Mixture Model Fitting: an Expectation Maximization approach

Author: **Antonino Ingargiola** (github page (https://github.com/tritemio))

Abstract

In this notebook I describe the fitting of a 2-component mixture sample (i.e. two Gaussians) using the Expectation Maximization (http://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm) approach.

I start showing that the direct minimization of the log-likelihood function is not very roboust and poses convergence problems. Then I introduce the Expectation Maximization method and apply it to the fitting of two gaussians. Finally I consider the heteroscetastic case (samples with different variance). Drawing an analogy with the Weights Least Squares () method (for a 1-compontent population), I propose a "Weighted Expectation Maximization" method that improves the fitting accuracy of a 2-component mixture population.

Introduction

When a sample is drawn from a number k of different distributions we talk of a **Mixture Model (http://en.wikipedia.org/wiki/Mixture_model)**.

In the common case in which the k distributions are Gaussian we talk of Gaussian Mixture Model (GMM) (http://en.wikipedia.org/wiki/Mixture_model#Gaussian_mixture_model).

Fitting a Mixure model, if the number k of ditributions is known, can be done by:

- K-mean (http://en.wikipedia.org/wiki/K-means_clustering) or K-medians (http://en.wikipedia.org/wiki/K-medians_clustering) algorithm. These are clustering algorithms and return only the centroids and the boundaries of the different componets (although variance can be of course computed empirically after clustering).
- Expectation Maximization (EM) (http://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm): finds the Maximum Likelihood (ML) (http://en.wikipedia.org/wiki/Maximum_likelihood) estimated of the model parameters.

In case the number of components k is unknown can be esitmated with the BIC Criterion (http://scikit-learn.org/stable/modules/mixture.html#selecting-the-number-of-components-in-a-classical-gmm).

Another alternative is using the Dirichlet Process GMM (http://scikit-learn.org/stable/modules/mixture.html#dpgmm).

In this notebook we describe the EM method for a GMM distribution with only 2 components.

References

- Estimating Gaussian Mixture Densities with EM – A Tutorial (https://www.cs.duke.edu/courses/spring04/cps196.1/handouts/EM/tomasiEM.pdf) *(PDF)*
- EM for Gaussian Mixtures (http://www.slideshare.net/petitegeek/expectation-maximization-and-gaussian-mixture-models) *(slides)*
- What is the expectation maximization algorithm? (http://www.nature.com/nbt/journal/v26/n8/full/nbt1406.html?pagewanted=all)
- Data Mining Algorithms In R/Expectation Maximization (EM) (http://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Clustering/Expectation_Maximization_%28EM%29)

Existing Python implementations

Python packages implementing EM for GMM:

- scikit-learn (http://scikit-learn.org/stable/modules/mixture.html)
- PyMix (http://www.pymix.org/pymix/index.php?n=PyMix.Tutorial)
- PyPR (http://pypr.sourceforge.net/mog.html)
- PyMC (http://pymc-devs.github.io/pymc/README.html)

Short examples/scripts implementing EM:

- 577735-expectation-maximization (http://code.activestate.com/recipes/577735-expectation-maximization/)

Simple Mixture Model

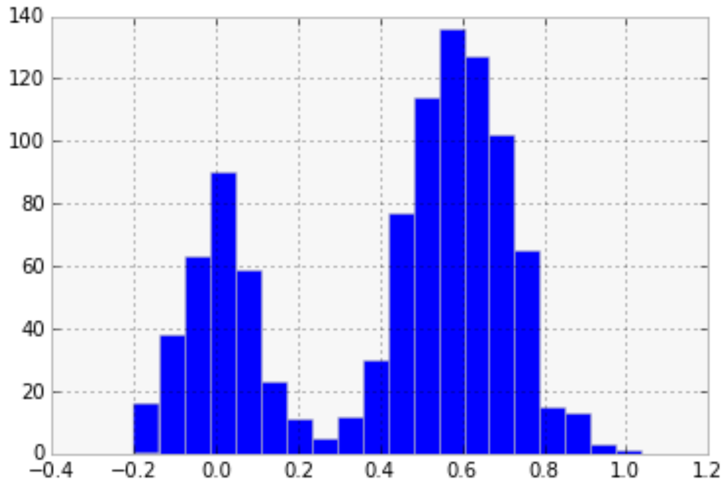
Problem introduction

In [3]:
```
%pylab inline
from numpy import random
from scipy.optimize import minimize, show_options
```

```
Populating the interactive namespace from numpy and matplotlib
```

As a didactical example we want to fit a mixture of two univariate Gaussian distributions. Let call $s = \{s_i\}$ a sample of N elements extracted form the mixture distribution, and $s_1 = \{s_{1i}\}$ and $s_2 = \{s_{2i}\}$ the samples extracted from each single Gaussian distribution. In the following we will assume that s_1 and s_2 are not known.

In [4]:
```
N = 1000
a = 0.3
s1 =  normal(0, 0.08, size=N*a)
s2 = normal(0.6,0.12, size=N*(1-a))
s = concatenate([s1,s2])
hist(s, bins=20);
```



The model for this sample is the linear combination of two Gaussian PDF:

$$f(x|p) = \frac{\pi_1}{\sigma_1\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu_1)^2}{2\sigma_1^2}\right\} + \frac{\pi_2}{\sigma_2\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu_2)^2}{2\sigma_2^2}\right\}$$

where $p = [\mu_1, \sigma_1, \mu_2, \sigma_2, \pi_1]$. Note that π_2 is not included in p since $\pi_2 = 1 - \pi_1$.

In python we can define $f(x|p)$ using `normpdf()` implemented by *Numpy*:

In [5]:
```
def pdf_model(x, p):
    mu1, sig1, mu2, sig2, pi_1 = p
    return pi_1*normpdf(x, mu1, sig1) + (1-pi_1)*normpdf(x, mu2, sig
2)
```

This function will be used in the following sections.

Maximum Likelihood: direct maximization

NOTE

This section illustrate the direct Maximum Likelihood method (i.e. direct minimization of the -log-likelihood function). The purpose is to show that a direct minimization is much more complex and less robust than the EM algorithm. You can skip this section if you are already conviced and you want to read about the EM algorithm.

If we try to apply the the **Maximum Likelihood (ML)** method directly, we find that the log-likelihood function is quite difficult to minimize numerically.

Given a sample $s = \{s_i\}$ of size N extracted from the mixture distribution, the likelihood function is

$$\mathcal{L}(p, s) = \prod_i f(s_i | p)$$

and the log-likelihood function is:

$$\ln \mathcal{L}(p, s) = \sum_i \ln f(s_i | p)$$

Now, since $f(\cdot)$ is the sum of two terms, the term $\log f(s_i | p)$ can't be simplified (it's the log of a sum). So for each s_i we must compute the log of the sum of two exponetial. It's clear that not only the computation will be slow but also the numerical errors will be amplified. Moreover, often the likelihood function has local maxima other than the global one (in other terms the function is not convex).

In python the log-likelihood function can be defined as:

```
In [6]: def log_likelihood_two_1d_gauss(p, sample):
            return -log(pdf_model(sample, p)).sum()
```

If we try to minimize it starting with a close-to-real intial guess $p_0 = [-0.2, 0.2, 0.8, 0.2, 0.5]$ we fail with the most common methods.

Python NOTES:

- Here we use the function `minimize()` from `scipy.optimization`. This function allows to choose several minimization methods. The list of (currently) available minimization methods is `'Nelder-Mead'` *(simplex)*, `'Powell'`, `'CG'`, `'BFGS'`, `'Newton-CG'`,> `'Anneal'`, `'L-BFGS-B'` *(like BFGS but bounded)*, `'TNC'`, `'COBYLA'`, `'SLSQPG'`.

  The documetation can be found here (http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize).

In [7]:
```
# Initial guess
p0 = array([-0.2,0.2,0.8,0.2,0.5])
```

In [8]:
```
# Minimization 1
res = minimize(log_likelihood_two_1d_gauss, x0=p0, args=(s,), method
='BFGS')
#res # NOT CONVERGED
```

In [10]:
```
# Minimization 2
res = minimize(log_likelihood_two_1d_gauss, x0=p0, args=(s,), method
='powell',
        options=dict(maxiter=10e3, maxfev=2e4))
#res # NOT CONVERGED
```

In [11]:
```
# Minimization 3
res = minimize(log_likelihood_two_1d_gauss, x0=p0, args=(s,), method
='Nelder-Mead',
        options=dict(maxiter=10e3, maxfev=2e4))
res
```

Out[11]:
```
  status: 0
    nfev: 428
 success: True
     fun: -191.90433284079629
       x: array([ 0.00650312,  0.08413543,  0.59445438,  0.12201147,
 0.30045492])
 message: 'Optimization terminated successfully.'
     nit: 262
```

In [12]:
```
res.x
```

Out[12]:
```
array([ 0.00650312,  0.08413543,  0.59445438,  0.12201147,  0.3004549
2])
```

In [13]:
```
# Minimization 4
res = minimize(log_likelihood_two_1d_gauss, x0=p0, args=(s,), method
='L-BFGS-B',
    bounds=[(-0.5,2),(0.01,0.5),(-0.5,2),(0.01,0.5),(0.01,0.99)])
res
```

Out[13]:
```
  status: 0
 success: True
    nfev: 54
     fun: -191.90437529910372
       x: array([ 0.59445438,  0.12199155,  0.00648396,  0.08411654,
 0.69951773])
 message: 'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     jac: array([ 0.00275406,  0.00739533, -0.00014779, -0.00086118,
 0.00196962])
     nit: 31
```

In [14]:
```
# Finds additional options for the different solvers:
#show_options('minimize', 'powell')
```

Executing the minimizations we see that we have convergence only using the `'Nelder-Mead'` (simplex) and the `'L-BFGS-B'`. methods. The latter is much ~10x faster as requires only 28 iteration and 40 function evaluation (instead of 256 iterations and 413 function evaluations). Refer to the **Scipy** documentation for a description of the methods (http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize).

Expectation Maximization

Introduction

The EM method allows to fit a statistical model in the case where the experimental data has unknown (latent) variables. In the context of a mixture-model fitting we can assume that the latent variables "tell" which component has generated each sample.

With the EM method, we first "assign" each sample to each components of the distribution. After that, we can compute MLE estimators of parameters of each component of the mixture. For Gaussian components the MLE will be basically the empirical mean and variance.

It is possible to do the assigment choosing, for each sample, the component that has the highest probability to generate the sample (**hard assignment**).

Alternatively it's possible to compute, for each sample s_i, the "fraction" of s_i generated by each component (**soft assignment**).

In our example, for each sample s_i we have two coefficients $\gamma(i,1)$ and $\gamma(i,2)$ that represent the fraction of s_i that belongs to (respectively) component 1 and 2. The sum $\gamma(i,1)+\gamma(i,2)=1$. $\gamma()$ is called in litterature **responsibility fuction**. This soft assignment method is the one commonly used in the context of EM algorithms for mixture model fitting.

Only the **soft assignment** scheme will be used is the following sections.

Algorithm

Starting from the PDF $f_1()$ and $f_2()$ of the single components:

$$f_1(x|p) = \frac{1}{\sigma_1\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu_1)^2}{2\sigma_1^2}\right\} \qquad f_2(x|p) = \frac{1}{\sigma_2\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu_2)^2}{2\sigma_2^2}\right\}$$

the mixture PDF is:

$$f(x|p) = \pi_1 f_1(x|p) + \pi_2 f_2(x|p)$$

If we know (or guess initially) the parameters p, we can compute for each sample and each component the **responsibility function** defined as:

$$\gamma(i,k) = \frac{\pi_k f_k(s_i|p)}{f(s_i|p)}$$

and starting from the "effective" number of samples for each category (N_k) we can compute the "new" estimation of parameters:

$$N_k = \sum_{i=1}^{N} \gamma(i,k) \qquad\qquad k=1,2 \quad (\text{note that} \quad N_1 + N_2 = N)$$

$$\mu_k^{new} = \frac{1}{N_k}\sum_{i=1}^{N} \gamma(i,k)\cdot s_i$$

$$\sigma_k^{2\,new} = \frac{1}{N_k}\sum_{i=1}^{N} \gamma(i,k)\cdot (s_i - \mu_k^{new})^2$$

$$\pi_k^{new} = \frac{N_k}{N}$$

Now we just loop

1. recompute $\gamma(i,k)$
2. estimate updated parameters

until convergence.

REMARKS

- There is no minimization! It's just an iterative algorithm that theory guarantee to converge to a (local) minimum.
- We don't even write the likelihood function but we obtain a ML estimation.
- Even if the components of the mixture are not Gaussian, the method works as far as it's possible to determine the distribution parameters from empirical moments. For example:
  - Poisson distribution, MLE:

$$\hat{\mu} = \frac{1}{N}\sum s_i$$

  - Binomial distribution, MLE:

$$\hat{p} = \frac{N_{success}}{N}$$

Implementation

Let implement this EM algorithm in python. For simplicity the iteration is stopped only after a fixed number of iteration:

In [15]:
```
max_iter = 100

# Initial guess of parameters and initializations
p0 = array([-0.2,0.2,0.8,0.2,0.5])
mu1, sig1, mu2, sig2, pi_1 = p0
mu = array([mu1, mu2])
sig = array([sig1, sig2])
pi_ = array([pi_1, 1-pi_1])

gamma = zeros((2, s.size))
N_ = zeros(2)
p_new = p0

# EM loop
counter = 0
converged = False
while not converged:
    # Compute the responsibility func. and new parameters
    for k in [0,1]:
        gamma[k,:] = pi_[k]*normpdf(s, mu[k], sig[k])/pdf_model(s, p
_new)
        N_[k] = 1.*gamma[k].sum()
        mu[k] = sum(gamma[k]*s)/N_[k]
        sig[k] = sqrt( sum(gamma[k]*(s-mu[k])**2)/N_[k] )
        pi_[k] = N_[k]/s.size
    p_new = [mu[0], sig[0], mu[1], sig[1], pi_[0]]
    assert abs(N_.sum() - N)/float(N) < 1e-6
    assert abs(pi_.sum() - 1) < 1e-6

    # Convergence check
    counter += 1
    converged = counter >= max_iter
```

In [16]:
```
print "Means:   %6.3f  %6.3f" % (p_new[0], p_new[2])
print "Std dev: %6.3f  %6.3f" % (p_new[1], p_new[3])
print "Mix (1): %6.3f " % p_new[4]
```

```
Means:    0.006   0.594
Std dev:  0.084   0.122
Mix (1):  0.300
```

In [17]:
```
print pi_.sum(), N_.sum()
```

```
1.0 1000.0
```

In [18]:
```
res.x
```

Out[18]:
```
array([ 0.59445438,  0.12199155,  0.00648396,  0.08411654,  0.6995177
3])
```

Heteroscetastic Mixture Model

Introduction

In the first examples the samples are extracted from a mixture of two gaussians. The samples of each gaussian component are i.i.d. (http://en.wikipedia.org/wiki/Independent_and_identically_distributed_random_variables) or, in other words each component is homoscedastic (http://en.wikipedia.org/wiki/Homoscedasticity).

In some cases, the components of the mixture are not **homoscedastic** but **heteroscedastic (http://en.wikipedia.org/wiki/Heteroscedasticity)**. This means that within a single component/population of the mixture each sample has a different variance. The variance is assumed to be known.

Non-mixture case: Weighted Least Squares

Let consider first the case of a single population $s = \{s_i\}$ (no mixture), in which all the samples have same mean μ but different variances σ_i^2. Assuming that the mean square error is **normally distributed** with variance σ_i, than the **Weighted Least Square** (http://en.wikipedia.org/wiki/Weighted_least_squares#Weighted_least_squares) method allows to obtain a **MLE** of the mean:

$$\hat{\mu}_{ML} = \frac{\sum_i w_i \cdot s_i}{\sum_i w_i} \qquad \text{and} \qquad w_i = \frac{1}{\sigma_i^2}$$

We can still compute the **mean square error** of the samples:

$$\sigma_m^2 = \frac{1}{N} \sum_i (s_i - \hat{\mu}_{ML})^2$$

but it will now have the meaning of "distribution parameter" (i.e. the variance) like in the case of i.i.d. gaussian samples. Nonetheless σ_m^2 provides descriptive (https://en.wikipedia.org/wiki/Descriptive_statistics) information about the sample variability (or dispersion).

We may also introduce other measures of variability by weighting the errors in different ways:

$$\sigma_w^2 = \frac{\sum_i w_i (s_i - \hat{\mu}_{ML})^2}{\sum_i w_i}$$

For example let say we want to compare the variability of the two population. An higher variability will mean necessary higher error in the estimator $\hat{\mu}$.

But if we compute $\hat{\mu}_{ML}$ weighting each sample, then samples with high variance will affect less the average than samples with small variance. Therefore if we want to estimate the "accuracy" (or simply the variance) of $\hat{\mu}_{ML}$ then we need to weight **less** the **mean square errors** of samples with high variance. In this context seems natural to use $w_i = 1/\sigma_i^2$ also to compute σ_w^2.

> **NOTE** A practical consideration is due here. If the sample population is large enough then a small variation in the sample dispersion will not significantly affect $\hat{\mu}$. In fact, the fitting error it's already small (due to the high # samples). If the theorethical fitting error is below a (application-specific) "significativity threshold" then the effect of a "small" variance difference beween two population becomes completely negligible. In this case other effects (for example model mismatch) will dominate the fitting error.

Mixture case: Weighted Expectation Maximization

To generalize the EM method in case of samples with different variance we can apply different schemes.

$$\gamma(i,k) = \frac{\pi_k f_k(s_i|p)}{f(s_i|p)}\, w_i$$

N_k, μ_k^{new}, $\sigma_k^{2\,new}$, π_k^{new} computed as the no-weight case (but using the new definition of γ).

Requirement: $\sum_i w_i = N$

NOTE In this scheme we **weight more** samples with lower variance. They contribute more to $N_k = \sum_i \gamma(k,i)$ (thus $\pi_k = N_k/N$). Also μ_k^{new} and to $\sigma_k^{2\,new}$ are "weighted means": we simply redistribute the weights including also the information on the variance. The effect on $\sigma_k^{2\,new}$ is that the contribution of high variance samples is less, so the extimated variance is smaller. This helps to resolve better the sub-populations. The reduction in the estimated variance is "consistent" (not too big) because it is a weighted mean: a sample with high variance "weights" less in the sum, but also contribute less to the sum of weights (i.e. the normalization of the weighted mean).

- **Question**: $N_1 + N_2 = N$?
  - YES:
    1. We set as requirement: $\sum_i w_i = N$.
    2. From the N_k definition: $N_1 + N_2 = \sum_{i,k} \gamma(i,k)$.
    3. For each i: $\gamma(i,1) + \gamma(i,2) = w_i$ *(this sum is 1 in the no-weight case)*
    4. $\Rightarrow \quad N_1 + N_2 = \sum_{i,k} \gamma(i,k) = \sum_i w_i = N$
- **Question**: $\pi_1 + \pi_2 = 1$?
  - YES: Since $\pi_k = N_k/N$ and $N_1 + N_2 = N$,
    $\Rightarrow \quad \pi_1 + \pi_2 = N_1/N + N_2/N = 1$

SCHEME 2

Here we compute the mean μ_k^{new} exactly as in SCHEME1, "reweighting" the weights with w_i (here the equation is explicit, in SCHEME1 we included w_i in γ):

$$\mu_k^{new} = \frac{\sum_{i=1}^{N} w_i \gamma(i,k) \cdot s_i}{N_k \sum_i w_i}$$

No other modification is done compared to the no-weight case

NOTES

This approach is less "invasive" and has the advantage to give real empirical variance (or mean square error) of each sub-population. In contrast with the no-weight case this apprach gives better estimate of the mean. However (like the no-weight) is less efficient (http://en.wikipedia.org/wiki/Efficiency_(statistics)). In ohter words it uses "less" information than SCHEME1 (and requires more iterations to converge).

WARNING This method has serious converge problems if initial guess is not very close to the real value.

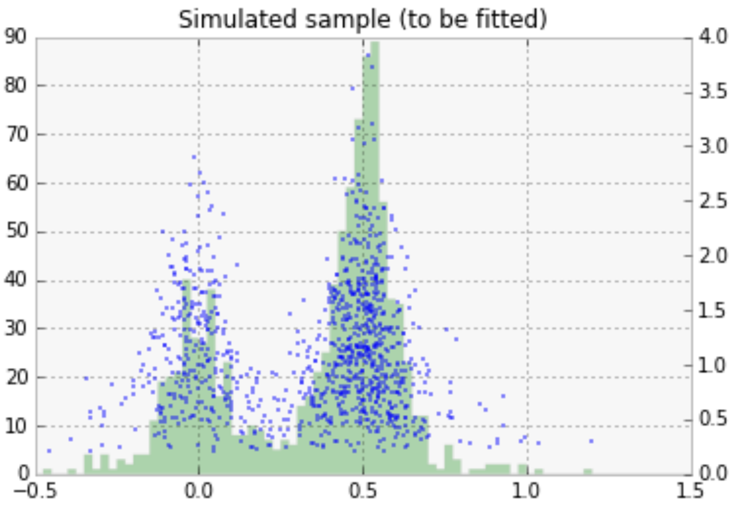
The code

First we define a functions to generate the sample populations:

In [1]:
```
from scipy.stats import expon

def sim_single_population(mu, N=1000, max_sigma=0.5, mean_sigma=0.08
):
    """Extract samples from a normal distribution
    with variance distributed as an exponetial distribution
    """
    exp_min_size = 1./max_sigma**2
    exp_mean_size = 1./mean_sigma**2
    sigma = 1/sqrt(expon.rvs(loc=exp_min_size, scale=exp_mean_size,
size=N))
    return normal(mu, scale=sigma, size=N), sigma
```

In [45]:
```
N = 1000
a = 0.3
s1, sig1 = sim_single_population(0, N=N*a)
s2, sig2 = sim_single_population(0.5, N=N*(1-a))
s = concatenate([s1, s2])
sigma_tot = concatenate([sig1, sig2])
hist(s, bins=r_[-1:2:0.025], alpha=0.3, color='g', histtype='stepfil
led');
ax = twinx(); ax.grid(False)
ax.plot(s, 0.1/sigma_tot, 'o', mew=0, ms=2, alpha=0.6, color='b')
xlim(-0.5, 1.5); title('Simulated sample (to be fitted)')
print "Means:   %6.3f  %6.3f" % (s1.mean(), s2.mean())
print "Std dev: %6.3f  %6.3f" % (sqrt((sig1**2).mean()), sqrt((sig2*
*2).mean()))
print "Mix (1): %6.3f " % a
```

```
Means:    -0.007   0.496
Std dev:  0.144   0.146
Mix (1):  0.300
```



Then we use again the EM algoritm (with optional addition of weights) and print/plot the results:

In [61]:
```
max_iter = 300
weights = 1./sigma_tot**2

# Renormalizing the weights so they sum to N
weights *= 1.*weights.size/weights.sum()

# No weights case
#weights = ones(s.size)

# Initial guess of parameters and initializations
p0 = array([-0.05,0.1,0.6,0.1,0.5])
mu1, sig1, mu2, sig2, pi_1 = p0
mu = array([mu1, mu2])
sig = array([sig1, sig2])
pi_ = array([pi_1, 1-pi_1])

gamma = zeros((2, s.size))
N_ = zeros(2)
p_new = p0

# EM loop
counter = 0
converged = False
while not converged:
    # Compute the responsibility func. and new parameters
    for k in [0,1]:
        gamma[k,:] = weights*pi_[k]*normpdf(s, mu[k], sig[k])/pdf_mo
del(s, p_new) # SCHEME1
        #gamma[k,:] = pi_[k]*normpdf(s, mu[k], sig[k])/pdf_model(s,
p_new)         # SCHEME2
        N_[k] = gamma[k,:].sum()
        mu[k] = sum(gamma[k]*s)/N_[k] # SCHEME1
        #mu[k] = sum(weights*gamma[k]*s)/sum(weights*gamma[k]) # SCH
EME2
        sig[k] = sqrt( sum(gamma[k]*(s-mu[k])**2)/N_[k] )
        pi_[k] = 1.*N_[k]/N
    p_new = [mu[0], sig[0], mu[1], sig[1], pi_[0]]
    assert abs(N_.sum() - N)/float(N) < 1e-6
    assert abs(pi_.sum() - 1) < 1e-6

    # Convergence check
    counter += 1
    converged = counter >= max_iter
```

In [55]:
```
print ">> NO WEIGHTS"
print "Means:   %6.3f  %6.3f" % (p_new[0], p_new[2])
print "Std dev: %6.3f  %6.3f" % (p_new[1], p_new[3])
print "Mix (1): %6.3f " % p_new[4]
```

```
>> NO WEIGHTS
Means:   -0.010   0.509
Std dev:  0.126   0.113
Mix (1):  0.315
```

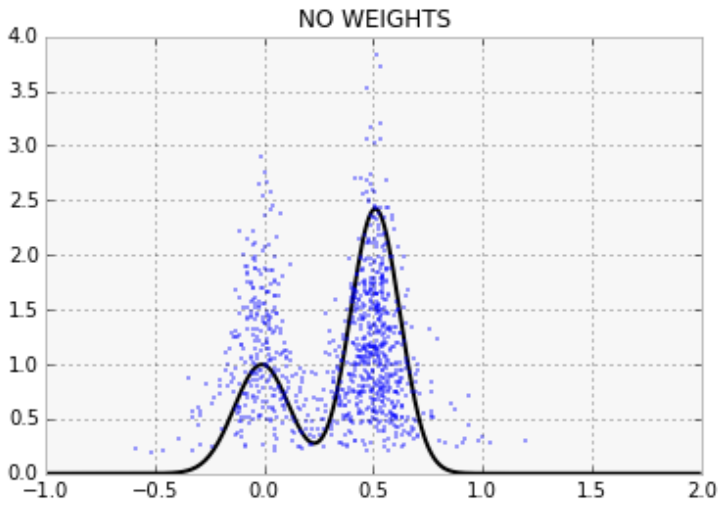
In [51]:
```
print ">> WEIGHTED SCHEME1"
print "Means:   %7.4f  %7.4f" % (p_new[0], p_new[2])
print "Std dev: %7.4f  %7.4f" % (p_new[1], p_new[3])
print "Mix (1): %7.4f " % p_new[4]
```

```
>> WEIGHTED SCHEME1
Means:   -0.0105   0.5035
Std dev:  0.0748   0.0724
Mix (1):  0.2850
```

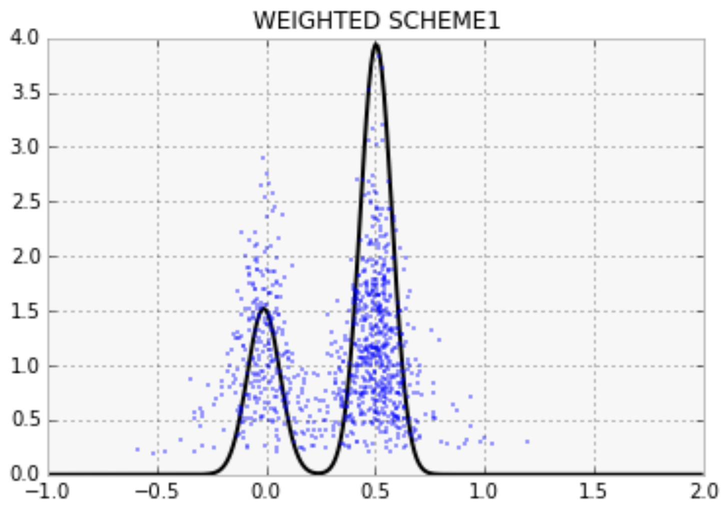
In [53]:
```
print ">> WEIGHTED SCHEME2"
print "Means:   %6.3f  %6.3f" % (p_new[0], p_new[2])
print "Std dev: %6.3f  %6.3f" % (p_new[1], p_new[3])
print "Mix (1): %6.3f " % p_new[4]
```

```
>> WEIGHTED SCHEME2
Means:   -0.009   0.503
Std dev:  0.125   0.113
Mix (1):  0.314
```

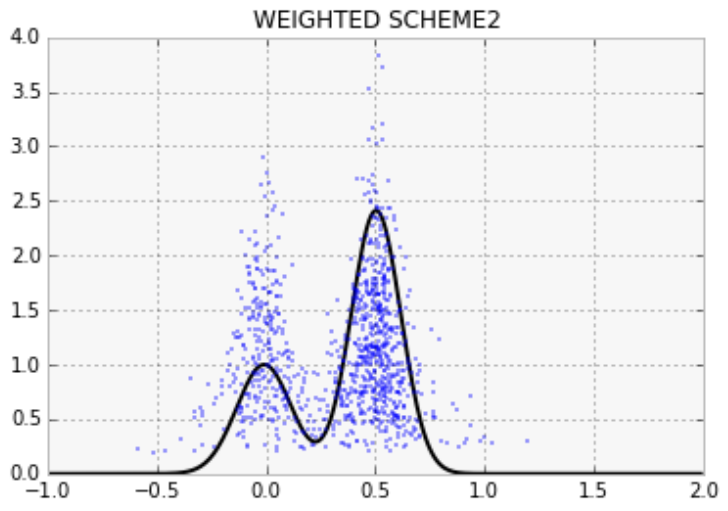
In [56]:
```
title('NO WEIGHTS')
#hist(s, bins=r_[-1:2:0.05], normed=True);
x = r_[-1:2:0.01]
plot(x, pdf_model(x, p_new), color='k', lw=2); grid(True)
plot(s, 0.1/sigma_tot, 'o', mew=0, ms=2, alpha=0.5);
```



In [58]:
```
title('WEIGHTED SCHEME1')
#hist(s, bins=r_[-1:2:0.05], normed=True);
x = r_[-1:2:0.01]
plot(x, pdf_model(x, p_new), color='k', lw=2); grid(True)
plot(s, 0.1/sigma_tot, 'o', mew=0, ms=2, alpha=0.5);
```



In [60]:
```
title('WEIGHTED SCHEME2')
#hist(s, bins=r_[-1:2:0.05], normed=True);
x = r_[-1:2:0.01]
plot(x, pdf_model(x, p_new), color='k', lw=2); grid(True)
plot(s, 0.1/sigma_tot, 'o', mew=0, ms=2, alpha=0.5);
```



Compare EM to other methods

In [63]:
```
%pylab inline
from scipy.stats import poisson, expon, binom
from scipy.optimize import minimize, leastsq
from scipy.special import erf
```

```
Populating the interactive namespace from numpy and matplotlib

WARNING: pylab import has clobbered these variables: ['poisson']
`%pylab --no-import-all` prevents importing * from pylab and numpy
```

It seems natural to empirically bechmark the performance of the EM algorithm compared to other common fitting methods. The most simple method is the histogram fitting. Another method is curve-fitting the empirical CDF (ECDF (http://en.wikipedia.org/wiki/Empirical_distribution_function)) which does not require binning.

Let define the fitting function we want to compare:

In [64]:
```
def fit_two_peaks_EM(sample, sigma, weights=False, p0=array([0.1,0.2
,0.6,0.2,0.5]),
        max_iter=300, tollerance=1e-3):

    if not weights: w = ones(sample.size)
    else: w = 1./(sigma**2)
    w *= 1.*w.size/w.sum() # renormalization so they sum to N

    # Initial guess of parameters and initializations
    mu = array([p0[0], p0[2]])
    sig = array([p0[1], p0[3]])
    pi_ = array([p0[4], 1-p0[4]])

    gamma, N_ = zeros((2, sample.size)), zeros(2)
    p_new = array(p0)
    N = sample.size

    # EM loop
    counter = 0
    converged, stop_iteration = False, False
    while not stop_iteration:
        p_old = p_new
        # Compute the responsibility func. and new parameters
        for k in [0,1]:
            gamma[k,:] = w*pi_[k]*normpdf(sample, mu[k], sig[k])/pdf
_model(sample, p_new) # SCHEME1
            #gamma[k,:] = pi_[k]*normpdf(sample, mu[k], sig[k])/pdf_
model(sample, p_new) # SCHEME2
            N_[k] = gamma[k,:].sum()
            mu[k] = sum(gamma[k]*sample)/N_[k] # SCHEME1
            #mu[k] = sum(w*gamma[k]*sample)/sum(w*gamma[k]) # SCHEME
2
            sig[k] = sqrt( sum(gamma[k]*(sample-mu[k])**2)/N_[k] )
            pi_[k] = 1.*N_[k]/N
        p_new = array([mu[0], sig[0], mu[1], sig[1], pi_[0]])
        assert abs(N_.sum() - N)/float(N) < 1e-6
        assert abs(pi_.sum() - 1) < 1e-6

        # Convergence check
        counter += 1
        max_variation = max((p_new-p_old)/p_old)
        converged = True if max_variation < tollerance else False
        stop_iteration = converged or (counter >= max_iter)
    #print "Iterations:", counter
    if not converged: print "WARNING: Not converged"
    return p_new
```

In [65]:
```
def fit_two_gauss_mix_hist(s, bins=r_[-0.5:1.5:0.001],
weights=None,
            p0=[0,0.1,0.5,0.1,0.5]):
    """Fit the (optionally weighted) histogram with a 2-Gaussian
mix PDF.
    """
    H = histogram(s, bins=bins, weights=weights, normed=True)
    x = .5*(H[1][1:]+H[1][:-1])
    y = H[0]
    assert x.size == y.size

    err_func = lambda p, x_, y_: (y_ - pdf_model(x_, p))
    res = leastsq(err_func, x0=p0, args=(x,y), full_output=1)
    #print res
    return array(res[0])
```

In [66]:
```
def fit_two_gauss_mix_cdf(s, p0=[0.2,1,0.8,1,0.3], weights=None):
    """Fit the sample s with two gaussians.
    """
    ## Empirical CDF
    ecdf = [sort(s), arange(0.5,s.size+0.5)*1./s.size]

    ## CDF for a Gaussian distribution, and for a 2-Gaussian mix
    gauss_cdf = lambda x, mu, sigma: 0.5*(1+erf((x-mu)/(sqrt(2)*sigm
a)))
    two_gauss_cdf = lambda x, m1, s1, m2, s2, a:\
            a*gauss_cdf(x,m1,s1)+(1-a)*gauss_cdf(x,m2,s2)

    ## Fitting the empirical CDF
    fit_func = lambda p, x: two_gauss_cdf(x, *p)
    err_func = lambda p, x, y: fit_func(p, x) - y
    p,v = leastsq(err_func, x0=p0, args=(ecdf[0],ecdf[1]))
    return array(p)
```

A simple function to draw samples from a mixture of 2 Gaussians:

In [67]:
```
def sim_two_gauss_mix(p, N=1000):
    s1 =  normal(p[0], p[1], size=N*p[4])
    s2 = normal(p[2], p[3], size=N*(1-p[4]))
    s = concatenate([s1,s2])
    return s
```

Now define a function perform the fitting an **high number** populations:

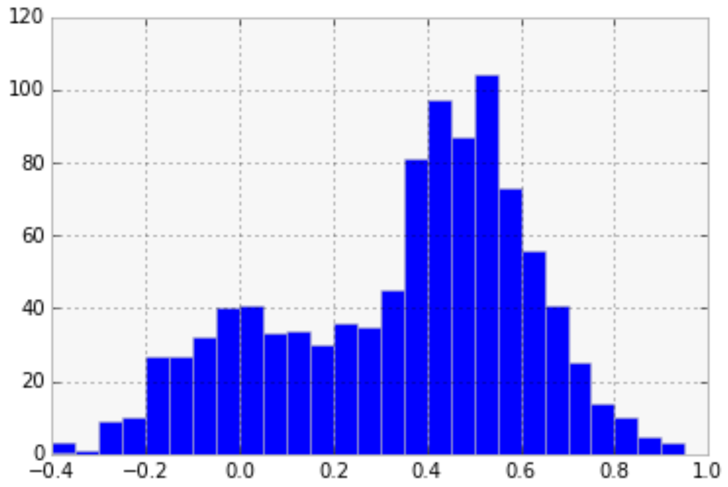
In [68]:
```
def test_accuracy(N_test=200, **kwargs):

    fit_kwargs = dict()
    if 'p0' in kwargs:
        fit_kwargs.update(p0=kwargs.pop('p0'))
    if 'max_iter' in kwargs:
        fit_kwargs.update(max_iter=kwargs.pop('max_iter'))

    pop_kwargs = dict()
    pop_kwargs.update(kwargs)

    P_em = zeros((N_test, 5))
    P_h = zeros((N_test, 5))
    P_cdf = zeros((N_test, 5))
    for i_test in xrange(N_test):
        s = sim_two_gauss_mix(**pop_kwargs)

        P_em[i_test,:] = fit_two_peaks_EM(s, sigma=None, weights=Fal
se, **fit_kwargs)
        if 'max_iter' in fit_kwargs: fit_kwargs.pop('max_iter')
        P_h[i_test,:] = fit_two_gauss_mix_hist(s, bins=r_[-0.5:1.5:0
.01], weights=None, **fit_kwargs)
        P_cdf[i_test,:] = fit_two_gauss_mix_cdf(s)

    return P_em, P_h, P_cdf
```

In [72]:
```
def plot_accuracy(P_LIST, labels, name="Fit comparison", ip=0,
                  bins=(r_[0.2:0.5:0.004]+0.002)):
    print "METHOD\t   MEAN  STD.DEV."
    for P, label in zip(P_LIST, labels):
        hist(P[:,ip], bins=bins, histtype='stepfilled', alpha=0.5, l
abel=label);
        print "%s:\t %6.2f %6.2f" % (label, P[:,ip].mean()*100, P[:,
ip].std()*100)
    legend(); grid(True); title(name);
```

Study of different populations

CASE 1: Half-mixed populations

In [70]:
```
s = sim_two_gauss_mix(N=1000, p=[0,0.15,0.5,0.15,0.3])
hist(s, bins=r_[-0.4:1:0.05]);
```

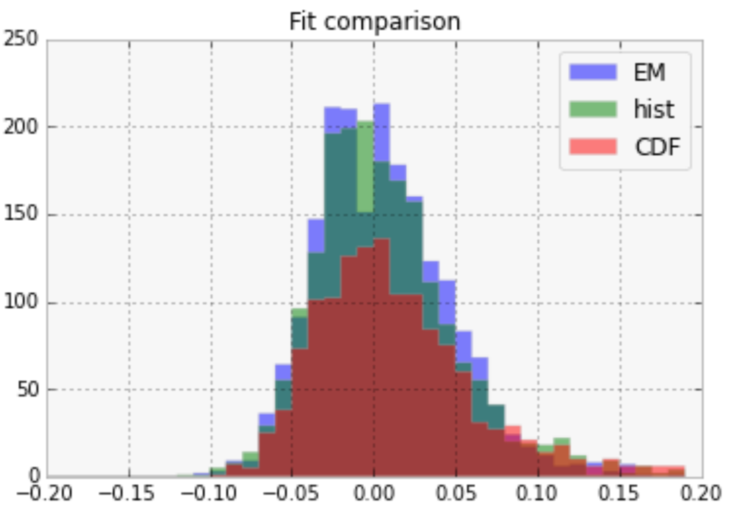


In [71]:
```
P_em, P_h, P_cdf = test_accuracy(N_test=2000, N=200, p=[0,0.15,0.5,0
.15,0.3],
                                                     p0=[0,0.1,0.6,0.1,0.5]
)
```

```
-c:37: RuntimeWarning: divide by zero encountered in divide
/home/anto/.local/lib/python2.7/site-packages/scipy/optimize/minpack.
py:402: RuntimeWarning: Number of calls to function has reached maxfe
v = 1200.
  warnings.warn(errors[info][0], RuntimeWarning)
```

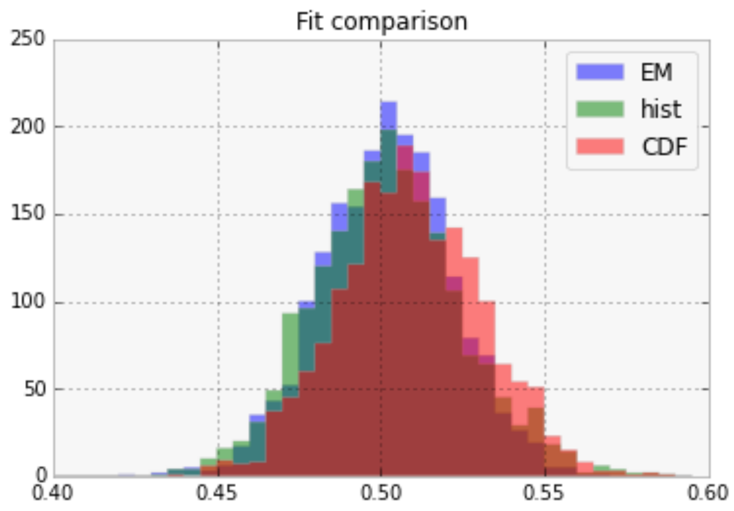
In [73]:
```
plot_accuracy([P_em, P_h, P_cdf], labels=['EM', 'hist', 'CDF'], ip=0
,
              bins=(r_[-.2:.2:0.01]))
```

```
METHOD      MEAN  STD.DEV.
EM:         0.64   4.17
hist:       1.79   6.78
CDF:       10.04  14.28
```



In [74]:
```
plot_accuracy([P_em, P_h, P_cdf], labels=['EM', 'hist', 'CDF'], ip=2
,
                    bins=(r_[.4:.6:0.005]))
```

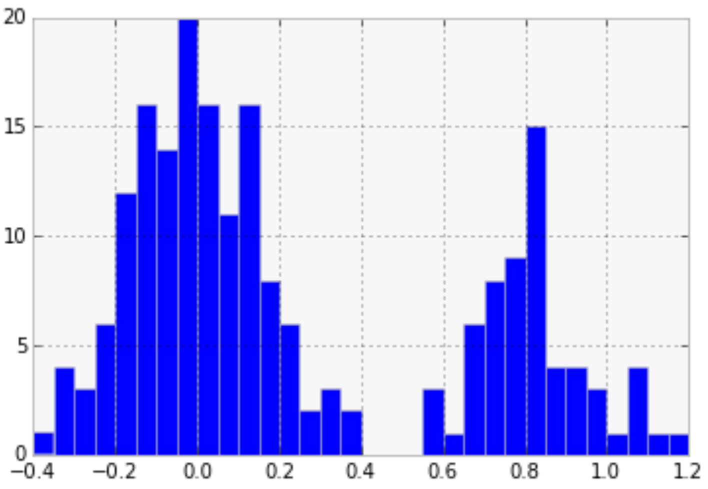
```
METHOD      MEAN  STD.DEV.
EM:        50.20   2.00
hist:      50.24   2.31
CDF:       48.48  10.96
```



> **EM** is better both for bias and accuracy (CDF is worst)

CASE 2: Separated populations

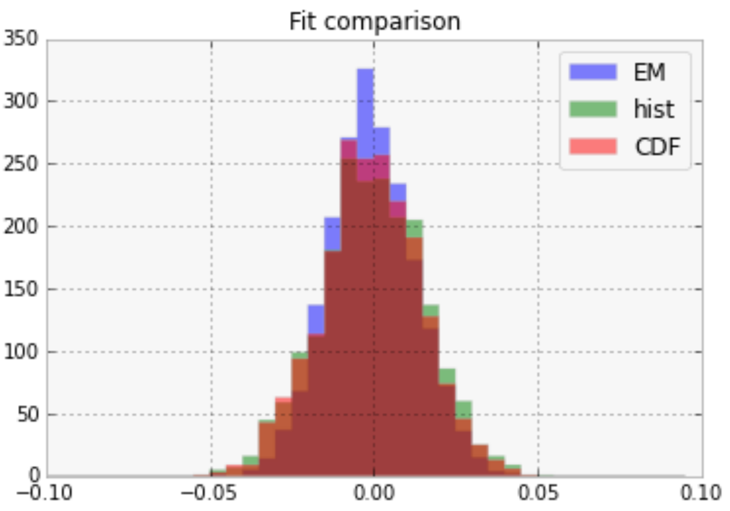
In [76]:
```
sample_parameters = dict(N=200, p=[0,0.15,0.8,0.15,0.7])
s = sim_two_gauss_mix(**sample_parameters)
hist(s, bins=r_[-0.4:1.4:0.05]);
```



In [77]:
```
P_em, P_h, P_cdf = test_accuracy(N_test=2000, p0=[0,0.1,0.6,0.1,0.5]
,
                                 **sample_parameters)
```

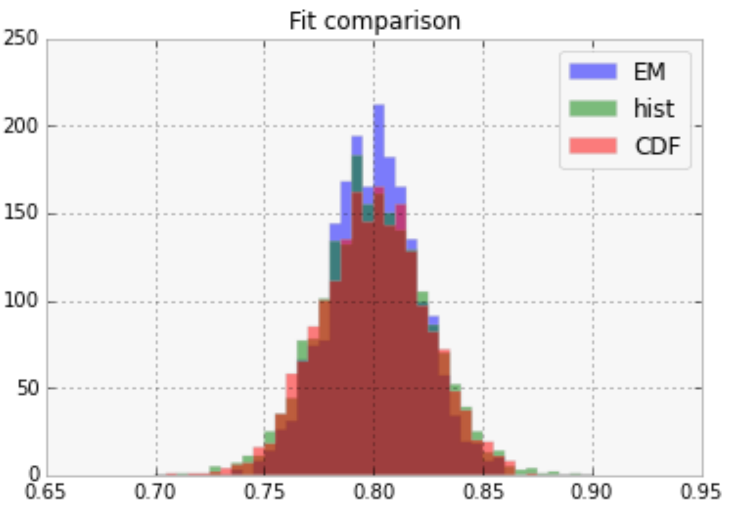
In [78]:
```
plot_accuracy([P_em, P_h, P_cdf], labels=['EM', 'hist', 'CDF'], ip=0
,
                 bins=(r_[-.1:.1:0.005]))
```

```
METHOD      MEAN  STD.DEV.
EM:        -0.06   1.31
hist:      -0.04   1.60
CDF:       -0.08   1.53
```



In [79]:
```
plot_accuracy([P_em, P_h, P_cdf], labels=['EM', 'hist', 'CDF'], ip=2
,
                 bins=(r_[.7:.9:0.005]))
```

```
METHOD      MEAN  STD.DEV.
EM:        80.04   2.04
hist:      80.03   2.58
CDF:       79.02   6.03
```



> **EM** is better both for bias and accuracy (CDF is worst)

In []:

In []:

In []:

In []:

In []:

In []:

In []:

In [82]:

```
from IPython.core.display import HTML
def css_styling():
    styles = open("./styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[82]:

In []: