Clustering with K-Means and EM

Guilherme Franca (guifranca@gmail.com (mailto:guifranca@gmail.com)), 07/28/2016

Abstract: We briefly review K-means and the Expectation Maximization (EM) algorithms. For EM we consider Gaussian Mixture Models (GMM) only. We implement both algorithms and show some simple simulations.

K-means

Let us briefly review the K-means algorithm. Let C_k, $k = 1, \dots, K$, denote a cluster of points with center $\mu_k \in \mathbb{R}^D$. Consider a data set $\{x_n\}_{n=1}^N$, where $x_n \in \mathbb{R}^D$. For each x_n introduce a binary vector z_n with components

$$z_{nk} = \begin{cases} 1 & \text{if } x_n \in C_k \\ 0 & \text{otherwise} \end{cases}$$

thus the variable z_n specifies to which cluster the point x_n belongs to. Notice that $\sum_{j=1}^K z_{nk} = 1$. Define the *distortion measure*

$$J = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \|x_n - \mu_k\|^2$$

which is the sum of intra-cluster square distances. The problem we need to solve is

$$\min_{\{z_{nk}\},\{\mu_k\}} J$$

We solve this problem through an iterative procedure consisting of two steps. First, we choose some initial values for each μ_k. Then we minimize J with respect to z_{nk} while keeping μ_k fixed (this is the analogous of the E-step in EM). Then we minimize J whith respect to μ_k while keeping z_{nk} fixed (this is the analogous of the M-step). We repeat this until convergence is attained. More specifically, the E-step consists of the update

$$z_{nk} = \begin{cases} 1 & \text{if } k = \text{argmin}_j \|x_n - \mu_j\|^2 \\ 0 & \text{otherwise} \end{cases}$$

For the M-step, differentiate J with respect to μ_k and equate to zero, yielding

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} z_{nk} x_n$$

where we defined $N_k = \sum_{n=1}^N z_{nk}$, the number of points belonging to cluster C_k.

In simple words, K-means just associate the point x_n to the cluster with closest center, then update the centers of each cluster by taking the average of the points associated to it. K-means is said to be hard-clustering since each point belongs to one and only cluster (in EM each point has a probability of belonging to each cluster, which is a soft-clustering).

This algorithm converges to a local minimum, however, it is highly sensitive to the initial conditions. A good initialization procedure is described in the following.

K-means++

This initialization makes the algorithm more stable. Denote $D(x_i, \mu_j) = \|x_i = \mu_j\|^2$.

1. Choose μ_1 at random from $\{x_n\}$.
2. For each x_n, assign the value $d_n = \min\{D(x_n, \mu_1), \ldots, D(x_n, \mu_k)\}$, where $k \leq K$ is the number of centers already chosen at this stage.
3. Form a probability vector $p \in \mathbb{R}^n$ out of $\{d_n\}$ such that $p_n = \frac{d_n}{\sum_{j=1}^{k} d_j}$. Choose μ_k at random from $\{x_n\}$ with probability distribution given by p.
4. Repeat until $k = K$.

Implementation and Simulation

The above algorithm is implemented in the file **kmeans.py**. Below we make use of this code.

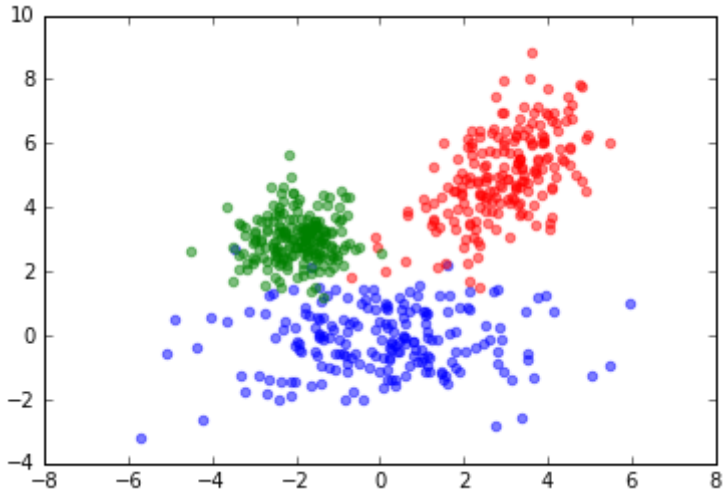
In [23]:
```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

import kmeans
```

First we generate artificial data. We generate 3 clusters from a 2-dimensional gaussian distribution, and we plot this original data set.

In [61]:
```
mean = np.array([0, 0])
cov = np.array([[4, 0], [0, 1]])
data1 = np.random.multivariate_normal(mean, cov, 200)

mean = np.array([3, 5])
cov = np.array([[1, 0.8], [0.8, 2]])
data2 = np.random.multivariate_normal(mean, cov, 200)

mean = np.array([-2, 3])
cov = np.array([[0.5, 0], [0, 0.5]])
data3 = np.random.multivariate_normal(mean, cov, 200)

data = np.concatenate((data1, data2, data3))
```

In [62]:
```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(data1[:, 0], data1[:, 1], color='blue', alpha=0.5)
ax.scatter(data2[:, 0], data2[:, 1], color='red', alpha=0.5)
ax.scatter(data3[:, 0], data3[:, 1], color='green', alpha=0.5)
```

Out[62]:
```
<matplotlib.collections.PathCollection at 0x7f0bfee95750>
```



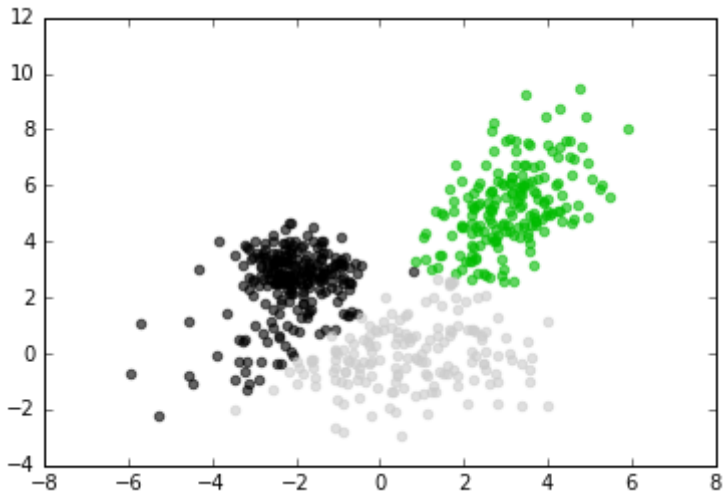
Now let us use K-Means to cluster this data set.

In [26]:
```
K=3
labels, centers = kmeans.kmeans(K, data)
```

In [27]:
```
print centers
```

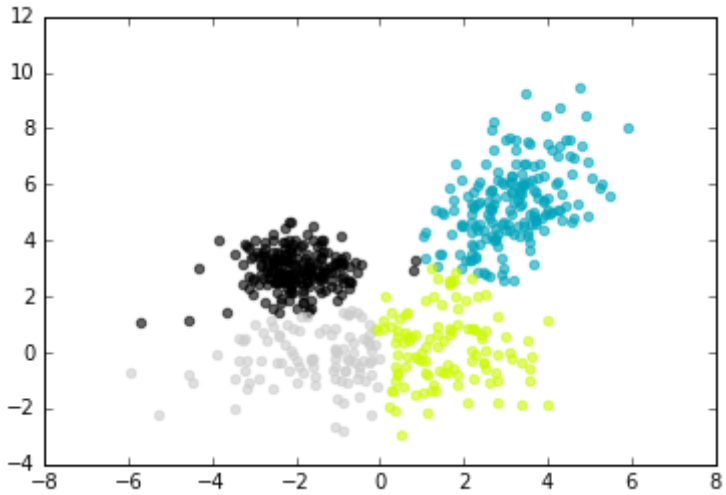
```
[[-2.1271375   2.50650622]
 [ 3.10084627  5.26202224]
 [ 0.70895433 -0.07376314]]
```

In [28]:
```
fig = plt.figure()
ax = fig.add_subplot(111)
colors = getattr(cm, 'spectral')(np.linspace(0, 1, K))
for k in range(K):
    xs = data[:,0][np.where(labels==k)]
    ys = data[:,1][np.where(labels==k)]
    ax.scatter(xs, ys, color=colors[k], alpha=.6)
```



In [29]:
```
K=4
labels, centers = kmeans.kmeans(K, data)
```

In [30]:
```
fig = plt.figure()
ax = fig.add_subplot(111)
colors = getattr(cm, 'spectral')(np.linspace(0, 1, K))
for k in range(K):
    xs = data[:,0][np.where(labels==k)]
    ys = data[:,1][np.where(labels==k)]
    ax.scatter(xs, ys, color=colors[k], alpha=.6)
```



Maximum Likelihood

Given a data set $\{x_n\}$ and a probability density function $f(x|\theta)$, where θ represent the parameters, the data likelihood function is given by

$$L(\theta) = \prod_{n=1}^{N} f(x_n|\theta)$$

The goal of maximum likelihood estimation (MLE) is to solve

$$\widehat{\theta} = \text{argmax}_\theta L(\theta)$$

given the data. It is more convenient to work with the log likelihood function

$$\ell(\theta) = \sum_{n=1}^{N} \log f(x_n|\theta)$$

and one tries to solve $\partial_\theta \ell(\theta) = 0$ to obtain a closed form solution, or if this is not possible one must solve the above optimization problem numerically.

MLE for a Single Gaussian

Consider a Gaussian distribution

$$\mathcal{N}(x|\mu, \Sigma) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} \exp\left\{ -\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu) \right\}$$

We thus have

$$\ell(\mu, \Sigma) = -\frac{ND}{2}\log 2\pi - \frac{N}{2}\log|\Sigma| - \frac{1}{2}\sum_{n=1}^{N}(x_n-\mu)^T\Sigma^{-1}(x_n-\mu)$$

Solving $\partial_\mu \ell = 0$ and $\partial_\Sigma \ell = 0$ we obtain the MLE estimators

$$\widehat{\mu} = \frac{1}{N}\sum_{n=1}^{N} x_n \qquad \widehat{\Sigma} = \frac{1}{N}\sum_{n=1}^{N}(x-\mu)(x-\mu)^T$$

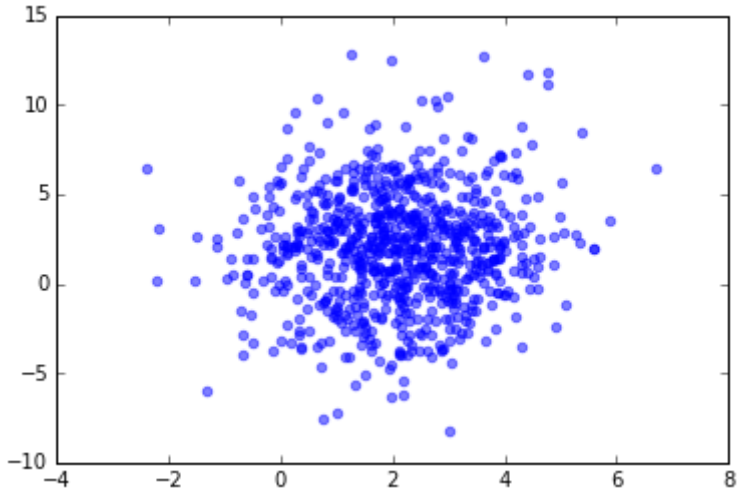
One can check that $\mathbb{E}[\widehat{\mu}] = \mu$ (unbiased) and $\mathbb{E}[\widehat{\Sigma}] = \frac{N-1}{N}\sigma$ (biased). Thus MLE underestimate the variance, but it is consistent. This is only a problem for small N. We redefine an unbiased estimator for the covariance matrix through

$$\widehat{\Sigma} = \frac{1}{N-1}\sum_{n=1}^{N}(x-\mu)(x-\mu)^T$$

Implementation: A simple MLE estimation for a single Gaussian is found in file **gaussian_mle.py**. Let us use this code in the following.

In [58]:
```
import gaussian_mle as gmle

# generate data from a 2D Gaussian
mu = np.array([2, 2])
sigma = np.array([[2, 0.5], [0.5, 10]])
gauss_data = np.random.multivariate_normal(mu, sigma, 800)

plt.scatter(gauss_data[:,0], gauss_data[:,1], color='blue',
alpha=0.5)
```

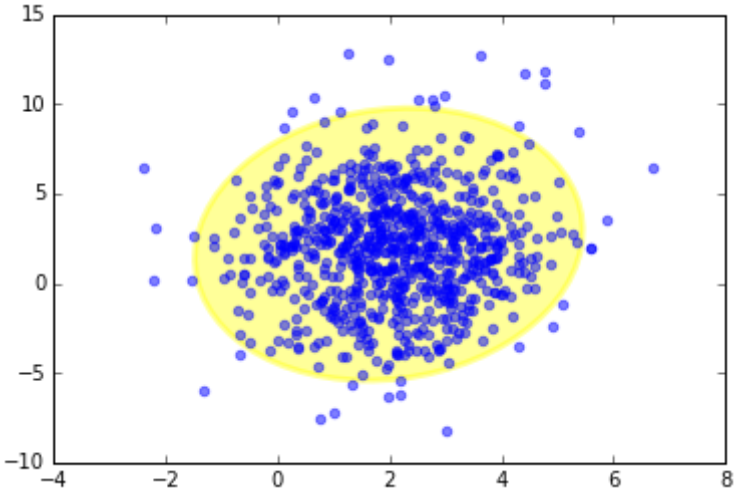
Out[58]: <matplotlib.collections.PathCollection at 0x7f0bfed01390>



In [60]:
```
muhat, sigmahat = gmle.gaussian_mle_estimator(gauss_data)
print muhat
print sigmahat
```

```
[ 2.07313258  1.98093719]
[[ 1.77838537  0.27426216]
 [ 0.27426216  9.94595305]]
```

In [59]:
```
# plot the confidence interval for the estimated Gaussian containing
 95% of points
gmle.scatter_ellipse(gauss_data, muhat, sigmahat)
```



To plot this ellipse we solve the eigenvalue problem $\Sigma v = \lambda v$, and in this 2D case we obtain two solutions (λ_i, v_i) for $i = 1, 2$, and we assume $\lambda_1 \geq \lambda_2$. Now the axis of maximum variation makes an angle $\alpha = \arctan \frac{(v_1)_y}{(v_1)_x}$. Now the equation of the ellipse is

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{a}\right)^2 = 1$$

with $a = 2\sqrt{c\lambda_1}$ and $b = \sqrt{c\lambda_2}$ and the major axis of the ellipse must make an angle α with the x-axis. The constant $c = 5.991$ give a 95% confidence interval.

MLE for Gaussian Mixtures

A mixture of K Gaussian components has the form

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

Integrating this over x implies that $\sum_k \pi_k = 1$. Moreover, since $p(x) \geq 0$ and $\mathcal{N}(\cdot|\cdot) \geq 0$ it implies $\pi_k \geq 0$ and thus $0 \leq \pi_k \leq 1$. Thus π_k is the (prior) probability that any given point belongs to component k

Now let $z \in \mathbb{R}^K$ be a binary vector such that $z_k \in \{0, 1\}$ and $\sum_k z_k = 1$. This vector tells to which component a point x was drawn from. There are K different ways of building such a z, which is a latent variable. Notice that $p(z_k = 1) = \pi_k$. We have $p(x, z) = p(x|z)p(z)$ where $p(z) = \prod_k \pi_k^{z_k}$ and $p(x|z) = \prod_k \mathcal{N}(x|\mu_k, \sigma_k)^{z_k}$. Therefore,

$$p(x) = \sum_z p(x, z) = \sum_z p(x|z)p(z) = \sum_z \prod_k \left(\mathcal{N}(x|\mu_k, \Sigma_k)\pi_k\right)^{z_k}$$

The sum over z implies that

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

So we obtain a GMM through a latent variable z. Sometimes x is called the observed data and z the unobserved data.

$p(z_k = 1) = \pi_k$ is the prior probability of any point x coming from component k. The posterior probability, $\gamma(z_k) = p(z_k = 1|x)$, is called the responsability. From Bayes' theorem we have

$$\gamma(z_k) = \frac{\pi_k \mathcal{N}(x|\mu_k, \Sigma_k)}{\sum_l \pi_l \mathcal{N}(x|\mu_l, \Sigma_l)}$$

The log likelihood function reads

$$\ell(\mu, \Sigma, \pi) = \sum_{n=1}^{N} \log\left(\sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)\right) + \lambda\left(\sum_k \pi_k - 1\right)$$

where we introduced a Lagrange multiplier. We introduce one latent variable z_n for each data point x_n . z_{nk} denotes the kth component of this vector. Now solving $\partial_{\mu_k}\ell = 0$, $\partial_{\Sigma_k}\ell = 0$, and $\partial_{\pi_k}\ell = 0$ we obtain

$$\widehat{\mu}_k = \frac{1}{N_k}\sum_{n=1}^{N} \gamma(z_{nk})x_n, \quad \widehat{\Sigma}_k = \frac{1}{N_k}\sum_{n=1}^{N} \gamma(z_{nk})(x_n - \mu_k)(x_n - \mu_k)^T, \quad \hat{\pi}_k = \frac{N_k}{N}$$

where $N_k = \sum_{n=1}^{N} \gamma(z_{nk})$ is the effective number of points in component k. It seems this is a closed form solution but it's not since γ has an involved dependency on the data. The EM algorithm solves this problem iteratively as follows.

The EM Algorithm

It is an iterative algorithm consisting of two steps, the E-step and the M-step.

1. Initialize μ_k and Σ_k for all $k = 1, \ldots, K$. Compute $\ell(\mu, \Sigma, \pi)$. (We may use K-Means to initialize these values.)
2. **E-step.** The expectation step consists in computing the responsabilities $\widehat{\gamma}(z_{nk})$ for each data point x_n

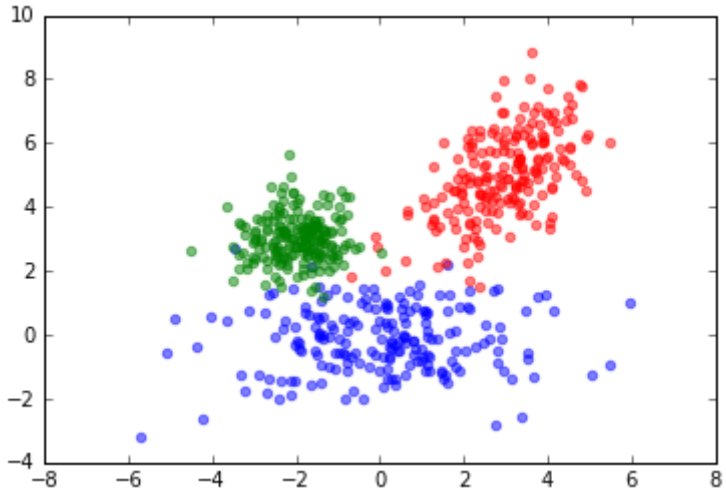
based on the current values of parameters (μ, Σ, π).

3. **M-Step.** The maximization step consists in maximizing ℓ with the current values of responsabilities, thus using the previously derived solution for $\widehat{\mu}$, $\widehat{\Sigma}$, and $\widehat{\pi}$.
4. We re-evaluate $\ell(\widehat{\mu}, \widehat{\Sigma}, \widehat{\pi})$ and check for convergence. Repeat from step 2 until convergence is attained.

This algorithm is implemented in file **gmm.py** and illustrated below.

In [63]:
```
# let's use the same data as for K-means
# just ploting to remind ourselves
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(data1[:, 0], data1[:, 1], color='blue', alpha=0.5)
ax.scatter(data2[:, 0], data2[:, 1], color='red', alpha=0.5)
ax.scatter(data3[:, 0], data3[:, 1], color='green', alpha=0.5)
```

Out[63]:
```
<matplotlib.collections.PathCollection at 0x7f0bfeab6e90>
```



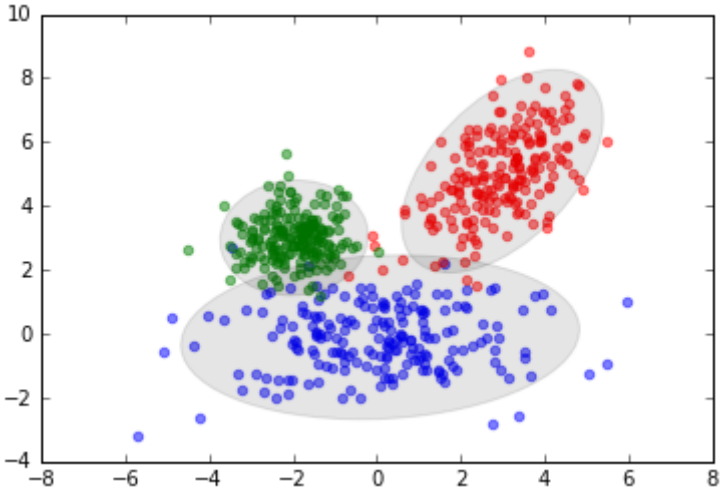
In [64]:
```
# let's choose some random initialization
mus = [ np.array([4, 4]),
        np.array([8, 8]),
        np.array([-4, -4])
]
sigmas = [ 2.*np.eye(2),
           3.*np.eye(2),
           1.5*np.eye(2)
]
pis = [.5, .25, .25]
```

In [65]:
```
# now let us apply EM algorithm to the above data set
import gmm

g = gmm.GMM(data, mus, sigmas, pis)
g.fit()
```

In [66]:
```
muhat, sigmahat, pihat = g.mu, g.sigma, g.pi
print muhat
print sigmahat
print pihat
```

```
[array([-1.96974825,  3.00045906]), array([ 2.99212999,  5.0872457
1]), array([ 0.08311805, -0.11027195])]
[array([[ 0.51994592,  0.0199688 ],
       [ 0.0199688 ,  0.54273839]]), array([[ 0.96275485,  0.6636772
4],
       [ 0.66367724,  1.6839943 ]]), array([[ 3.76324865,  0.1936229
1],
       [ 0.19362291,  1.10208097]])]
[0.34017553656784649, 0.32369292472312006, 0.33613153870903351]
```

In [67]:
```
from matplotlib.patches import Ellipse

fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(data1[:, 0], data1[:, 1], color='blue', alpha=0.5)
ax.scatter(data2[:, 0], data2[:, 1], color='red', alpha=0.5)
ax.scatter(data3[:, 0], data3[:, 1], color='green', alpha=0.5)

for mu, sigma in zip(muhat, sigmahat):
    vals, vecs = np.linalg.eigh(sigma)
    idx = vals.argsort()[::-1]
    vals = vals[idx]
    vecs = vecs[:,idx]
    alfa = np.degrees(np.arctan2(*vecs[:,0][::-1]))
    a, b = 2*np.sqrt(5.991*vals)
    ellipse = Ellipse(xy=mu, width=a, height=b, angle=alfa,
                      color='k', alpha=.1, zorder=1)
    ax.add_artist(ellipse)
```



Notice that EM does a much better job than K-means at clustering these points, however it's much slower.