.reshape(28,28), cmap=plt.cm.gray_r, interpolation='nearest')
        plt.show()


        Y = []
        for x in X_test:
            bayes_probs = []
            for klass in classes:
                prob = [klass, priors[klass] * np.dot(posteriors[klass],norm.pdf(x))]
                bayes_probs.append(prob)
            prediction = max(bayes_probs, key= lambda a: a[1])
            Y.append(prediction[0])


        errors = (Y_test != Y).sum()
        total = X_test.shape[0]
        print("Error rate= %f" % ((errors/float(self.val_size)/20)))



        indicies = np.array(np.where((Y_test != Y)==True))[0]

        index = 0
        rows = len(indicies)%10
        cols = 10
        dv = 0

        plt.figure(figsize=(15,7))
        for i in indicies:
            index += 1
            try:
                plt.subplot(rows, cols, index)
                displaychar(X_test[i])
                print(X_test[i])
                plt.title('exp:%i, act:%i' %( Y[i],Y_test[i]), fontsize = 10)
            except:
                dv += 1
```

```
def MLE(self):

    training_data = self.data['training_data']
    training_labels = self.data['training_labels']
    X_train, X_test, Y_train, Y_test = self.split(training_data,training_labels,self.val_size)
    #X_test = X_train
    #Y_test = Y_train

    tl = pd.DataFrame.from_dict(Y_test)
    tl = tl.iloc[:,0]
    dataPD = pd.DataFrame(X_test)
    grouped = dataPD.groupby(tl)
    classes = [c for c in grouped.groups.keys()]
    overall_mean = np.array(X_test.mean(axis = 0))
    S_B = np.zeros((X_test.shape[1], X_test.shape[1]))
    classwise = {}
    means = {}
    for c in classes:
        print(c)
        means[c] = np.array(X_test.mean(axis = 0))
        meansC = means[c]
        classwise[c] = grouped.get_group(c)
        S_B += np.multiply(len(classwise[c]),np.outer((meansC - overall_mean), (meansC - overall_mean)))

    S_W = np.zeros(S_B.shape)
    for c in classes:
        tmp = np.subtract(X_test.T, np.expand_dims(means[c], axis=1))
        S_W = np.add(np.dot(tmp, tmp.T), S_W)

    mat = np.dot(np.linalg.pinv(S_W), S_B)
    #print(mat)
    eigvals, eigvecs = np.linalg.eig(mat)
    eiglist = [(eigvals[i], eigvecs[:, i]) for i in range(len(eigvals))]
    eiglist = sorted(eiglist, key = lambda x : x[0], reverse = True)
    w = np.array([eiglist[i][1] for i in range(2)])
    x = X_test
    max_val = max_val_location = None
    mu_set={}
    sd_set={}
    cov_set = {}
    dv = 0
    for c in classes:
        proj = np.dot(w, X_train.T).T
        if c <9:
            sd_set[c] = np.std(proj, axis = 0)
            mu_set[c] = np.mean(means[c], axis = 0)
            cov_set[c] = np.cov(proj, rowvar=False)
        else:
            dv += 1
    #print(proj)
    try:
        for i in sd_set:
            ll_array = []
            for j in mu_set:
                temp_mm = 0
                for k in x:
                    try:
                        if math.isnan(np.log(norm.pdf(k, j, i))):
                            dv += 1
                            temp_mm += 0
                        else:
                            temp_mm += np.log(norm.pdf(k, j, i))
                        ll_array.append(temp_mm)
                        if (max_val is None):
                            max_val = max(ll_array)
                        elif max(ll_array) > max_val:
                            max_val = max(ll_array)
                            max_val_location = j
                    except:
                        dv += 1
    except:
        dv+=1
```

```
        try:
            for c in classes:
                plt.plot(mu_set[c], ll_array[c], label="sd: %.1f" % i)
                print(c)
                print("The max LL for sd %.2f is %.2f" % (i, max(ll_array)))
                plt.axvline(x=max_val_location, color='black', ls='-.')
                plt.legend(loc='lower left')
                plt.title("Maximim Likelihood Functions")
                plt.xlabel("Mean Estimate")
                plt.ylabel("Log Likelihood")
                plt.axis('auto')
                plt.show()
        except:
            dv+=1

    def QDA(self):
        print("QDA")

    def substract_mean(self, group_point):
        for i, a in enumerate(group_point):
            group_point[i] = group_point[i] - self.mean_global[i]

        return group_point

    #--gets the covariance matrix for a dataset/group (t(x).x/len(x))
    def get_cov_matrix(self, matrix):
        cov_mat = []
        for i in zip(*matrix):
            l = []
            for e in zip(*matrix):
                l.append(self.funcs_.dot(i, e)/len(matrix))
            cov_mat.append(l)
        return cov_mat

    #training the model
    def trainQDA(self):
        self.mean_sets = {}
        self.covariance_sets = {}
        self.lens ={}
        self.probability_vector = {}

        self.mean_global = self.funcs_.mean_nm(self.x, axis=0)
        for k, v in self.groups.iteritems():
            self.lens[k] = len(self.groups[k])
            self.probability_vector[k] = self.lens[k]/self.len_all
            self.mean_sets[k] = self.funcs_.mean_nm(self.groups[k], axis=0)
            #mean correcting each set i.e. set - global mean
            self.groups[k] = map(self.substract_mean, self.groups[k])
            self.covariance_sets[k] = self.get_cov_matrix(self.groups[k])

        #here is the major difference between QDA and LDA .. no calculation for the
        #global pooled covariance matrix, but each class has its own pooled
        #covariance matrix.
        for k, v in self.covariance_sets.iteritems():
            self.covariance_sets[k] = self.funcs_.inv_(self.covariance_sets[k])

    #prediction or discrimnant function
    #prediction function for QDA is a little bit different then that of LDA :-)
    def predict(self, v, key_only=True):
        difference_vectors = lambda n,m: n-m
        predictions  = {}
        for a in self.group_names:
            predictions[a] = self.funcs_.dot(self.funcs_.prod_2(self.funcs_.sclr_prod_(-0.5, map(difference_vectors, v
, self.mean_sets[a])), self.covariance_sets[a]), map(difference_vectors, v, self.mean_sets[a])) - \
                             (0.5*math.log(self.funcs_.det_(self.covariance_sets[a]))) + math.log(self.probability_vec
tor[a])
        if key_only:
            return max(predictions, key=predictions.get)
        else:
            return predictions

if __name__ == "__main__":
    print("Python Class Ready!")
```

```
Populating the interactive namespace from numpy and matplotlib
Finished loading libraries
Python Class Ready!
```

In [15]:

```
def predict_kaggle(name, training_data, training_labels, test_data, C=1.0):
    model = train(training_data, training_labels, C=C)
    test_labels = model.predic
    t(test_data)

    filename = os.path.join(os.path.split(__file__)[0], '%s_solution.csv' % name)
    f = open(filename, 'w')
    f.write("Id,Category\n")
    for i, y in enumerate(test_labels):
        f.write(str(i + 1) + ',' + str(y) + '\n')
    f.close()
```