COMP755-Lect07

September 10, 2018

1 COMP 755

Plan for today

1. Review multi-class logistic regression
2. Look at an example of online learning
3. Multivariate Gaussian distribution
4. Generative models
5. Naive Bayes

```
In [1]: import numpy
        import matplotlib.pyplot as plt
        % matplotlib inline
        # plotting
        def draw_line(w,b,c='k',linewidth=1):
            if numpy.abs(w[1]) > numpy.abs(w[0]):
                y1 = -(1.0*w[0] + b)/w[1]
                y0 = -b/w[1]
                x1 = 1
                x0 = 0
            else:
                x1 = -(1.0*w[1] + b)/w[0]
                x0 = -b/w[0]
                y1 = 1
                y0 = 0

            h = plt.plot([x0,x1],[y0,y1],
                         c,alpha=1.0,
                         linewidth=linewidth)
            plt.xlim([0,1])
            plt.ylim([0,1])
            return h

        hs = []
        def add_to_plot(sample,w,b):
            xi,yi = sample
            pt = plt.annotate(str(int(yi)),xy = xi,
                         alpha=1.0,
```

```
                horizontalalignment='center',
                verticalalignment='center')
h, = draw_line(w,b)

for other in hs:
    other.set_alpha(0.99*other.get_alpha())
hs.append(h)
hs.append(pt)
```

2 Log-Likelihood

1. There are N samples, each in one of C classes, and p features
2. Labels are represented using one-hot vectors y_i
3. Feature matrix X contains a column of 1s -- corresponding to the bias term.
4. First row of weight matrix B are bias terms.
5. $\mathbf{fi}_k$ is k^{th} column of matrix B

	Variable	Dimensions
feature matrix	X	$N \times (p+1)$
label matrix	Y	$N \times C$
weight matrix	B	$(p+1) \times C$

Likelihood is

$$\mathcal{L}(B|Y,X) = \underbrace{\prod_{i=1}^{N}}_{\text{samples}} \underbrace{\prod_{c=1}^{C}}_{\text{classes}} \left[\frac{\exp\left\{\mathbf{x}_i^T \mathbf{fi}_c\right\}}{\sum_{k=1}^{C} \exp\left\{\mathbf{x}_i^T \mathbf{fi}_k\right\}} \right]^{y_{i,c}}$$

Log-likelihood is

$$\mathcal{LL}(\beta_0, B|Y,X) = \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \left(\mathbf{x}_i^T \mathbf{fi}_c - \log\left\{ \sum_{k=1}^{C} \exp\left\{\mathbf{x}_i^T \mathbf{fi}_k\right\} \right\} \right)$$

3 Ridge regularized log-likelihood

Ridge regularized log-likelihood

$$\mathcal{PLL}(B|Y,X) = \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \left(\mathbf{x}_i^T \mathbf{fi}_c - \log\left\{ \sum_{k=1}^{C} \exp\left\{\mathbf{x}_i^T \mathbf{fi}_k\right\} \right\} \right) - \frac{\lambda}{2} \sum_{k=1}^{C} \sum_{j=1}^{p} \beta_{j,k}^2$$

Note that we keep the last column of B fixed at 0 to get rid of excess parameters.
These parameters will not contribute to the regularization -- sum of their squares is 0.

4 Gradients of multi-class logistic regression likelihood

Using

$$\mu_{i,l} = \frac{\exp\left\{\mathbf{x}_i^T \mathbf{fi}_l\right\}}{\sum_{k=1}^{C} \exp\left\{\mathbf{x}_i^T \mathbf{fi}_k\right\}},$$

partial derivatives are

$$\frac{\partial}{\partial \beta_{j,l}} \mathcal{LL}(B) = \sum_i \underbrace{x_{i,j}}_{\text{feature } j} \underbrace{(y_{i,l} - \mu_{i,l})}_{\text{residual in predicting class } l} = \mathbf{x}_{:,j}^T (\mathbf{y}_{:,l} - \mu_{:,l}),$$

gradient of log likelihood with respect to a column of B

$$\nabla_{\beta_c} \mathcal{LL}(B) = \sum_i (y_{i,c} - \mu_{i,c}) \mathbf{x}_i,$$

and gradient of ridge regularized log-likelihood with respect to a column of B

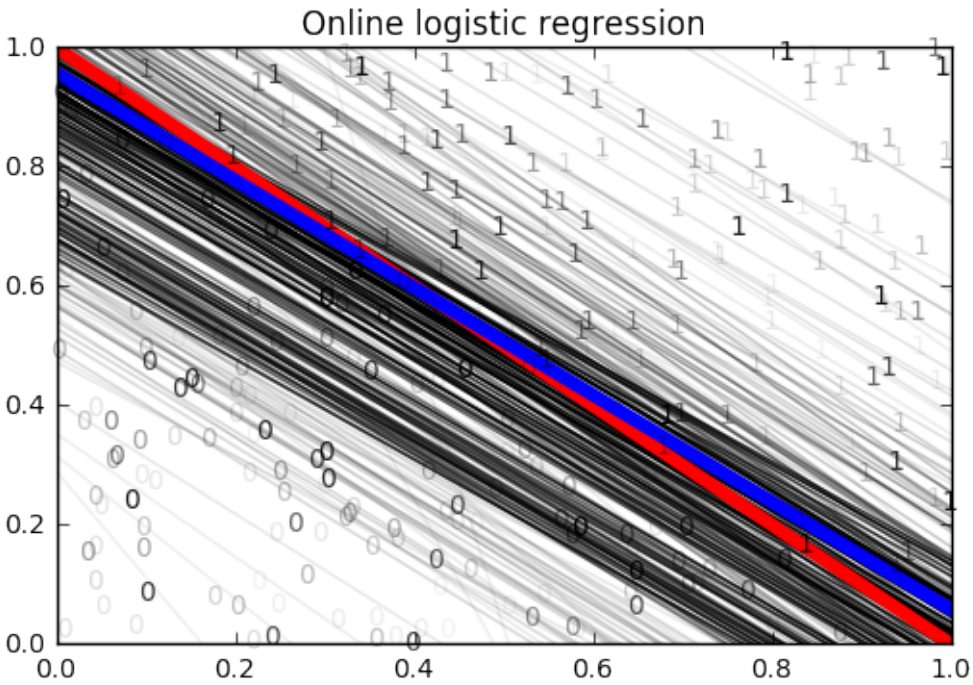
$$\nabla_{\beta_c} \mathcal{PLL}(B) = \sum_i (y_{i,c} - \mu_{i,c}) \mathbf{x}_i - \lambda \begin{bmatrix} 0 \\ \mathbf{1}_p \end{bmatrix}$$

In [326]:
```
# A simple example of online learning

# sample_generator generates a sample at a time
# if you are curious about what is going on here
# read about python generators.
def sample_generator(N,true_w,true_b):
    true_w = numpy.asarray(true_w)
    d = len(true_w)
    for i in range(N):
        x = numpy.random.rand(d)
        y = float((numpy.dot(x,true_w) + true_b) > 0)
        yield x,y

def update(w,b,sample,it):
    xi,yi = sample
    v = numpy.exp(numpy.dot(xi.transpose(),w) + b)
    res = yi - v/(1.0 + v)
    gw = res*xi
    gb = res
    # step size diminishes with iterations
    step = 1.0/(it**0.25)
    w = w + step*gw
    b = b + step*gb
    return w,b

numpy.random.seed(2)
# synthetic data generator for illustration purposes
true_w = [1.0,1.0]
```

```
true_b = -1.0
N = 300
data_stream = sample_generator(N,true_w,true_b)


# actual fitting of the parameters
# initialization
w = numpy.asarray([-0.1,-0.2])
b = 0.05
it = 0

# while more data is available
while True:
    try:
        (xi,yi) = data_stream.next()
        it += 1
        w,b = update(w,b,(xi,yi),it)
        add_to_plot((xi,yi),w,b)
    except StopIteration:
        # no more data
        break

draw_line(true_w,true_b,'r',linewidth=5)
draw_line(w,b,'b',linewidth=5)

plt.title('Online logistic regression')

print w,b
```

This is python2 version. In python3, __next__

```
[ 2.94467154  3.2747466 ] -3.13538881976
```



5 Online learning and Stochastic Gradient Descent (SGD)

Samples are delivered in small mini-batches

Parameters are updated using gradient computed on the single most recent sample.

Mini-batch gradients are noisy compared to the gradient on the whole data batch. Hence the name **stochastic gradient ascent/descent**.

Sophisticated SGD schemes involve use of momentum -- averaging of gradients seen thus far.

Typically SGD used in large data settings, where processing whole batch of data is expensive.

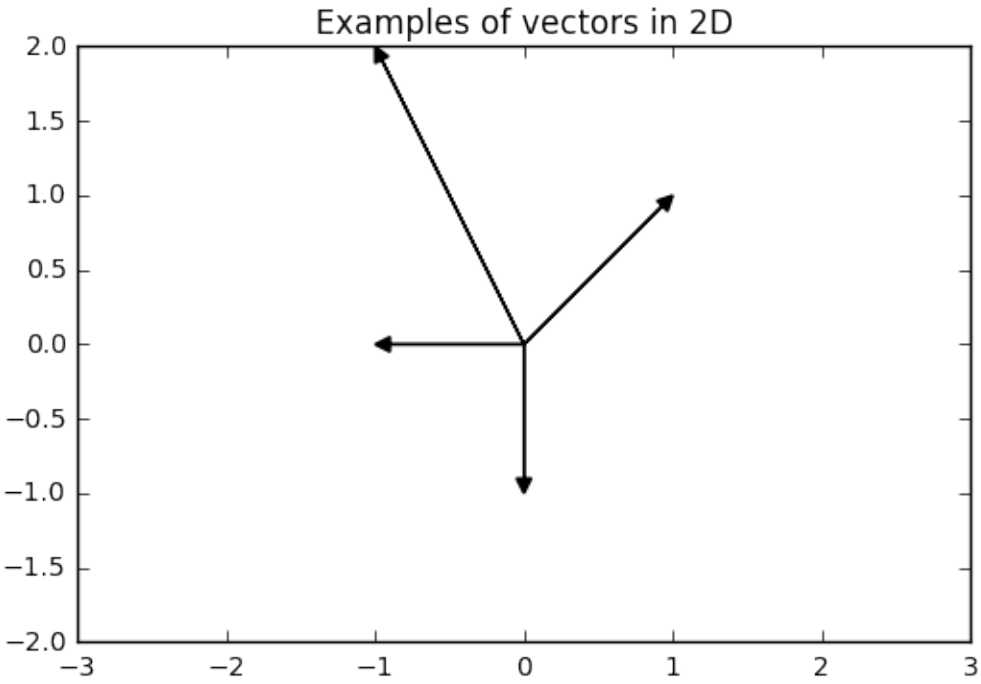
6 Linear transformations

We are going to review some of the linear algebra/analytic geometry

We will look at examples in 2D, $\mathbf{x} \in \mathbb{R}^2$, and use 2×2 matrices.

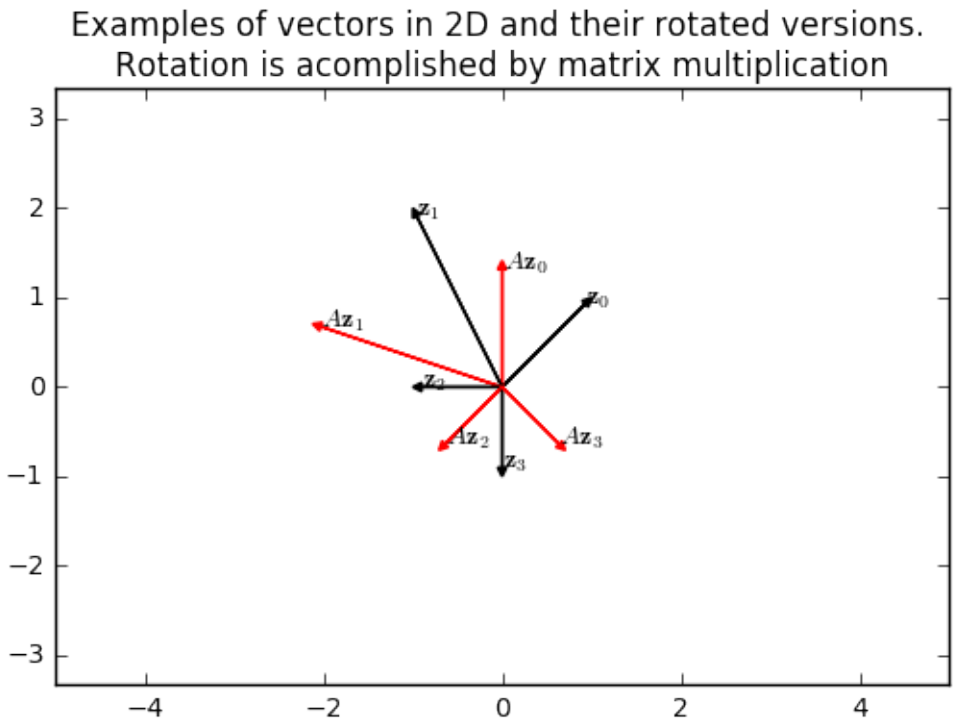
```
In [327]: import numpy
          import matplotlib.pyplot as plt

          def draw_vector(zz,color='k',label=None):
              z1 = zz[0]
              z2 = zz[1]
              length = numpy.sqrt(z1**2+z2**2)
              head_length = 0.1*numpy.min([length,1.0])
              z1 = z1 - z1/length*head_length
```

```
        z2 = z2 - z2/length*head_length
        plt.arrow(0,0,z1,z2,ec=color,fc=color,
                  head_length=head_length,head_width=0.1)
        if not label is None:
            plt.annotate(label,(z1,z2))

draw_vector([1.0,1.0])
draw_vector([-1.0,2.0])
draw_vector([-1.0,0])
draw_vector([0,-1.0])
plt.axis('equal')
plt.xlim([-3,3])
plt.ylim([-3,3])
plt.title('Examples of vectors in 2D');
```




```
In [290]: def illustrate_linear_transformation(A,Z):
              for i in range(Z.shape[1]):
                  zz = Z[:,i]
                  draw_vector(zz,label='$\mathbf{z}_'+str(i)+'$')
                  xx = numpy.dot(A,zz) # perform linear transformation
                  draw_vector(xx,'r',label='$A\mathbf{z}_'+str(i) + '$')

              plt.axis('equal')
              plt.axis([-5,5,-5,5])
```

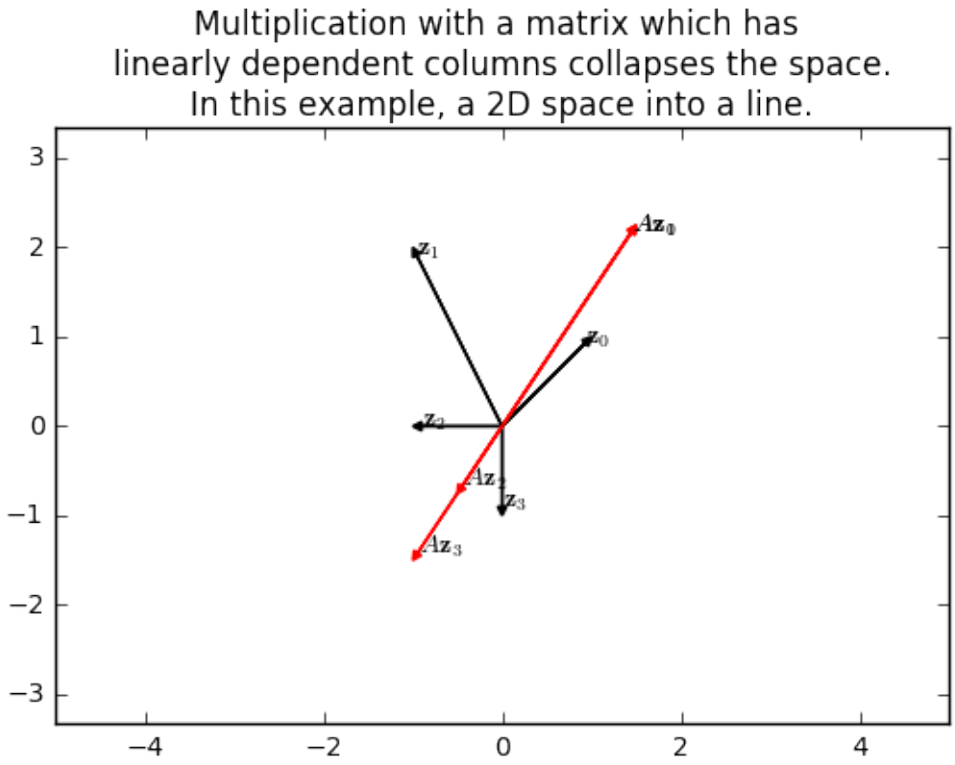
```
Z = numpy.asarray([[1.0,-1.0,-1.0, 0.0],
                   [1.0, 2.0, 0.0,-1.0]])

theta = 2.0*numpy.pi/8.0
A = numpy.asarray([[numpy.cos(theta), -numpy.sin(theta)],
                   [numpy.sin(theta), numpy.cos(theta)]])

illustrate_linear_transformation(A,Z)

plt.title(('Examples of vectors in 2D and their rotated versions. \n'+
           'Rotation is acomplished by matrix multiplication'));
```



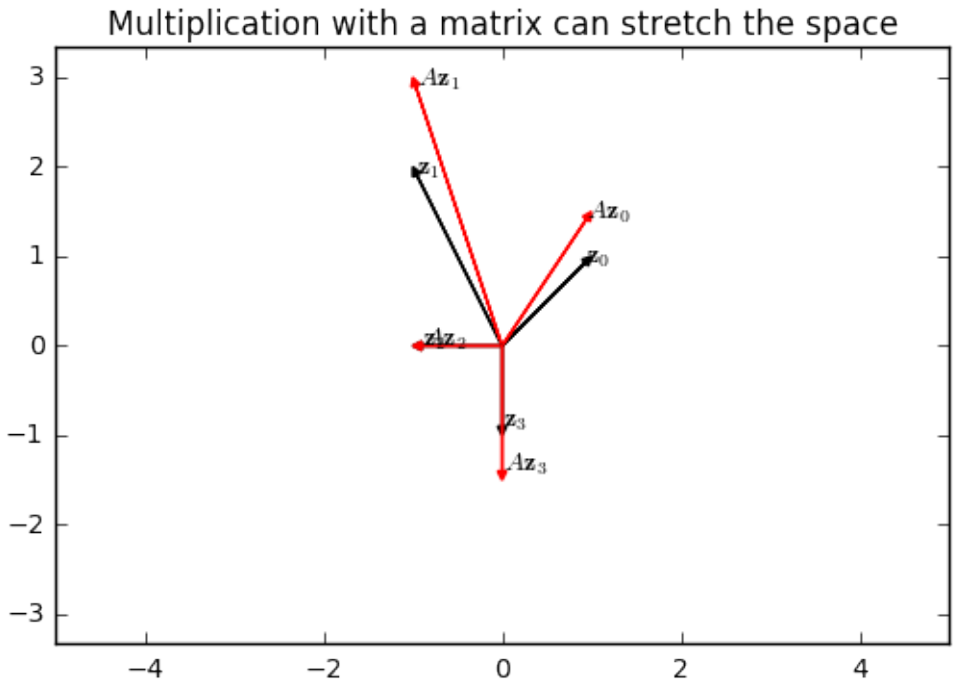
In [329]:
```
Z = numpy.asarray([[1.0,-1.0,-1.0, 0.0],
                   [1.0, 2.0, 0.0,-1.0]])

A = numpy.asarray([[0.5 ,1.0],
                   [0.75,1.5]])

print "Transformed vectors:"
print numpy.dot(A,X)

illustrate_linear_transformation(A,Z)
```

```
        plt.title(('Multiplication with a matrix which has \n'+
                   'linearly dependent columns collapses the space.\n' +
                   'In this example, a 2D space into a line.'));
```

```
Transformed vectors:
[[ 1.5   1.5  -0.5  -1.  ]
 [ 2.25  2.25 -0.75 -1.5 ]]
```



In [292]:
```
A = numpy.asarray([[1.0,0.0],
                   [0.0,1.5]])
Z = numpy.asarray([[1.0,-1.0,-1.0, 0.0],
                   [1.0, 2.0, 0.0,-1.0]])

illustrate_linear_transformation(A,Z)
plt.title('Multiplication with a matrix can stretch the space');
```



7 Eigenvectors and eigenvalues

For a linear transformation represented by matrix A, $\mathbf{x}$ is an **eigenvector** if it is non-zero and

$$A\mathbf{x} = \lambda\mathbf{x}$$

λ is called **eigenvalue**.

In practice an eigenvector will be of length 1 $\|\mathbf{x}\| = \sqrt{\sum_i x_i^2} = 1$. You can interpret it as direction along which space gets scaled by λ

Matrix

$$A = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 2.0 \end{bmatrix}$$

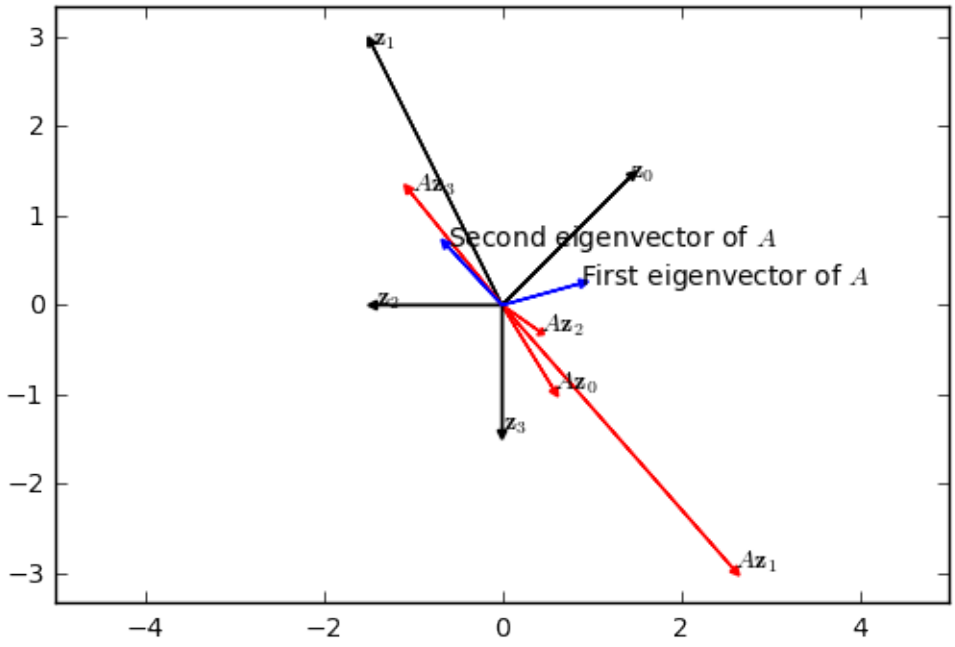
has eigenvectors $\mathbf{x}_1 = \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix}$ and $\mathbf{x}_2 = \begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix}$, with corresponding eigenvalues $\lambda_1 = 1.0$ and $\lambda_2 = 2.0$.

```
In [293]: Z = 1.5*numpy.asarray([[1.0,-1.0,-1.0, 0.0],
                                 [1.0, 2.0, 0.0,-1.0]])
          numpy.random.seed(6)
          A = numpy.random.randn(2,2)

          illustrate_linear_transformation(A,Z)
          l,W = numpy.linalg.eig(A)
          draw_vector(W[:,0],'b',label='First eigenvector of $A$')
```

```
draw_vector(W[:,1],'b',label='Second eigenvector of $A$')
print "Eigenvectors"
print W
print "Eigenvalues"
print l

Winv = numpy.linalg.inv(W)
L = numpy.diag(l)
# A = W*diag(L)*W^{-1}
A - numpy.dot(numpy.dot(W,L),Winv)
```

```
Eigenvectors
[[ 0.96391077 -0.67879055]
 [ 0.26622551  0.73433194]]
Eigenvalues
[-0.11043782 -1.10043765]
```

```
Out[293]: array([[ -5.55111512e-17,  -2.22044605e-16],
                 [  0.00000000e+00,   1.11022302e-16]])
```

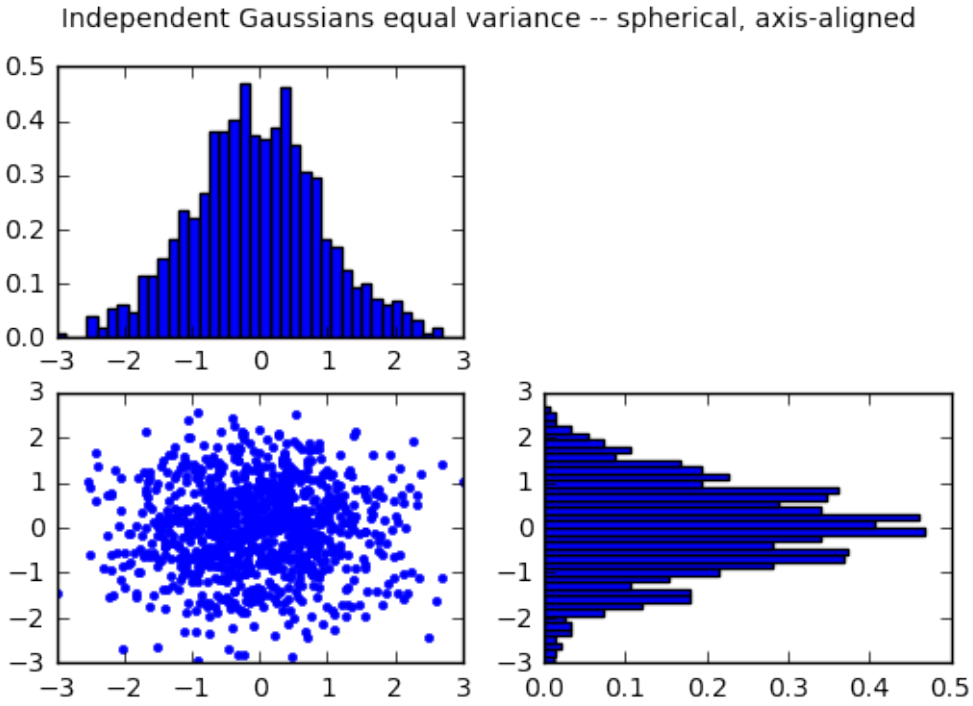


8 Multivariate Gaussian distribution

We will first consider a simple generalization of a Gaussian distribution to 2D

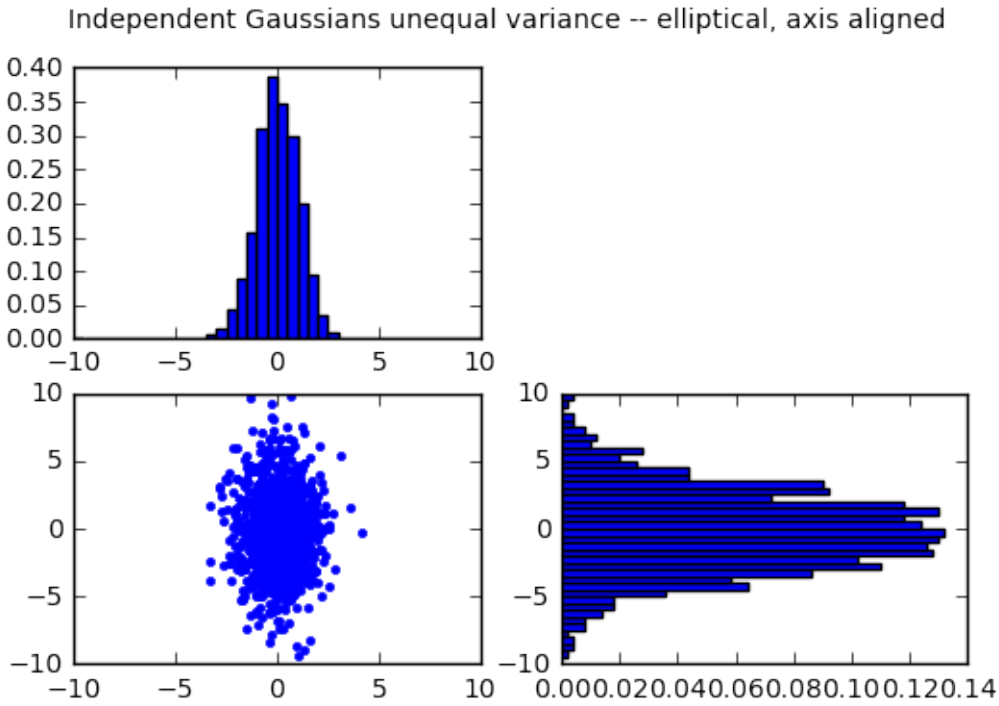
$$z_1 \sim \mathcal{N}\left(0, \sigma_1^2\right)$$
$$z_2 \sim \mathcal{N}\left(0, \sigma_2^2\right)$$

In [330]:
```
import matplotlib.pyplot as plt
sigma_1 = 1
sigma_2 = 1
bounds = [-3,3]
z1 = sigma_1*numpy.random.randn(1000)
z2 = sigma_2*numpy.random.randn(1000)

plt.subplot(2,2,1)
plt.hist(z1,range=bounds,bins=40,normed=True)
plt.subplot(2,2,4)
plt.hist(z2,range=bounds,bins=40,orientation='horizontal',normed=True)

plt.subplot(2,2,3)
plt.plot(z1,z2,'.')
plt.xlim(bounds)
plt.ylim(bounds)
plt.suptitle('Independent Gaussians equal variance -- spherical, axis-aligned');
```



In [331]:
```
import matplotlib.pyplot as plt
sigma_1 = 1
sigma_2 = 3
bounds = [-10,10]
z1 = sigma_1*numpy.random.randn(1000)
z2 = sigma_2*numpy.random.randn(1000)
```

```
plt.subplot(2,2,1)
plt.hist(z1,range=bounds,bins=40,normed=True)
plt.subplot(2,2,4)
plt.hist(z2,range=bounds,bins=40,orientation='horizontal',normed=True)

plt.subplot(2,2,3)
plt.plot(z1,z2,'.')
plt.xlim(bounds)
plt.ylim(bounds)
plt.suptitle('Independent Gaussians unequal variance -- elliptical, axis aligned');
```



9 Multivariate Gaussian distribution -- independent case

Since z_1 and z_2 are independent we can write out the joint

$$\begin{aligned} p(z_1, z_2) &= p(z_1)p(z_2) \\ &= \frac{1}{\sqrt{2\pi\sigma_1^2}} \frac{1}{\sqrt{2\pi\sigma_2^2}} \exp\left\{-\frac{1}{2\sigma_1^2} z_1^2\right\} \exp\left\{-\frac{1}{2\sigma_1^2} z_2^2\right\} \\ &= \frac{1}{\sqrt{(2\pi)^2\sigma_1^2\sigma_2^2}} \exp\left\{-\frac{1}{2\sigma_1^2} z_1^2 - \frac{1}{2\sigma_2^2} z_2^2\right\} \end{aligned}$$

In fact, for multiple independent Gaussian random variables $z_1, ..., z_n$ the joint is

$$\begin{aligned} p(z_1, \ldots, z_n) &= \prod_i p(z_i) \\ &= (2\pi)^{-n/2} \left(\prod_{i=1}^{n} \sigma_i^2 \right)^{-1/2} \exp\left\{ -\sum_{i=1}^{n} \frac{1}{2\sigma_i^2} z_i^2 \right\} \end{aligned}$$

10 Multivariate Gaussian distribution -- dependent case

Suppose we have n standard random variables (0 mean, unit variance)

$$z_i \sim \mathcal{N}(0,1), \quad i = 1, \ldots n$$

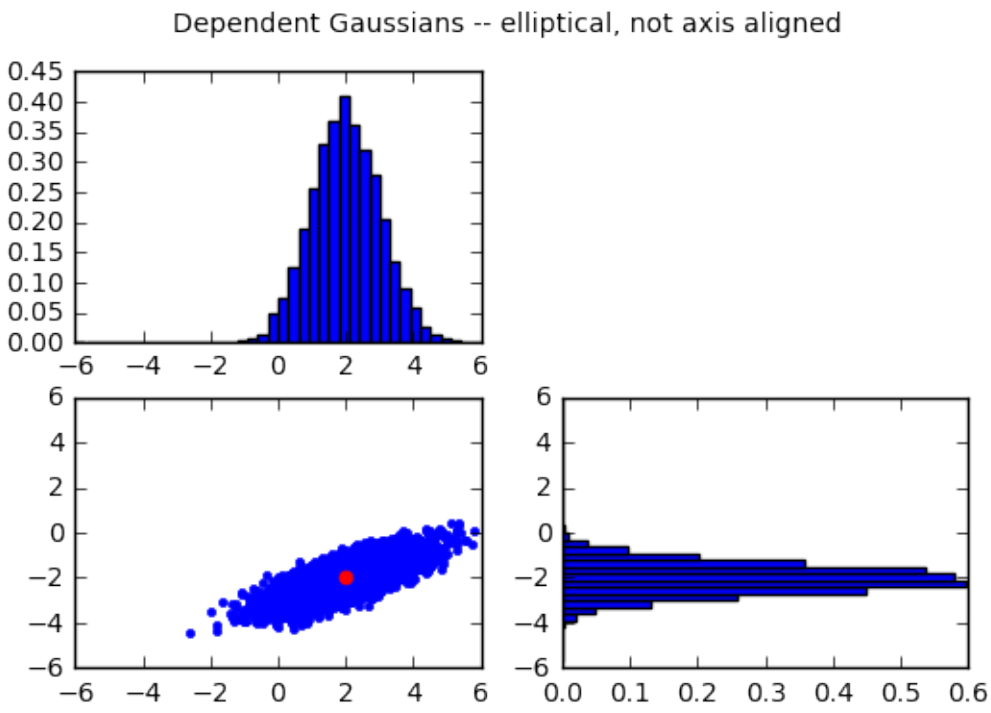
and we are given a vector $\bar{}$ of length n and a full-rank matrix A of size $n \times n$.

What does distribution of $\mathbf{x} = A\mathbf{z} + \bar{}$ look like?

```
In [335]: # independent 0 mean, unit variance
          z1 = numpy.random.randn(10000)
          z2 = numpy.random.randn(10000)

          # matrix 2 x 2
          A = numpy.asarray([[1,-0.2],[0.4,-0.5]])
          # column vector (2 x 1)
          mu = numpy.asarray([[2.0],[-2.0]])

          z = numpy.vstack((z1,z2))
          x = numpy.dot(A,z) + mu

          bounds = [-6, 6]
          plt.subplot(2,2,1)
          plt.hist(x[0,:],range=bounds,bins=40,normed=True)
          plt.subplot(2,2,4)
          plt.hist(x[1,:],range=bounds,bins=40,orientation='horizontal',normed=True)

          plt.subplot(2,2,3)
          plt.plot(x[0,:],x[1,:],'.')
          plt.xlim(bounds)
          plt.ylim(bounds)
          plt.plot(mu[0],mu[1],'r.',markersize=10)
          plt.suptitle('Dependent Gaussians -- elliptical, not axis aligned')
          print "Sample covariance:\n", numpy.cov(x)
          print "True covariance:\n", numpy.dot(A,A.transpose())
```

```
Sample covariance:
[[ 1.04019881  0.50597955]
 [ 0.50597955  0.41709157]]
True covariance:
[[ 1.04  0.5 ]
```

```
[ 0.5   0.41]]
```



11 Multivariate Gaussian distribution -- dependent case

Suppose we have n standard random variables (0 mean, unit variance)

$$z_i \sim \mathcal{N}(0,1), \quad i = 1, \ldots n$$

and we are given a vector $\bar{}$ of length n and a full-rank matrix A of size $n \times n$.

Distribution of $\mathbf{x} = A\mathbf{z} + \mu$ is

$$p(\mathbf{x}) = (2\pi)^{-\frac{n}{2}} (\det \Sigma)^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\mathbf{x} - \bar{})^T \Sigma^{-1} (\mathbf{x} - \bar{})\right\}$$

where $\Sigma = AA^T$.

- μ is **mean** of the Gaussian
- Σ is **covariance** matrix

12 Multivariate Gaussian distributions -- covariance matrices

Typical covariance matrices: * diagonal with constant diagonal -- data looks spherical and axis aligned

$$\Sigma = \sigma^2 I = \text{diag}\left(\sigma \mathbf{1}_n\right) = \begin{bmatrix} \sigma^2 & 0 & \dots & 0 & 0 \\ 0 & \sigma^2 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & \sigma^2 & 0 \\ 0 & 0 & \dots & 0 & \sigma^2 \end{bmatrix}$$

- diagonal with non-constant diagonal -- data looks elliptical and axis aligned

$$\Sigma = \text{diag}\left(\boldsymbol{\sigma}\right) = \begin{bmatrix} \sigma_1^2 & 0 & \dots & 0 & 0 \\ 0 & \sigma_2^2 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & \sigma_{n-1}^2 & 0 \\ 0 & 0 & \dots & 0 & \sigma_n^2 \end{bmatrix}$$

- unconstrained -- data looks elliptical but rotated

13 Maximum likelihood estimates of mean and covariance

Given data $\{\mathbf{x}_i \in \mathbb{R}^n | i = 1, \dots, T\}$ maximum likelihood estimates (MLE) of mean and covariance are:

$$\bar{\mathbf{x}}^{\text{MLE}} = \frac{1}{T} \sum_{i=1}^{T} \mathbf{x}_i$$

$$\Sigma^{\text{MLE}} = \frac{1}{T} \sum_{i=1}^{T} \underbrace{\left(\mathbf{x}_i - \bar{\mathbf{x}}^{\text{MLE}}\right)\left(\mathbf{x}_i - \bar{\mathbf{x}}^{\text{MLE}}\right)^T}_{\text{a matrix of size } n \times n}$$

Dimensionality * $\bar{\mathbf{x}}^{\text{MLE}}$ is of same dimension as a single data point $n \times 1$. * Σ^{MLE} is a matrix of size $n \times n$

Note that $\mathbf{x}\mathbf{x}^T$ and $\mathbf{x}^T\mathbf{x}$ are not the same. Former is a matrix, latter is a scalar.

14 Multivariate Gaussian distribution

Multivariate Gaussian distribution underlies many machine learning techniques: * Linear Discriminant Analysis and variants of Naive Bayes -- classification * Principal Component Analysis and Factor Analysis -- dimensionality reduction * Mixture of Gaussians -- clustering * Kalman filter -- sequential data denoising and prediction * Graphical models for speech, images and video

Hence, we will see this several times over. In due course, we will introduce closed form expressions for constructing marginal, conditional, and joint distribution.

15 Generative vs discriminative approaches to classification

Thus far, we posed classification problems in terms of learning conditional probabilities of labels y given features $\mathbf{x}$

$$p(y|\mathbf{x}, \theta)$$

and we optimized **conditional** log-likelihood

$$\mathcal{LL}(\theta|\mathbf{y}, X) = \sum_i \log \underbrace{p(y_i|\mathbf{x}_i, \theta)}_{\text{conditional probability}}$$

We did not care about how features $\mathbf{x}$ were distributed.

Our aim was to increase probability of labels given features.

This approach to learning is called **discriminative** -- we learn to discriminate between different classes.

16 Generative vs discriminative approaches to classification

Generative models describe all of the data

$$p(y, \mathbf{x}|\theta)$$

and optimize **joint** log-likelihood

$$\mathcal{LL}(\theta|\mathbf{y}, X) = \sum_i \log \underbrace{p(y_i, \mathbf{x}_i|\theta)}_{\text{joint probability}} = \sum_i \left[\log \underbrace{p(y_i|\mathbf{x}_i, \theta)}_{\text{conditional probability}} + \log \underbrace{p(\mathbf{x}_i|\theta)}_{\text{marginal probability}} \right]$$

In this setting, the log-likelihood can be improved by: 1. increasing conditional probability of labels given features $p(y_i|\mathbf{x}_i, \theta)$ 2. increasing probability of features $p(\mathbf{x}_i|\theta)$

However, given such a model we can describe how the data as a whole -- both features and labels -- were generated. This is particulary important if there is missing data.

This approach to learning is called **generative**.

17 Generative models for classification

There are two ways to factorize joint probability of labels and features

$$p(y, \mathbf{x}|\theta) = p(y|\mathbf{x}, \theta)p(\mathbf{x}|\theta) = p(\mathbf{x}|y, \theta)p(y|\theta)$$

The second one given us a simple process to *GENERATE* data:

1. First select label according $p(y|\theta)$, say it was c
2. Now generate features $p(\mathbf{x}|y = c, \theta)$

Once we have such a model we can obtain the conditional probability $p(y|\mathbf{x})$ using Bayes rule

$$p(y = c|\mathbf{x}) = \frac{p(y = c|\theta)p(\mathbf{x}|y = c, \theta)}{\sum_k p(y = k|\theta)p(\mathbf{x}|y = k, \theta)}$$

and we can predict label for a new feature vector $\mathbf{x}$

18 Generative models for classification -- prediction

If we are only interested in predicting the most likely class -- rather than computing probabilities -- we can simplify math a bit by observing

$$p(y=c|\mathbf{x}) = \frac{p(y=c|\theta)p(\mathbf{x}|y=c,\theta)}{\underbrace{\sum_k p(y=k|\theta)p(\mathbf{x}|y=k,\theta)}_{\text{does not depend on c}}}$$

Hence

$$p(y=c|\mathbf{x}) \propto p(y=c|\theta)p(\mathbf{x}|y=c,\theta)$$

and

$$\begin{aligned}\operatorname*{argmax}_c p(y=c|\mathbf{x}) &= \operatorname*{argmax}_c p(y=c|\theta)p(\mathbf{x}|y=c,\theta) \\ &= \operatorname*{argmax}_c \log p(y=c|\theta) + \log p(\mathbf{x}|y=c,\theta)\end{aligned}$$

19 An example of a generative model

Uniform prior on classes

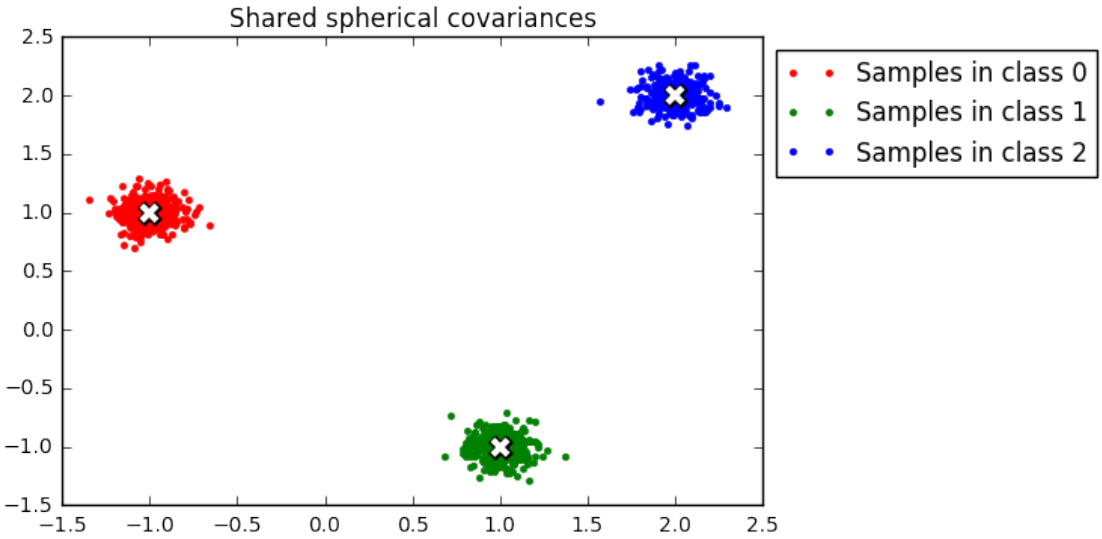
$$p(y=c) = \frac{1}{K}$$

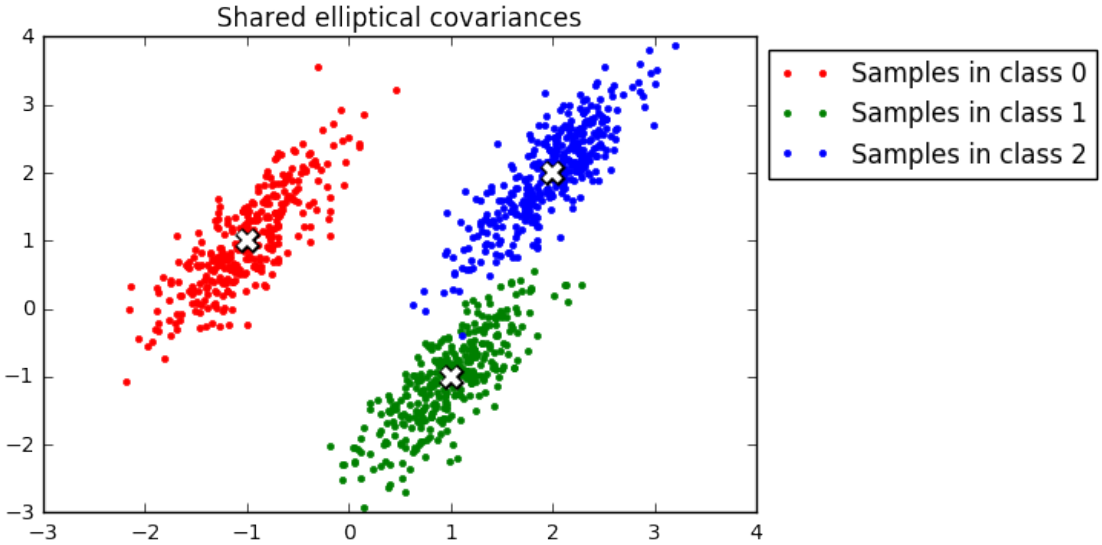
Gaussian distribution of features for each class, different means $\bar{}_c$, but same covariance

$$\mathbf{x}|y=c,\bar{},\sigma \sim \mathcal{N}(\bar{}_c,\Sigma)$$

```
In [334]: import numpy
          import matplotlib.pyplot as plt

          def generate_data(N,d,K,proby,mus,A):
              # This function will generate data from
              # K Gaussians. Each Gaussian has different mean
              # mus[:,c] is mean for the c-th Gaussian
              # Covariance Sigma = A^T A is shared between
              # Gaussians.

              ys = [0]*N
              xs = numpy.zeros((d,N))
              for i in range(N):
                  # Sample class according to the prior p(y)
                  # in this case it is uniform
                  ys[i] = numpy.random.choice(K,1)
                  # Sample feature values according to p(x|y)
                  # In this case, x ~ N(mmu[y[i]],sigma2*I)
                  # To accomplish this, draw z1,z2 ~ N(0,I)
                  z = numpy.random.randn(2,1)
                  # transform by matrix A and shift by class mean
                  x = numpy.dot(A,z) + mus[:,ys[i]]
```

```
        xs[:,i] = x[:,0]
    return xs,ys

def plot_samples(xs,ys,mus=None,colors=['r','g','b']):
    K = numpy.max(ys)+1
    colors = ['r','g','b']
    for c in range(K):
        # indices of samples assigned to class c
        ind = [i for i in range(N) if ys[i]==c]
        plt.plot(xs[0,ind],xs[1,ind],colors[c]+'.',label='Samples in class '+str(c))
        if not mus is None:
            plt.plot(mus[0,c],mus[1,c],'kx',markersize=9,markeredgewidth=5)
            plt.plot(mus[0,c],mus[1,c],'wx',markersize=7,markeredgewidth=3)

    plt.legend(loc=2, bbox_to_anchor=(1,1))

N = 1000 # 100 samples
K = 3   # 3 classes
d = 2   # x is 2d
proby = [1.0/K]*K # [1./3. , 1./3., 1./3.]
mus = 1.0*numpy.asarray([[-1.0,1.0,2],[1.0,-1.0,2.0]])
sigma = 0.1 # standard deviation, variance is 0.01
# A just scales each coordinate by sigma
A = sigma*numpy.eye(2)
xs,ys = generate_data(N,d,K,proby,mus,A)
plot_samples(xs,ys,mus)
plt.title('Shared spherical covariances')

plt.figure()
# this A scales and rotates
numpy.random.seed(2)
A = numpy.random.rand(2,2)
xs,ys = generate_data(N,d,K,proby,mus,A)
plot_samples(xs,ys,mus)
plt.title('Shared elliptical covariances');
```





20 Generative models for classification

In order to specify a generative model we have to specify its parts 1. Prior probability for a label $p(y|\theta)$ 2. Probability for features in each of the classes $p(\mathbf{x}|y=1,\theta), ..., p(\mathbf{x}|y=k,\theta)$.

Q: How would you specify probability of labels? What would be your guess? Can you learn that probability?

Q: Let's say that we have p features and C classes give a guess about the number of parameters need to specify $p(\mathbf{x}|y,\theta)$.

21 Generative models for classification

We will make a simplifying assumptions about feature distribution

$$p(\mathbf{x}|y,\theta) = \prod_j p(x_j|y,\theta)$$

Q: What does this say about features? Recall what it means to be able to factorize $p(x_1, x_2) = p(x_1)p(x_2)$

This assumption underlies the **Naive Bayes** method.

22 Naive Bayes

$$p(y = c|\pi) = \pi_k$$
$$p(\mathbf{x}|y = c,\theta) = \prod_j p(x_j|y = c,\theta_{j,c})$$

Parameters are * π_c prior probability that a sample comes from the class c * $\theta_{j,c}$ parameters for the j^{th} feature for class c

In general, there are many variants of Naive Bayes.

You can choose different distributions for $p(x_j|y = c)$ * Gaussian -- continuous features * Bernoulli -- binary features * Binomial -- count of positive outcomes * Categorical -- discrete features * Multinomial -- count of particular discrete outcomes

23 Naive Bayes with Gaussian features

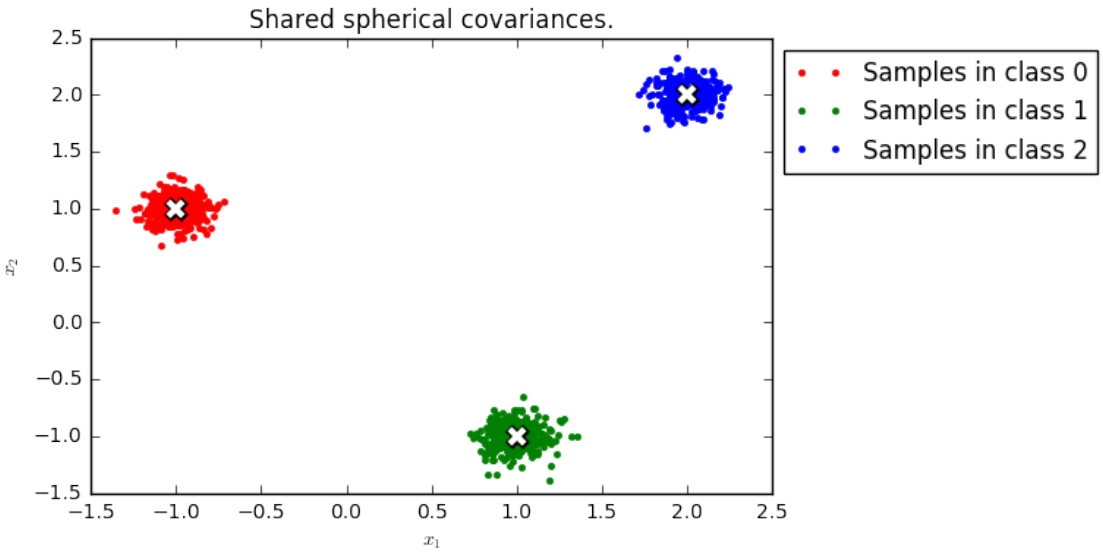
We will assume that

$$x_j|y_c,\theta \sim \mathcal{N}\left(\theta_{j,c},\sigma^2\right)$$

Each feature is Gaussian distributed around class specific mean and with shared spherical variance.

Let's take a look at the data we generated earlier and read-off these parameters.

```
In [333]: N = 1000 # 100 samples
          K = 3   # 3 classes
          d = 2   # x is 2d
          proby = [1.0/K]*K # [1./3. , 1./3., 1./3.]
          mus = 1.0*numpy.asarray([[-1.0,1.0,2],[1.0,-1.0,2.0]])
          sigma = 0.1 # standard deviation, variance is 0.01
          # A just scales each coordinate by sigma
          A = sigma*numpy.eye(2)
          xs,ys = generate_data(N,d,K,proby,mus,A)
          plot_samples(xs,ys,mus)
          plt.title('Shared spherical covariances.');
          plt.xlabel('$x_1$')
          plt.ylabel('$x_2$');
```



24 Learning parameters for Naive Bayes

In the synthetic data examples, we know the ground truth parameters -- after all, we generated the data.

Q: How do we go about learning parameters for a Naive Bayes model?

25 Learning parameters for Naive Bayes

Log-likelihood

$$\begin{aligned}
\mathcal{LL}(\theta, \pi | \mathbf{y}, X) &= \sum_i \log p(y_i, \mathbf{x}_i | \theta, \pi) && \text{definition of likelihood} \\
&= \sum_i \log p(y_i | \pi) + \log p(\mathbf{x}_i | y_i, \theta) && \text{factorization } p(y, \mathbf{x}) = p(y)p(\mathbf{x}|y) \\
&= \sum_i \log p(y_i | \pi) + \log \prod_j p(x_{j,i} | y_i, \theta_j) && \text{Naive Bayes assumption} \\
&= \sum_i \log p(y_i | \pi) + \sum_j \log p(x_{j,i} | y_i, \theta_j)
\end{aligned}$$

Note that we have not yet used our assumptions about distribution of $x_{j,i}$.

26 Learning parameters for Naive Bayes with Gaussian features

Log-likelihood

$$\begin{aligned}\mathcal{LL}(\theta, \pi|\mathbf{y}, X) &= \sum_i \left[\log p(y_i|\pi) + \sum_j \log p(x_{j,i}|y_i, \theta_j)\right] \\ &= \sum_i \left[\log \pi_{y_i} + \sum_j \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(x_{j,i} - \theta_{j,y_i})^2\right\}\right]\end{aligned}$$

Note that parameters π_c and $\theta_{i,c}$ are only used for samples that belong to class c ($y_i = c$)
Hence, we can learn of parameters for each class separately.

27 Learning parameters for Naive Bayes with Gaussian features

Closed form MLE for parameters are

$$\begin{aligned}\pi_k &= \frac{\sum_i [y_i = k]}{\sum_i 1} && \text{frequency of class } k \text{ in training data} \\ \theta_{j,k} &= \frac{\sum_i [y_i = k] x_{i,j}}{\sum_i 1} && \text{average of feature } j \text{ among samples in class } k \\ \sigma &= \frac{\sum_i (x_{j,i} - \theta_{j,y_i})^2}{\sum_i 1} && \text{variance across all features}\end{aligned}$$

We will work this out on board, if there is time. Otherwise, next lecture.

28 Class prediction using Naive Bayes with Gaussian features

Recall that

$$\underset{c}{\text{argmax}}\, p(y = c|\mathbf{x}) = \underset{c}{\text{argmax}} \log p(y = c|\theta) + \log p(\mathbf{x}|y = c, \theta)$$

After a little bit more manipulation

$$\log p(y = k|\mathbf{x}, \theta, \pi) = \log \pi_k - \sum_j \frac{1}{2\sigma^2}(x_{j,i} - \theta_{j,k})^2 + \text{const.}$$

Predicted class

$$y^* = \underset{k}{\text{argmax}} \log \pi_k - \underbrace{\sum_j (x_{j,i} - \theta_{j,k})^2}_{\text{distance to class center}} + \text{const.}$$

29 Today

- An example of online learning
- Linear Transformations review
- Multivariate Gaussian Distribution
- Generative vs Discriminative models
- Naive Bayes