Gaussian mixture models

Real-world data is rarely Gaussian. Sometimes there are clear clusters that we might reasonably model as separate Gaussians. Sometimes there are no clear clusters, but we might be able to approximate the underlying density as a combination of overlapping Gaussians. (Given enough Gaussians, we can closely approximate any reasonable density.)

The Mixture of Gaussians (MoG) model is used to represent the probability distribution of real-valued D-dimensional feature vectors, in this note $\mathbf{x}$. It's possible that the MoG model is interesting on its own, as it could reveal clusters in a dataset. However, MoGs are usually part of a larger probabilistic model. As a simple example, MoGs can be used to model the data in each class for a Bayes classifier[1], replacing the single Gaussian used for each class in examples earlier in the course.

As a more complex example, MoGs can be used to model the distribution over patches of pixels in an image. A noisy image patch $\mathbf{y}$ can be restored by finding the most probable underlying image $\mathbf{x}$ by maximizing:

$$p(\mathbf{x} \mid \mathbf{y}) \propto p(\mathbf{y} \mid \mathbf{x})\, p(\mathbf{x}),$$

where $p(\mathbf{y} \mid \mathbf{x})$ is a noise model, and $p(\mathbf{x})$ is the prior over image patches, obtained by fitting a mixture of Gaussians.[2]

Because modelling a joint density is such a general task, Mixtures of Gaussians have *many* possible applications. They were used as part of the vision system of an early successful self-driving car[3]. They also have several possible applications in astronomy[4]...

The model and its likelihood

According to the MoG model, each datapoint has an integer indicator $z^{(n)} \in \{1, .., K\}$, stating which of K Gaussians generated the point. However, we don't observe $\{z^{(n)}\}$ — these are not labels, but hidden or *latent* variables. Under the model, the latents are drawn from a general discrete or categorical distribution with probability vector $\boldsymbol{\pi}$:

$$z^{(n)} \sim \text{Discrete}(\boldsymbol{\pi}).$$

Then the datapoints are drawn from the corresponding Gaussian "mixture component":

$$p(\mathbf{x}^{(n)} \mid z^{(n)}{=}k, \theta) = \mathcal{N}(\mathbf{x}^{(n)}; \boldsymbol{\mu}^{(k)}, \Sigma^{(k)}),$$

where $\theta = \{\boldsymbol{\pi}, \{\boldsymbol{\mu}^{(k)}, \Sigma^{(k)}\}\}$ are the parameters of the model.

To fit the model by maximum likelihood, we need to maximize

$$\log p(\mathcal{D} \mid \theta) = \sum_{n=1}^{N} \log p(\mathbf{x}^{(n)} \mid \theta),$$

where $p(\mathbf{x}^{(n)} \mid \theta)$ doesn't involve the latent $z^{(n)}$ because they aren't present in our observed data. To find the probability of the observations, we need to introduce the unknown terms

---

1. Reminder: the standard terminology is confusing. A "Bayes classifier" is one where we build a model of the features in each class, and then apply Bayes rule at test time to predict the label. How the classifier is constructed is a separate issue from whether it is "Bayes*ian*". In Bayesian classifiers we average predictions using the posterior distribution over possible parameters. In this note we won't be Bayesian, we'll just fit the parameters of the mixture models.
2. For more details, keen students could consult: From learning models of natural image patches to whole image restoration, Zoran and Weiss (2011).
3. Self-supervised monocular road detection in desert terrain, Dahlkamp et al. (2006)
4. Extreme deconvolution, Bovy et al. (2011); Replacing Standard Galaxy Profiles, Hogg and Lang (2013), ...

as dummy variables that get summed out, as we have done several times before:

$$\begin{aligned} p(\mathbf{x}^{(n)} \mid \theta) &= \sum_k p(\mathbf{x}^{(n)}, z^{(n)} = k \mid \theta), && \text{(sum rule)} \\ &= \sum_k p(\mathbf{x}^{(n)} \mid z^{(n)} = k, \theta)\, P(z^{(n)} = k \mid \theta), && \text{(product rule)} \\ &= \sum_k \pi_k \mathcal{N}(\mathbf{x}^{(n)}; \boldsymbol{\mu}^{(k)}, \Sigma^{(k)}). \end{aligned}$$

So the negative log-likelihood cost function that we would like to minimize is:

$$-\log p(\mathcal{D} \mid \theta) = -\sum_{n=1}^{N} \log \left[\sum_k \pi_k \mathcal{N}(\mathbf{x}^{(n)}; \boldsymbol{\mu}^{(k)}, \Sigma^{(k)}) \right].$$

Unlike the log of a product, the log of a sum does not immediately simplify. We can't find the maximum likelihood parameters in closed form: setting the derivatives of this cost function to zero doesn't give equations we can solve analytically.

Gradient-based fitting

We can fit the parameters $\theta = \{\boldsymbol{\pi}, \{\boldsymbol{\mu}^{(k)}, \Sigma^{(k)}\}\}$ with gradient methods. However, to use standard gradient-based optimizers we need to transform the parameters to be unconstrained. As previously discussed for stochastic variational inference, one way to represent the covariance matrix is in terms of an unconstrained lower-triangular matrix $\tilde{L}$, where:

$$\begin{aligned} L_{ij} &= \begin{cases} \tilde{L}_{ij} & i \neq j \\ \exp(\tilde{L}_{ii}) & i = j. \end{cases} \\ \Sigma &= LL^\top. \end{aligned}$$

The $\boldsymbol{\pi}$ vector is also constrained: it must be positive and add up to one. We can represent it as the softmax of a vector containing arbitrary values (as discussed earlier in the course).

Mixtures of Gaussians aren't usually fitted with gradient methods (see next section). However, by using gradient-based optimization, we can consider a mixture of Gaussians as a "module" in a neural network or deep learning system. Here's a sketch of the idea: The vectors modelled by the MoG could be the target outputs in a probabilistic regression problem with multiple outputs. We're simply replacing the usual Gaussian assumption for regression with a mixture of Gaussians. The parameters of the mixture model, θ, would be specified by a hidden layer that depends on some inputs. The gradients of the MoG log-likelihood would be backpropagated through the neural network as usual. This model is known as a *Mixture Density Network*[5].

For keen students: there is some literature analyzing gradient-based methods for mixture models[6]. There are also more sophisticated gradient-based optimizers that can deal with the constraints, which work better in some cases[7].

The EM algorithm

The *Expectation Maximization* (EM) algorithm is really a framework for optimizing a wide class of models.[8] Applied to Mixtures of Gaussians, we obtain a simple and interpretable iterative fitting procedure, which is more popular than gradient-based optimization.

---

5. Bishop (1994), Williams (1996)
6. Optimization with EM and expectation-conjugate-gradient, Salakhutdinov et al., ICML 2003.
7. Matrix manifold optimization for Gaussian mixtures, Hosseini and Sra, NIPS 2015.
8. For example, you may have heard of Hidden Markov Models (HMMs) in another course. These can be fitted with the EM algorithm, which for HMMs is known as the Baum–Welch algorithm. The E-step for HMMs is called the forward-backward algorithm (and people might use "forward-backward" to refer to the whole EM algorithm).

If we knew the latent indicator variables $\{z^{(n)}\}$, then fitting the parameters by maximum likelihood would be easy. We'd just find the empirical mean and covariance of the points belonging to each component. In addition the *mixing proportion* of a component/cluster π_k would be set to the fraction of points that belong to component k. We'd be fitting the parameters of Gaussian Bayes classifier, with labels $\{z^{(n)}\}$.

To set up some notation, we can indicate which cluster is *responsible* for each datapoint with a vector of binary variables giving a "one-hot" encoding: $r_k^{(n)} = \delta_{z^{(n)},k}$. Then we can write down expressions for what the maximum likelihood parameters would be, if we knew the assignments of the datapoints to components:

$$\pi_k = \frac{r_k}{N}, \qquad \text{where } r_k = \sum_{n=1}^{N} r_k^{(n)}$$

$$\boldsymbol{\mu}^{(k)} = \frac{1}{r_k} \sum_{n=1}^{N} r_k^{(n)} \mathbf{x}^{(n)}$$

$$\Sigma^{(k)} = \frac{1}{r_k} \sum_{n=1}^{N} r_k^{(n)} \mathbf{x}^{(n)} \mathbf{x}^{(n)\top} - \boldsymbol{\mu}^{(k)} \boldsymbol{\mu}^{(k)\top}.$$

The EM algorithm uses the equations above, with probabilistic settings of the unknown component assignments. The algorithm alternates between two steps:

1) **E-step:** Set soft *responsibilities* using Bayes' rule:

$$r_k^{(n)} = P(z^{(n)} = k \,|\, \mathbf{x}^{(n)}, \theta) = \frac{\pi_k \mathcal{N}(\mathbf{x}^{(n)}; \boldsymbol{\mu}^{(k)}, \Sigma^{(k)})}{\sum_l \pi_l \mathcal{N}(\mathbf{x}^{(n)}; \boldsymbol{\mu}^{(l)}, \Sigma^{(l)})}$$

2) **M-step:** Update the parameters $\theta = \{\boldsymbol{\pi}, \{\boldsymbol{\mu}^{(k)}, \Sigma^{(k)}\}\}$ using the responsibilities from the E-step in the equations for the parameters above.

If these steps are repeated until convergence, it can be shown that the algorithm will converge to a local maximum of the likelihood. In practice we terminate after some maximum number of iterations, or when the parameters are changing by less than some tolerance. Early stopping based on a validation set could also be used.

Some parameter settings have infinite likelihood. For example, place the mean of one component on top of a datapoint, and set the corresponding covariance to zero. Infinite likelihoods can also be obtained by explaining D or fewer of the D-dimensional datapoints with one of the components and making its covariance matrix low-rank. There are a variety of solutions to this issue, including reinitializing ill-behaved components, and regularizing the covariances.

Whether we use EM or gradient-based methods, the likelihood is multi-modal, and different initializations of the parameters lead to different answers.

Theoretical basis

EM is an iterative "bound-based" optimizer for the likelihood. It can be shown that the E-step constructs a lower bound of the log-likelihood function (a function of the parameters θ), which is tight (correct) at the current parameters. The M-step moves the parameters to the ones that maximizes the lower bound constructed in the E-step. Because the bound was tight before the M-step, increasing the value of the lower bound must also increase the likelihood (Murphy Fig. 11.16, or Bishop Fig. 9.14).

The bound is based on a variational approximation to the posterior over the unknown component labels. Using the KL-divergence to compare the posterior over components to an

arbitrary distribution, we know:

$$D_{\mathrm{KL}}(Q(\mathbf{z})||P(\mathbf{z}\,|\,X,\theta)) = \sum_{\mathbf{z}} Q(\mathbf{z}) \log \frac{Q(\mathbf{z})}{P(\mathbf{z}\,|\,X,\theta)} \geq 0$$

subtracting the log-likelihood from both sides gives

$$\sum_{\mathbf{z}} Q(\mathbf{z}) \log \frac{Q(\mathbf{z})}{P(\mathbf{z}\,|\,X,\theta)\,p(X\,|\,\theta)} \geq -\log p(X\,|\,\theta)$$

or

$$\begin{aligned}\log p(X\,|\,\theta) \geq \mathcal{L}(Q,\theta) &= \sum_{\mathbf{z}} Q(\mathbf{z}) \log P(X,\mathbf{z}\,|\,\theta) - \sum_{\mathbf{z}} Q(\mathbf{z}) \log Q(\mathbf{z}) \\ &= \sum_n \sum_k Q(z^{(n)}=k) \log \left[\pi_k \mathcal{N}(\mathbf{x}^{(n)}; \boldsymbol{\mu}^{(k)}, \Sigma^{(k)})\right] - \sum_{\mathbf{z}} Q(\mathbf{z}) \log Q(\mathbf{z}).\end{aligned}$$

In the E-step at iteration t we can identify the responsibilities $\{r_k^{(n)}\}$ as the variational probabilities $Q(z^{(n)}=k) = P(z^{(n)}=k\,|\,\mathbf{x}^{(n)},\theta^{(t)})$. The corresponding bound based on the KL-divergence is tight for $\theta=\theta^{(t)}$.

We can then improve the likelihood by setting:

$$\theta^{(t+1)} = \arg\max_{\theta} \mathcal{L}(Q,\theta),$$

where Q was set using $\theta^{(t)}$. Only the first term in the equation for $\mathcal{L}(Q,\theta)$ depends on the parameters θ. The term inside the log is now simple (there is no sum), and with some calculus one can show that the M-step described in this note does maximize this term.

For keen students: it's possible to put a variational distribution on θ as well as $\mathbf{z}$ and derive a variational approximation to the posterior over parameters. But Bayesian mixtures of Gaussians are out of scope for this course.

For you to think about

- How could you pick K the number of components?
- If at test time an element of a vector $\mathbf{x}$ is missing, how can we compute the marginal probability of the remaining elements under the MoG? A Bayes classifier based on MoGs can easily deal with missing values. What other advantages might the Bayes classifier have over a neural network, and what advantages does the neural network have?
- What would happen if we performed an EM update using only a small number of datapoints? (For keen students: there are mini-batch versions of EM – Murphy describes them.)
- If we do multiple runs from different initializations, do we necessarily want to pick the parameters with the highest likelihood?

Further reading

Murphy: Mixture of Gaussians 11.2.1, EM algorithm 11.4

Barber: Chapter 20.1 to 20.3, skipping material on discrete models.