Unsupervised learning of a mixture model

We are going to

- define a "Mixture of Gaussians" generative model, as a joint probability distribution: a prior times a likelihood, normalised.
- show how to "invert" the model (calculate the posterior)
- derive the learning algorithm for training its parameters (maximize the likelihood)

The result is a clustering algorithm.

"learning" a *single* Gaussian

We'll stick to 1 dimensional data for simplicity. x_n is the n^{th} item in a data set $\mathcal{D}$. The log likelihood is a sum of logs:

$$\log P(\mathcal{D}|\mu,\sigma^2) = \sum_{n=1}^{N} \log P(x_n|\mu,\sigma^2)$$

where $$P(\mathbf{x}_n|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x_n-\mu)^2}{2\sigma^2}\right]$$

Log of exponential simplifies:

$$\log P(\mathcal{D}|\mu,\sigma^2) \;=\; -\frac{N}{2}\log(2\pi\sigma^2) \;-\; \sum_{n=1}^{N}\frac{(x_n-\mu)^2}{2\sigma^2}$$

so the log likelihood is just a **sum of inverted parabolas.**

Ex: finding the max-likelihood mean, μ

The gradient w.r.t. μ of the log likelihood is easy, so long as we do it in steps & don't panic:

$$\begin{aligned}\frac{\partial}{\partial\mu}\left[\log P(\mathcal{D}\mid\mu,\sigma^2)\right] &= -\frac{\partial}{\partial\mu}\left[\sum_{n=1}^{N}\frac{(x_n-\mu)^2}{2\sigma^2}\right]\\ &= -\frac{1}{2\sigma^2}\sum_{n=1}^{N}\frac{\partial}{\partial\mu}(x_n-\mu)^2\\ &= \frac{1}{\sigma^2}\sum_{n=1}^{N}(x_n-\mu)\\ &= \frac{1}{\sigma^2}(N\bar{x}-N\mu)\end{aligned}$$

Now set this to zero and solve, which gives $\mu = \bar{x}$.

ie. "max likelihood" mean is the empirical mean. (Same thing goes for variance, but we won't bother proving that here).

a mixture model is a weighted sum

Think of a world in which class k **generates** pattern $\mathbf{x}$. We could model the joint:

$$P(k,\mathbf{x}) \;=\; P(k)\;P(\mathbf{x}|k)$$

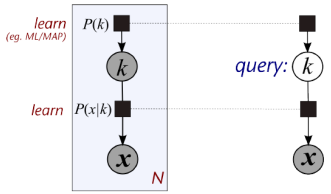
The generative model is that we choose a class k from a categorical distribution, and then generate a vector $\mathbf{x}$ from another distribution whose parameters depend on k. The density of patterns $\mathbf{x}$ must be

$$P(\mathbf{x}) \;=\; \sum_{k=1}^{K} P(k)\;P(\mathbf{x}|k)$$

This is called a *mixture* model for $\mathbf{x}$: the density of $\mathbf{x}$ is a mixture of several components.

- the prior probability $P(k)$ of a component is called its mixing coefficient, often denoted π_k.
- Each component of the mixture has its very own density, which is different for each class.

supervised learning of a mixture model



Joint:

$$P(k, \mathbf{x}) \;=\; P(k)\; P(\mathbf{x}|k)$$

If k is observed and $\mathbf{x}$ is Gaussian, we can trivially "fit" a Gaussian for each k.

Then for a novel $\mathbf{x}$ we can infer degree of belief in different class labels $P(k|\mathbf{x})$: just "invert the arrow" with Bayes theorem,

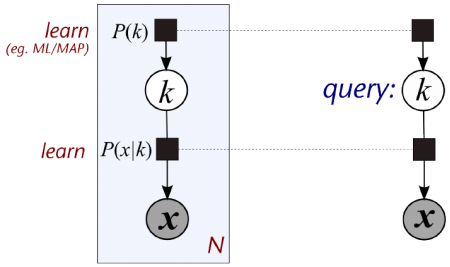
$$P(k|\mathbf{x}) \;\propto\; P(k)\; P(\mathbf{x}|k)$$

- π_k are mixing coefficients
- Each component of the mixture has its very own Gaussian density, parameterized by a vector of means $\mathbf{m}_k$ and a covariance matrix $\mathbf{C}_k$.

unsupervised learning

(note the shading, indicating nodes whose values are known).

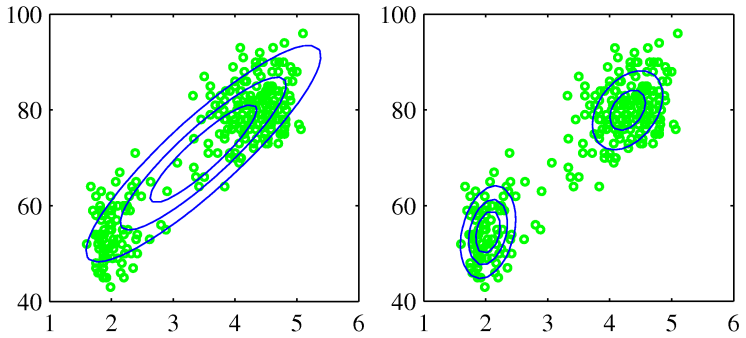
In **unsupervised learning** we only ever see $\mathbf{x}$ in the training set, never k!



Q: How to do it?

You can have a mixture of *anything*, but we will look at a particularly popular example in which the mixture components (ie. class distributions) are all Gaussians.

mixture model



Notice that there may be more data in one cluster than another, and our model ought to capture that aspect.

NOTICE: once we have "good" positions for the Gaussians, we can do "soft" clustering, by infering how much we believe each Gaussian was "responsible" for some novel input $\mathbf{x}$.

a mixture of Gaussians

So let's upgrade from a single Gaussian $P(\mathbf{x}) = \mathcal{N}(\mathbf{x} \mid \mathbf{m}, \mathbf{C})$ to a mixture:

$$P(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \quad \underbrace{P(\mathbf{x} \mid k)}_{\mathcal{N}(\mathbf{x}|\mathbf{m}_k, \mathbf{C}_k)}$$

Each component of the mixture has its very own Gaussian density, parameterized by a vector of means $\mathbf{m}_k$ and a covariance matrix $\mathbf{C}_k$.

the probability of generating $\mathcal{D}$

N data points, and K components.

As usual, the data is assumed to be generated i.i.d. so the log likelihood is a sum of logs:

$$\begin{aligned}\log P(\mathcal{D}) &= \sum_{n=1}^{N} \log P(\mathbf{x}) \\ &= \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \underbrace{P(\mathbf{x} \mid k)}_{\mathcal{N}(\mathbf{x}|\mathbf{m}_k, \mathbf{C}_k)}\end{aligned}$$

So how do we find the right π_k, $\mathbf{m}_k$ and $\mathbf{C}_k$?

Say we start with some randomly chosen values. As before, let's calculate the gradient of $\log P(\mathcal{D})$...

first, a trick

$$\frac{\partial}{\partial z} \log P(z) = \frac{1}{P(z)} \frac{\partial P(z)}{\partial z}$$

In other words, we can write

$$\frac{\partial P(z)}{\partial z} = P(z) \frac{\partial}{\partial z} \log P(z)$$

This trick gets used once in each form, in the next slide.

We will assume spherically symmetric Gaussians, and work through the calculation for the j^{th} mean, $\mathbf{m}_j$.

The gradient of the log likelihood

We'll look at one element in the training set, say the n^{th}:

$$\begin{aligned}\frac{\partial}{\partial \mathbf{m}_j} \log P(\mathbf{x}_n) &= \sum_n \frac{1}{P(\mathbf{x}_n)} \frac{\partial}{\partial \mathbf{m}_j} P(\mathbf{x}_n) && \longleftarrow \text{trick} \\ &= \sum_n \frac{1}{P(\mathbf{x}_n)} \frac{\partial}{\partial \mathbf{m}_j} \left[\sum_{k=1}^{K} \pi_k \; P(\mathbf{x}_n \mid k) \right] \\ &= \sum_n \frac{1}{P(\mathbf{x}_n)} \frac{\partial}{\partial \mathbf{m}_j} \left[\pi_j \; P(\mathbf{x}_n \mid j) \right] \\ &= \sum_n \frac{\pi_j}{P(\mathbf{x}_n)} \frac{\partial}{\partial \mathbf{m}_j} P(\mathbf{x}_n \mid j) \\ &= \sum_n \underbrace{\frac{\pi_j \; P(\mathbf{x}_n \mid j)}{P(\mathbf{x}_n)}}_{\text{posterior}} \frac{\partial}{\partial \mathbf{m}_j} \log P(\mathbf{x}_n \mid j) && \longleftarrow \text{trick} \\ &= \sum_n \underbrace{P(j \mid \mathbf{x}_n)}_{\text{"responsibility", } r_{nj}} \underbrace{\frac{\partial}{\partial \mathbf{m}_j} \log P(\mathbf{x}_n \mid j)}_{\text{Gaussian, so this is } \frac{\mathbf{x}_n - \mathbf{m}_j}{\sigma^2}}\end{aligned}$$

Plugging that in...

$$\frac{\partial}{\partial \mathbf{m}_j} \log P(\mathcal{D}) = \sum_n r_{nj} \frac{(\mathbf{x}_n - \mathbf{m}_j)}{\sigma^2}$$

which we are setting to zero and solving, to give:

$$\mathbf{m}_j = \frac{\sum_n r_{nj} \, \mathbf{x}_n}{\sum_n r_{nj}}$$

This is exactly the K-means update! except that the "responsibilities" r are now numbers between zero and one: they're "soft".

Exercise: in the same way, figure out what the new values for π should be.

The update for σ is more complicated, but is still a sum over r_{nj} just like the mean is. One can also figure this out for general covariance $\mathbf{C}$. See textbooks for all the details.

the EM algorithm (expectation maximization)

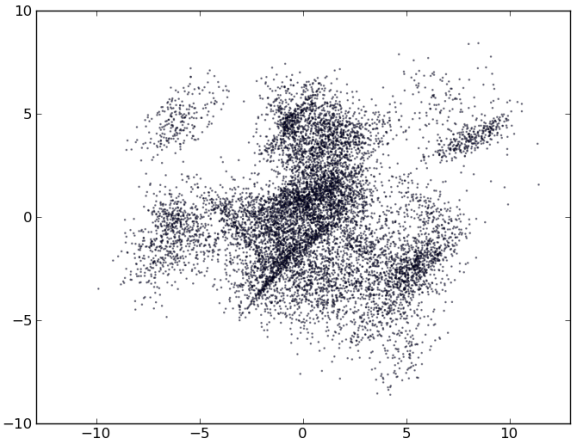
These updates move the parameters π, $\mathbf{m}$ and $\mathbf{C}$, which means the responsibilities are now "out of date", so we iterate. Every step increases the log likelihood!

Initialise $\mathbf{m}_k$ for all k, somewhere near the data,
Initialise $\mathbf{C}_k$ sensibly, for all k
Guestimate initial values for the π_k.

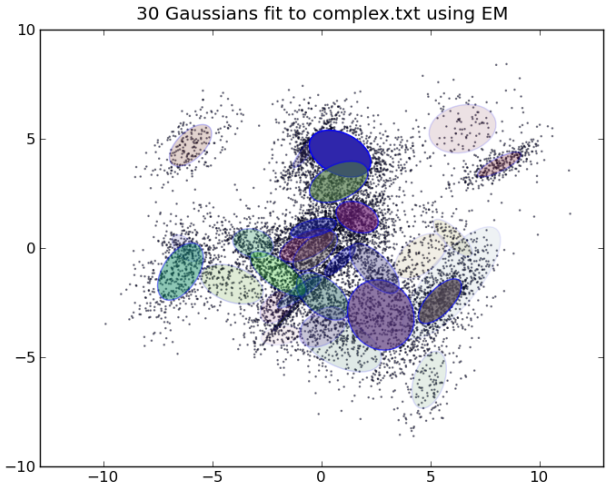
Until changes get tiny:

1. E step: For each data point $\mathbf{x}_n$, calculate the responsibilities, r
2. M step: Taking those responsibilities to be "soft" class assignments, reset all the parameters to max likelihood values.

example

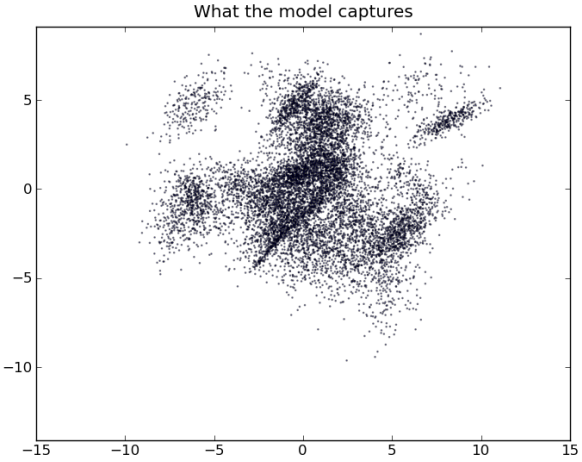


example



This is after 50 or so iterations of EM with `EMmix.py`.

example



This is what you get by *generating data* from the learned model.

I will put example code for this up a.s.a.p.