Mixture models

Kevin P. Murphy

Last updated November 24, 2006

* Denotes advanced sections that may be omitted on a first reading.

1 Introduction

Most of the density functions we have considered so far are **unimodal**, that is, they have at most one peak. However, often we want to be able to represent densities with multiple modes. A common way to do this is to create a **mixture model**, which is a convex combination of pdf's:

$$p(x) = \sum_{k=1}^{K} \pi_k p(x|k) \tag{1}$$

where K is the (fixed) number of mixture component, π is a vector of **mixing weights**, and $p(x|k)$ are the densities for each component. We consider some examples below.

2 Gaussian mixture models

Consider the dataset of height and weight in Figure 1. It is clear that there are two subpopulations in this data set, and in this case they are easy to interpret: one represents males and the other females. Within each class or **cluster**, the data is fairly well represented by a 2D Gaussian (as can be seen from the fitted ellipses), but to model the data as a whole, we need to use a **mixture of Gaussians (MoG)** or a **Gaussian mixture model (GMM)**. This is defined as follows:

$$p(x|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) \tag{2}$$

where $\theta = \{\pi_k, \mu_k, \Sigma_k\}$ are the parameters. The π_k are called the **mixing weights**, and satisfy $0 \leq \pi_k \leq 1$ and $\sum_{k=1}^{K} \pi_k = 1$. μ_k and Σ_k are the mean and covariance of mixture component k. K is the number of components.

Note that the clusters do not always have an obvious meaning. For example, in Figure 2 we see a dataset which seems to have two clusters. Discovering the number and shape of clusters in data is an example of **unsupervised learning**. The program **Autoclass** uses a Bayesian approach to fit a GMM (which we will explain later), and has been used to discover new kinds of stars.

A helpful way to think about GMMs is to imagine that each data point x_n has an associated **latent indicator variable** $z_n \in \{1, \ldots, K\}$ that specifies which mixture component that data point came from. These are like the class labels in a Bayesian classifier, except we assume the labels are hidden (since this is unsupervised learning). We can write the **complete data log likelihood** as

$$\begin{aligned} \ell_c(\theta) &= \log p(x_{1:N}, z_{1:N}|\theta) \qquad (3)\\ &= \log \prod_n p(z_n|\pi) p(x_n|z_n, \theta) \qquad (4) \end{aligned}$$

where $p(z_n = k|\pi) = \pi_k$ is a multinomial, and $p(x_n|z_n = k, \theta) = \mathcal{N}(x_n|mu_k, \Sigma_k)$ is a Gaussian. This is illustrated as a DGM in Figure 3.

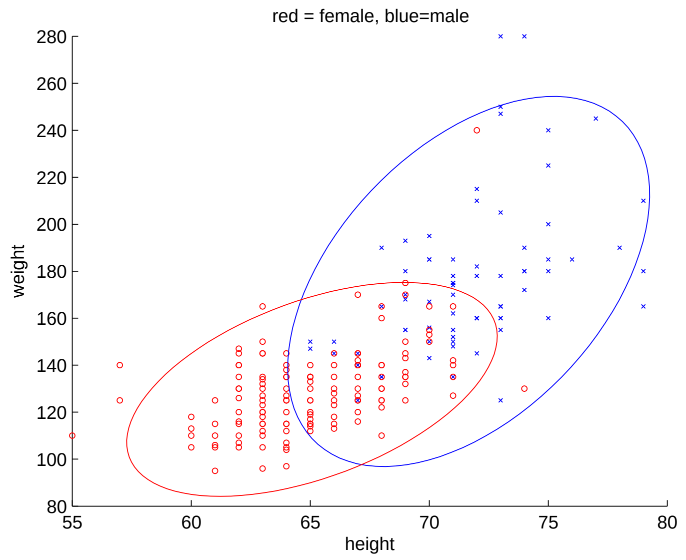

Figure 1: A scatterplot of height and weight of various men (blue crosses) and women (red circles). We superimpose two 2D Gaussians, with the 2 Σ ellipsoids representing the 95% confidence interval. Produced by `biometric_plot.m`.

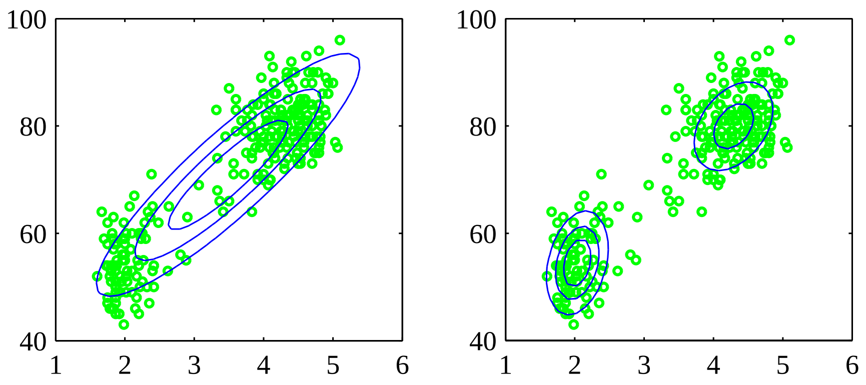

Figure 2: The Old Faithful data. Horizontal axis is duration of the erruption in minutes. Vertical axis is time until the next erruption in minutes. (a) A single Gaussian. (b) A mixture of two Gaussians. Source: [Bis06] Figure 2.21.

The corresponding **incomplete data log likelihood** is

$$\ell(\theta) = \log p(x_{1:N}|\theta) \tag{5}$$

$$= \sum_n \log p(x_n|\theta) \tag{6}$$

$$= \sum_n \log \sum_{z_n} p(x_n, z_n|\theta) \tag{7}$$

$$= \sum_n \log \sum_{z_n=1}^{K} \pi(z_n)\mathcal{N}(x_n|z_n, \mu(z_n), \Sigma(z_n)) \tag{8}$$

Note that this likelihood function has multiple modes. (In 1D, it has K modes, but if $d > 1$, it can have more than K modes [CPW03].) Hence finding the global maximum will be difficult. One can use gradient based methods, or the

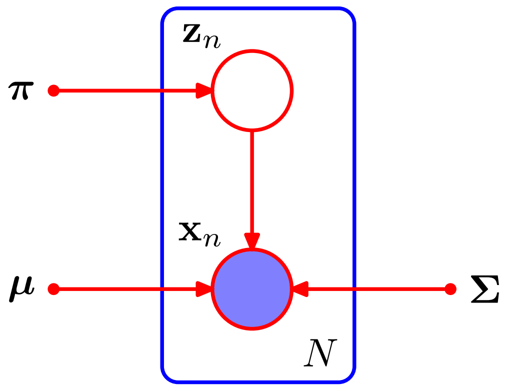

Figure 3: A GMM represented as a GMM. Source: [Bis06] Figure 9.6.

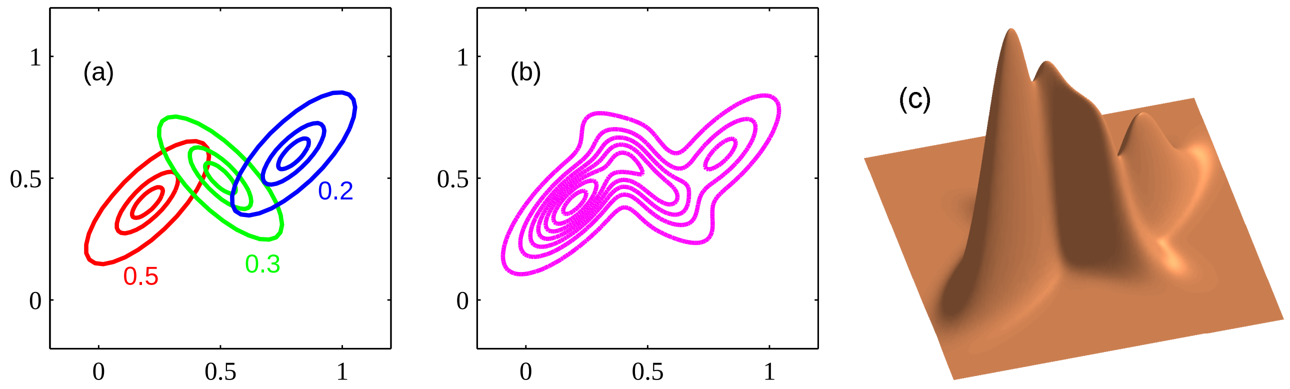

Figure 4: A mixture of 3 Gaussians in 2D. (a) We show the contours of constant probability and the mixing weights. (b) A contour plot of the 2D density. (c) A 3D surface plot of the 2D density. Source: [Bis06] Figure 2.23.

EM algorithm, to find the MLE or an MAP estimate. However, both will get stuck in local minima. We discuss this in more detail below.

2.1 Kernel density estimation

Given a sufficiently large number of mixture components, a GMM can be used to approximate any density. See Figure 4 for an example. If we associate a single Gaussian with every datapoint, we get what is called a **kernel density estimate (kde)** or **Parzen window estimate**. This is a **non parametric** density estimator. This does not mean it does not have parameters, rather it means that the number of parameters grows (linearly) with the number of data points. Specifically, a kde model has the form

$$p(x|h) = \frac{1}{N}\sum_{n=1}^{N} \frac{1}{h^d} k(\frac{x - x_n}{h}) \tag{9}$$

where $k(u)$ is called a **kernel function** and the parameter h is called the **bandwidth**. In the case that

$$k(u) = e^{-||u||^2/2} \tag{10}$$

we recover a GMM where each data point becomes a cluster with mean $\mu_n = x_n$, covariance $\Sigma_n = h^2 I_d$ (so $|\Sigma_n|^{-\frac{1}{2}} = 1/h^d$) and mixture weight $\pi_n = 1/N$. In general we can use any kernel function provided $k(u) \geq 0$ and $\int k(u)du = 1$.

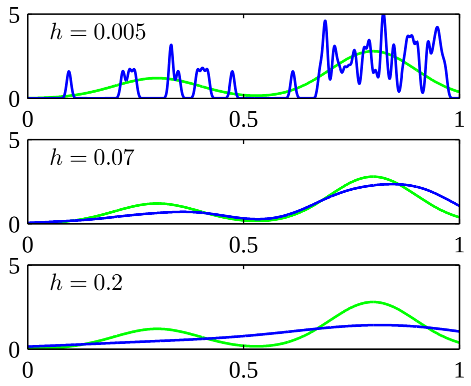

Figure 5: A nonparametric (Parzen) density estimator in 1D with a Gaussian kernel. The green curve is the true density, and the blue curve is the approximation using different smoothing parameters h. Source: [Bis06] Figure 2.25.

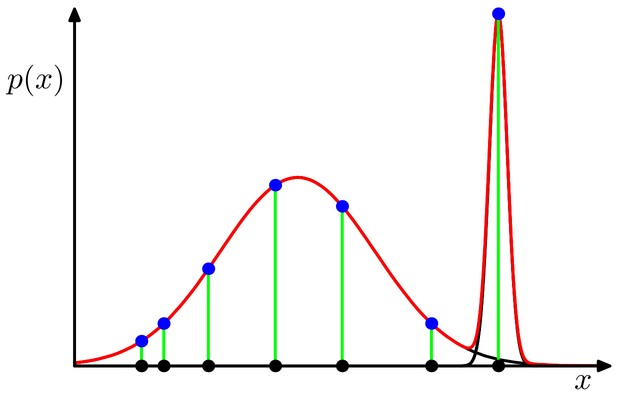

Figure 6: Illustration of how singularities can arise in the likleihood function of GMMs. Source: [Bis06] Figure 9.7.

h is called a **smoothing parameter**, and controls how smooth the resulting density is. See Figure 5 for an example. It is common to pick h by cross validation.

Although nonparametric models are very flexible, and sometimes easy to fit (e.g., for kde, just memorize all the data!), they take a a lot of memory and can be slow to use at test time.

2.2 Reducing the number of parameters in a GMM

The number of parameters in a GMM is $O(Kd^2)$, since each covariance matrix has $O(d^2)$ parameters. Although this is constant with respect to N (since this is a parametric model), it may still be a lot of parameters to estimate if d is large. In particular, estimating Σ_k can be a problem, since the empirical covariance matrix may not be positive definite. One common restriction is to assume a diagonal matrix, $\Sigma_k = \text{diag}(\sigma_{k1}^2, \ldots, \sigma_{kd}^2)$. Another even stronger restriction is to assume a spherical matrix, $\Sigma_k = \sigma_k^2 I_d$. In general, we can use **Gaussian graphical models** to represent each component density using a number of parameters that is anywhere between 1 and $O(d^2)$, depending on the conditional independence assumptions encoded in the graph. We can also impose priors on Σ_k and use MAP estimates, or the posterior mean, instead of the MLE.

Orthogonal to the issue of how to represent Σ_k within each cluster is how to represent the dependencies between the parameters across clusters. A common assumption is that all the lusters have the same "shape", i.e. Σ_k is **tied** across classes.

2.3 Singularities

There is a fundamental flaw in trying to maximize the likelihood function of a GMM. To understand the problem, suppose for simplicity that $\Sigma_k = \sigma_k^2 I$, and that $K = 2$. It is possible to get an infinite likelihood by assigning one of the centers μ_k, say μ_j, to a single data point, say x_n, and letting $\sigma_j \rightarrow 0$, and using the second Gaussian to fit the remaining data points, since the nth term makes the following contribution to the likelihood:

$$\mathcal{N}(x_n | x_n, \sigma_k^2 I) = \frac{1}{(2\pi)^{\frac{1}{2}}} \frac{1}{\sigma_j} \tag{11}$$

Hence we can drive this term to infinity by letting $\sigma_j \rightarrow 0$. See Figure 6.

There are various heuristics which are used to avoid this situation, such as jumping to a random point in parameter space if $\det \Sigma_k$ gets too small. A more principled solution is to find a MAP estimate by adding a prior to each Σ_k. A fully Bayesian approach also avoids the problem.

3 Mixture of multivariate Bernoullis

A mixture of Gaussians is appropriate if the data is real-valued, $x_n \in \mathbb{R}^d$. But what about binary data? For example, consider clustering the binary images in Figure 7. In this case, we can replace the Gaussian densities with a product of Bernoullis:

$$p(x|z = k, \theta) = \prod_{i=1}^{K} Be(x_i|\theta_{ki}) \prod_{i=1}^{K} x_i^{\theta_{ki}} (1 - x_i)^{1-\theta_{ki}} \tag{12}$$

This can be represented as a DGM as shown in Figure 8. Note that this is the same structure as a naive Bayes classifier, except that the class labels z_n are hidden.

One can show (exericse) that the mean and covariance of this mixture distribution are given by

$$E[x] = \sum_k \pi_k \mu_k \tag{13}$$

$$\text{Cov}[x] = \sum_k \pi_k [\Sigma_k + \mu_k \mu_k^T] - E[x]E[x]^T \tag{14}$$

where $\Sigma_k = \text{diag}(\theta_{ki}(1 - \theta_{ki}))$. Because the covariance matrix is no longer diagonal, the mixture distribution can capture correlations between variables, unlike a single product of Bernoullis.

Note that this model, unlike the GMM, does not suffer from singularities, because the likelihood function is bounded above, since $0 \leq p(x_n|\theta_k) \leq 1$.

After fitting this model (with EM) to the binary image data, the resulting parameter vectors θ_k can be visualized as gray scale images: see Figure 7(bottom left). If we fit the model using a single mixture component (i.e., a factored distribution), we get a result which has blurred all the components together.

4 Fitting GMMs with EM

For GMMs, we saw that the log likelihood is

$$\ell(\theta) = \sum_n \log \sum_{z_n=1}^{K} \pi(z_n) \mathcal{N}(x_n | z_n, \mu(z_n), \Sigma(z_n)) \tag{15}$$

This is a difficult function to optimize, because the log is outside the sum.

A further problem is that the model is not **identifiable**, which means there are many latent settings which have the same likelihood. Specifically, in a mixture model with K components, then for each setting of the parameter θ, there are $K! - 1$ other parameter settings (say θ') that have the same likelihood, $p(X|\theta) = p(X|\theta')$. This is called the **label switching problem**. For example, in Figure 1, we might label the male class as $Z = 1$ and the female as $Z = 2$, or vice versa. Of course, not all of these settings are maxima. In 1D, for Gaussians, there are K modes in the likelihood, although in higher dimensions, there may be more [CPW03].

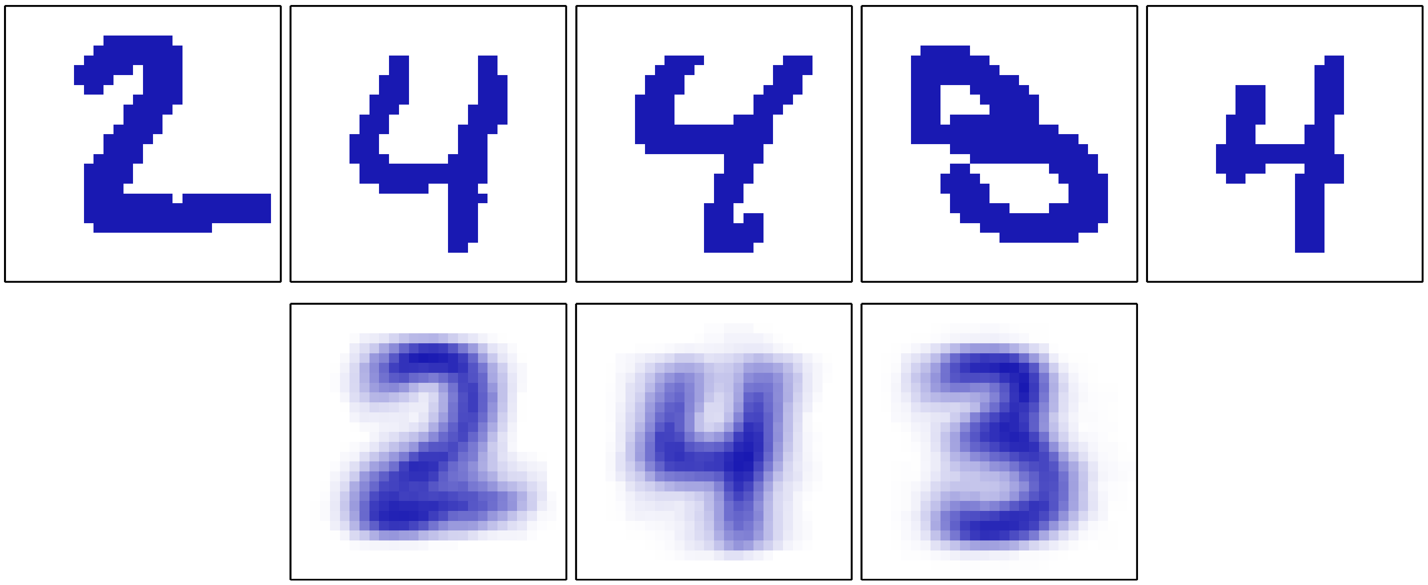
Figure 7: Illustration of the Bernoulli mixture model. Top: 3 kinds of binary digits. Each image is a 28×28 binary image, derived by thresholding the gray scale MNIST digits at 0.5. Bottom: the 3 cluster centers that are learned. Source: [Bis06] Figure 9.10.

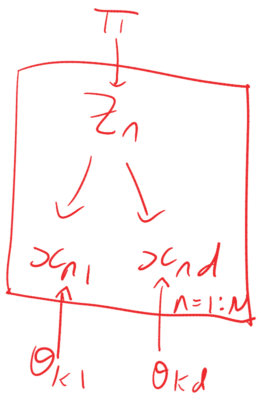

Figure 8: Mixture of Bernoullis as a DGM.

One can find a local optimum of $\ell(\theta)$ using a gradient based technique, such as conjugate gradient ascent. However, this requires setting a step size parameter, and enforcing the constraints that $\sum_k \pi_k = 1$ and that Σ_k are positive definite. There is an alternative algorithm called **Expectation Maximization (EM)** that is often preferred for finding MLE or MAP estimates of mixture models, because of its simplicity.

The key insight behind EM is this: if we knew the values of the latent variables z_n, then optimizing the (complete data) likelihood wrt θ would be easy: we would simply esimate μ_k and Σ_k applying the standard closed-form formula to all the data assigned to cluster k. Since we don't know the z_n, let's estimate them, and use their **filled in** values as substitutes for the real values. More precisely, we will optimize the *expected* complete data log likelihood instead of the actual complete data log likelihood. Since the estimates of z_n depend on the parameters θ, we need to re-estimate them after each update to θ. This algorithm can be shown to monotonically increase a lower bound on the log likelihood, and hence it will converge.

In more detail, the EM algorithm is as follows.

1. Initialize θ.

2. Repeat until $\ell(\theta)$ stops changing

   (a) E step: compute $p(z_n|x_n, \theta^{old})$ for each case n.

   (b) M step: compute

   $$\theta^{new} = \arg\max_\theta Q(\theta, \theta^{old}) \tag{16}$$

   where **auxiliary function** Q is the expected complete data log likelihood:

   $$\begin{aligned} Q(\theta, \theta^{old}) &= \sum_z p(z|x, \theta^{old}) \log p(x, z|\theta) & (17) \\ &= \sum_{z_{1:N}} p(z_{1:N}|x_{1:N}, \theta^{old})[\sum_n \log p(x_n, z_n|\theta)] & (18) \\ &= \sum_n \sum_{z_n} p(z_n|x_n, \theta^{old}) \log p(x_n, z_n|\theta) & (19) \end{aligned}$$

   (c) Compute the log likelihood

   $$\ell(\theta) = \log \sum_n \sum_{z_n} p(z_n, x_n|\theta) \tag{20}$$

Because EM will only find a local minimum, good initialization is important. But how to do this is problem dependent. A general strategy is to try **multiple restarts** at random locations.

We now explain how to implement these steps for GMMs. We usually initialize by assigning each μ_k to one of the data points chosen at random, and setting Σ_k to be a small fraction of the global covariance.

The E step is to compute

$$p(z_n = k|x_n, \theta^{old}) = r_{nk} = \frac{\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\sum_j \pi_j \mathcal{N}(x_n|\mu_j, \Sigma_j)} \tag{21}$$

The value r_{nk} is called the **responsibility** of cluster k for data point n.

The M step involves maximizing the expected complete data log likelihood:

$$\begin{aligned} Q(\theta, \theta^{old}) &= E \sum_n \log p(x_n, z_n|\theta) & (22) \\ &= E \sum_n \log \prod_k \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)^{I(z_n=k)} & (23) \\ &= E \sum_n \sum_k I(z_n = k) \log[\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)] & (24) \\ &= \sum_n p(z_n|x_n, \theta^{old}) \sum_k I(z_n = k) \log[\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)] & (25) \\ &= \sum_n \sum_k r_{nk} \log[\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)] & (26) \\ &= \sum_n \sum_k r_{nk} \log \pi_k + \sum_n \sum_k r_{nk} \log \mathcal{N}(x_n|\mu_k, \Sigma_k)] & (27) \\ &= J(\pi) + J(\mu_k, \Sigma_k) & (28) \end{aligned}$$

This can be optimized wrt π and μ_k, Σ_k separately. For the π term, we need to add a Lagrange multiplier to ensure $\sum_k \pi_k = 1$. Hence we solve

$$\frac{\partial}{\partial \pi_i}[\sum_n \sum_k r_{nk} \log \pi_k + \lambda(1 - \sum_k \pi_k)] = 0 \tag{29}$$

to find

$$\pi_k = \frac{1}{N} \sum_n r_{nk} \tag{30}$$

This makes sense: it is just the (weighted) number of points assigned to cluster k.

For the μ_k, Σ_k term, let us expand out the Gaussian:

$$\mathcal{N}(\vec{x}|\vec{\mu}, \Sigma) \overset{\text{def}}{=} \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp[-\tfrac{1}{2}(\vec{x}-\vec{\mu})^T\Sigma^{-1}(\vec{x}-\vec{\mu})] \tag{31}$$

Hence dropping constant terms we have

$$J(\mu_k, \Sigma_k) = -\tfrac{1}{2}\sum_n \sum_k r_{nk} \log|\Sigma_k| + (x_n-\mu_k)^T\Sigma^{-1}(x_n-\mu_k) \tag{32}$$

This is just a weighted version of the standard problem of estimating a MVN. Taking derivatives in the usual way results in the following

$$\mu_k^{new} = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}} \tag{33}$$

$$\Sigma_k = \frac{\sum_n r_{nk}(x_n - \mu_k^{new})(x_n - \mu_k^{new})^T}{\sum_n r_{nk}} \tag{34}$$

An example of the algorithm in action is shown in Figure 9. We start with $\mu_1 = (-1, 1)$, $\Sigma_1 = I_1$, $\mu_2 = (1, -1)$, $\Sigma_2 = I_1$. We color code points such that blue points come from cluster 1 and red points from cluster 2. More precisely, we set the color to

$$\text{color}(n) = r_{n1}\text{blue} + r_{n2}\text{red} \tag{35}$$

so ambiguous points appear purple. After 20 iterations, the algorithm has converged on a good clustering. (The data was **standardized**, by removing the mean and dividing by the standard deviation, before processing. This often helps convergence.)

5 The K means algorithm

The K-means algorithm is a variant of the EM algorithm for GMMs. It assumes that all the clusters have the same fixed spherical covariance matrix, $\Sigma_k = I$, which is not updated. Also, in the E step, K-means uses a **hard assignment**, which means it assigns each data point z_n to the nearest cluster centre μ_k, rather than computing a weighted combination of points. This can be seen as a delta function approximation to the posterior (in the context of HMMs, this is called **Viterbi training**):

$$p(z_n|x_n, \theta) \approx \delta(z_n - z_n^*) \tag{36}$$

where z_n^* is the cluster than z_n is assigned to (at the current iteration):

$$z_n^* = \arg\max_k p(k|x_n, \theta) \tag{37}$$

$$= \arg\max_k \exp(-\tfrac{1}{2}||x_n - \mu_k||^2) \tag{38}$$

$$= \arg\min_k ||x_n - \mu_k||^2 \tag{39}$$

Thus the E step just requires finding the Euclidean distance between N data points and K cluster centers, which takes $O(NKD)$ time. This is faster than using full covariance matrices, where the E step takes $O(NKD^2)$ time. Furthermore, since we are just picking a nearest neighbor, various data structures, such as **kd-trees**, can be used to speed this up in high dimensions [Moo98]. In addition, one can use the triangle inequality to avoid some redundant computations [Elk03].

The K-means algorithm is faster than EM for GMMs with full covariance matrices, but its results tend to be not as good, since it cannot model the shape of the clusters (hence it does not learn a distance metric). Also, it is not robust to ambiguous points that are equidistant between clusters, because it uses hard assignment. (Probabalistic inference in the E step would softly assign such ambiguous points to multiple clusters.)

Although k-means is implemented in the Matlab statistics toolbox (function name `kmeans`), it is a very simple algorithm, so it is instructive to look at the source code. Below is my "bare bones" implementation. (The `sqdist` function is from Tom Minka's toolbox.)

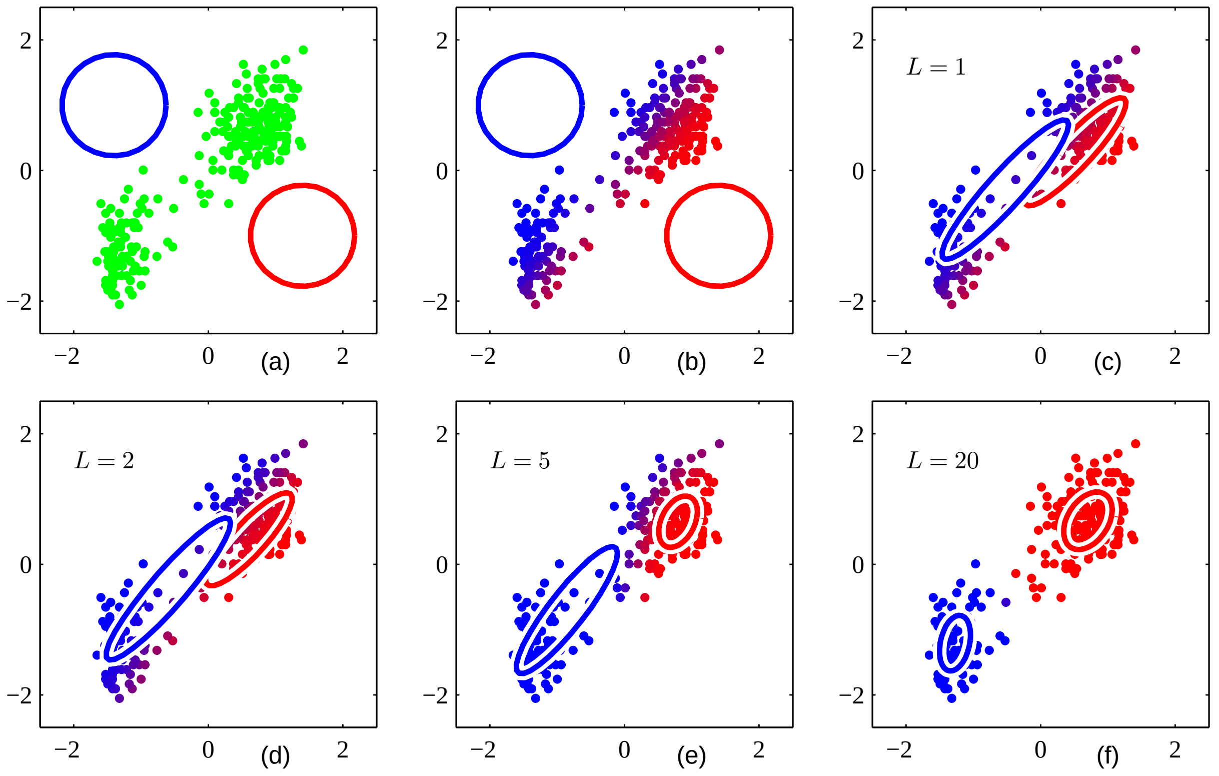

Figure 9: Illustration of the EM for a GMM applied to the Old Faithful data. Source: [Bis06] Figure 9.8.

```
function mu = kmeans(data, K, maxIter, thresh)

[N D] = size(data);
% initialization by picking random pixels
% mu(k,:) = k'th center
perm = randperm(N);
mu = data(perm(1:K),:);

converged = 0;
iter = 1;
while ~converged & (iter < maxIter)
  newmu = zeros(K,D);
  % dist(i,k) = squared distance from pixel i to center k
  dist = sqdist(data', mu');
  [junk, assign] = min(dist,[],2);
  for k=1:K
    newmu(k,:) = mean(data(assign==k,:), 1);
  end
  delta = abs(newmu(:) - mu(:));
  if max(delta./abs(mu(:))) < thresh
    converged = 1;
  end
  mu = newmu;
  iter = iter + 1
end

if ~converged
  error(sprintf('did not converge within %d iterations', maxIter))
end

%%%%%%%%%%

function m = sqdist(p, q, A)
% SQDIST      Squared Euclidean or Mahalanobis distance.
% SQDIST(p,q)   returns m(i,j) = (p(:,i) - q(:,j))'*(p(:,i) - q(:,j)).
% SQDIST(p,q,A) returns m(i,j) = (p(:,i) - q(:,j))'*A*(p(:,i) - q(:,j)).

%  From Tom Minka's lightspeed toolbox

[d, pn] = size(p);
[d, qn] = size(q);

if nargin == 2
  pmag = sum(p .* p, 1);
  qmag = sum(q .* q, 1);
  m = repmat(qmag, pn, 1) + repmat(pmag', 1, qn) - 2*p'*q;
  %m = ones(pn,1)*qmag + pmag'*ones(1,qn) - 2*p'*q;
else
  if isempty(A) | isempty(p)
    error('sqdist: empty matrices');
  end
  Ap = A*p;
  Aq = A*q;
  pmag = sum(p .* Ap, 1);
  qmag = sum(q .* Aq, 1);
  m = repmat(qmag, pn, 1) + repmat(pmag', 1, qn) - 2*p'*Aq;
end
```

The K centers μ_k learned by K-means are often called a **codebook**. This can then be used for discretizing data. This is called **vector quantization (VQ)**. The idea is that each new data point x is replaced by its nearest **prototype** μ_k. The result is a set of N discrete symbols. This can take much less space to store than the original data, and is therefore an example of **lossy data compression**. For example, consider the image in Figure 10(a). This is a $N = 512 \times 512 = 262,144$ pixel image, and each pixel is represented by three 8-bit numbers (each ranging from 0 to 255) that represent the green, red and blue intensity values for that pixel. The straightforward representation of this image therefore takes about $24N = 6,291,456$ bits. We applied vector quantization to this using $K = 5, 10, 15$ codewords using the code below.

```
% vqDemo

% Learn code book on small image (for speed)
A = double(imread('mandrill-small.tiff'));
figure; imshow(uint8(round(A)));
[nrows ncols ncolors] = size(A);
% data(i,:) = rgb value for pixel i
data = reshape(A, [nrows*ncols ncolors]);
maxIter = 100; % usually converges long before 100 iterations!
thresh = 1e-2;
K = 15;
mu = kmeansKPM(data, K, maxIter, thresh);

% Apply codebook to quantize  large image
B = double(imread('mandrill-large.tiff'));
imshow(uint8(round(B)));
[nrows ncols ncolors] = size(B);
data = reshape(B, [nrows*ncols ncolors]);
[N D] = size(data);
dist = sqdist(data', mu');
[junk, assign] = min(dist,[],2);
qdata = zeros(size(data));
for k=1:K
  ndx = find(assign==k);
  Nassign = length(ndx);
  qdata(ndx, :) = repmat(mu(k,:), Nassign, 1);
end
Qimg = reshape(qdata, [nrows ncols ncolors]);
figure; imshow(uint8(round(Qimg)));
title(sprintf('K=%d',K))
```

The results are shown in Figure 10. We need $\lceil \log_2 K \rceil$ bits to represnt each pixel (to specify which codebook entry to use), plus $24K$ bits to represent the μ_k themselves, totalling $24K + N \log_2 K$. If $K = 5$, the total space is 608,800 bits, which is about 10 times smaller. Greater compression could be achieved if we modelled spatial correlation between the pixels, e.g., if we encoded 5x5 blocks (as used by JPEG). For more information machine learning and data compression, see [Mac03].

When using Euclidean distance, as above, these centers are averages of the data points. For spaces in which such

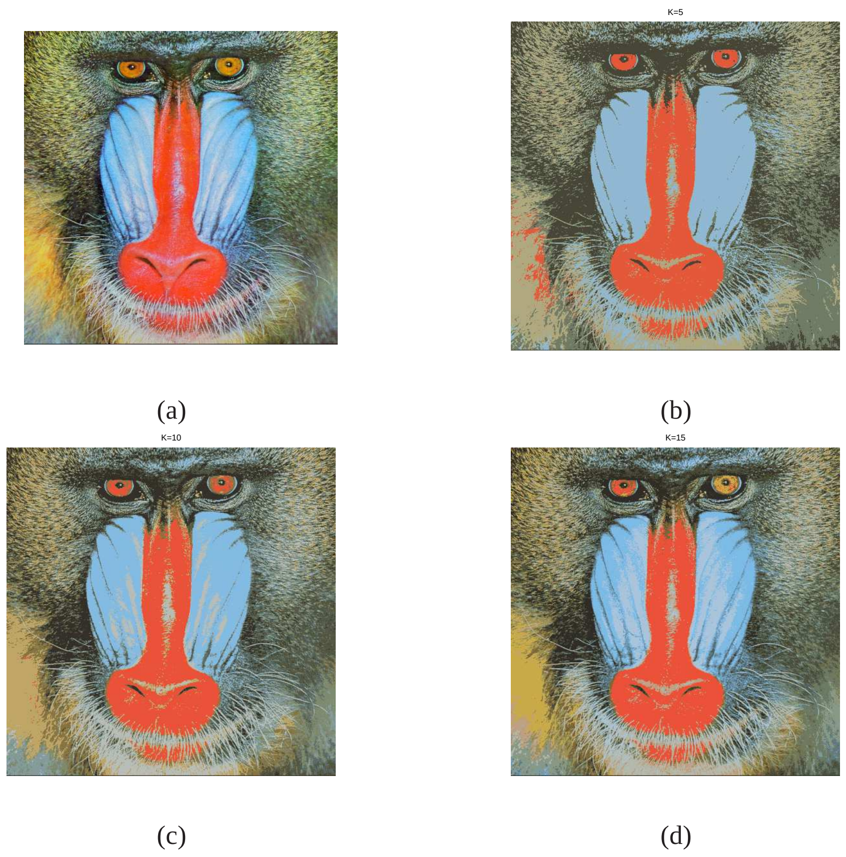

Figure 10: A color image compressed using vector quantization with a codebook of size K. (a) Original. (b) $K = 5$. (c) $K = 10$. (d) $K = 15$. Best viewed in colour. Produced by `vqDemo.m`.

averaging does not make sense, it is possible to replace the Euclidean distance measure by some other measure of similarity/ distance, $V(x_n, \mu_k)$, so the E step becomes

$$z_n^* = \arg\min_k V(x_n, \mu_k) \tag{40}$$

In the M step, we search over all N_k points assigned to cluster k to find the one with minimum distance to all the others, and that is chosen as the prototype μ_k. This algorithm is called the **K-medoids** algorithm.

6 EM variants

It is straightforward to modify EM to find MAP estimates instead of MLEs. Simply modify the M step to optimize

$$E_Z \log p(Z, x|\theta) + \log p(\theta) \tag{41}$$

The $p(\theta)$ term acts like a **regularizer**.

In the derivation of the M step for the GMM model, the new estimate of Σ_k relies on the new μ_k, but not vice versa, so we can solve them in sequence. If we have parameters that are mutually dependent, we can either optimize them jointly, or optimize them iteratively, one at a time; the latter is called **expectation conditional maximization (ECM)**.

Sometimes we cannot find the maximum of Q in closed form. In such cases, we may perform a partial maximization, i.e., we find θ^{new} st $Q(\theta^{new}, \theta^{old}) > Q(\theta^{old}, \theta^{old})$. This is called the **generalized EM (GEM)** algorithm.

When we have a lot of data (N is large), computing $p(z_n|x_n, \theta)$ for all n can be slow. In addition, it may be unncessary, since this estimate is likely to be poor if the parameters are bad (as they are initially). Hence we may take a **mini batch** of data, and just sum up over the case n in each batch. A batch size of $b = 10$ is common. This allows us to update the parameters more frequently. This version is called **incremental EM**. The special case of $b = 1$ is called **stochastic EM**, since we are approximating $E \log p(x, Z|\theta)$ by drawing a single data point.

In many models, computing $p(z_n|x_n, \theta)$ is intractable, because there are too many hidden variables, or their posterior is hard to compute. One approach is to use sampling to approximate $p(z_n|x_n, \theta)$. This version is called **Monte Carlo EM** (not to be confused with stochastic EM!).

7 Estimating the number of mixture components

How do we choose K, the number of mixture components? Maximum likelihood will always favor the largest possible value of K, since that gives the greatest number of parameters and maximizes the ability to fit the data. But this may result in overfitting. A simple alternative is to use **cross validation**, i.e., to find the value of K that maximizes the log likleihood of a validation set. Alternatively, we could try to find the K that maximizes some kind of **penalized likelihood function**. One example is the **Bayesian information criterion (BIC)** score, defined as

$$BIC(\theta) = \log p(D|\hat{\theta}^{ML}) - \frac{d}{2} \log N \tag{42}$$

where d is the "dimensionality" of the model, and N is the number of data cases. For simple models, d is the number of free parameters, but computing d for latent variable models is more difficult, since the parameters are not independent. A similar objective function is the **Akaike information criterion (AIC)**, defined as

$$AIC(\theta) = \log p(D|\hat{\theta}^{ML}) - d \tag{43}$$

Arguably the most conceptually elegant approach is to use **Bayesian model selection**, in which we compute the posterior over K:

$$\begin{aligned} p(K = k|D) &\propto p(K = k)p(D|K = k) & (44) \\ &= p(K = k) \left[\int p(D, \theta|K = k) d\theta \right] & (45) \end{aligned}$$

Unfortunately, computing the marginal likelihood term inside the brackets can be computationally expensive. However, one can use fast deterministic approximations, such as **variational Bayes** [Bis06].

The Bayesian approach offers an even more appealing strategy, which is to allow an "infinite" (i.e., unbounded) number of components, since in most applications we don't really believe there is a "true" number of clusters. By integrating over the parameters, such an infinite model will not overfit. As the data set gets larger and more heterogeneous, the number of components grows automatically. This can be modelled using **Dirichlet process mixture models** [Ras00], which is an example of a **non-parametric Bayesian model**.

8 Hierarchical clustering

GMMs can be used to perform clustering, but the clustering is "flat". Often we want to learn hierarchical clusters. The most popular approach is to use **agglomerative clustering**, which we will describe below.

Agglomerative clustering starts with N groups, each initially containing one training instance, merging similar groups to form larger groups, until there is a single group. This takes $O(N^2)$ time. At each iteration of an agglomerative algorithm, we choose the two closest groups to merge.

In **single link clustering**, the distance between two groups is defined as the distance between the two closest members of each group:

$$D_{SL}(G_i, G_j) = \min_{x^r \in G_i, x^s \in G_j} D(x^r, x^s) \tag{46}$$

where $D(x^r, x^s)$ is a distance measure between two feature vectors, such as Euclidean distance or the **city block distance**

$$D_{cb}(x^r, x^s) = \sum_{i=1}^{d} |x_i^r - x_i^s| \tag{47}$$

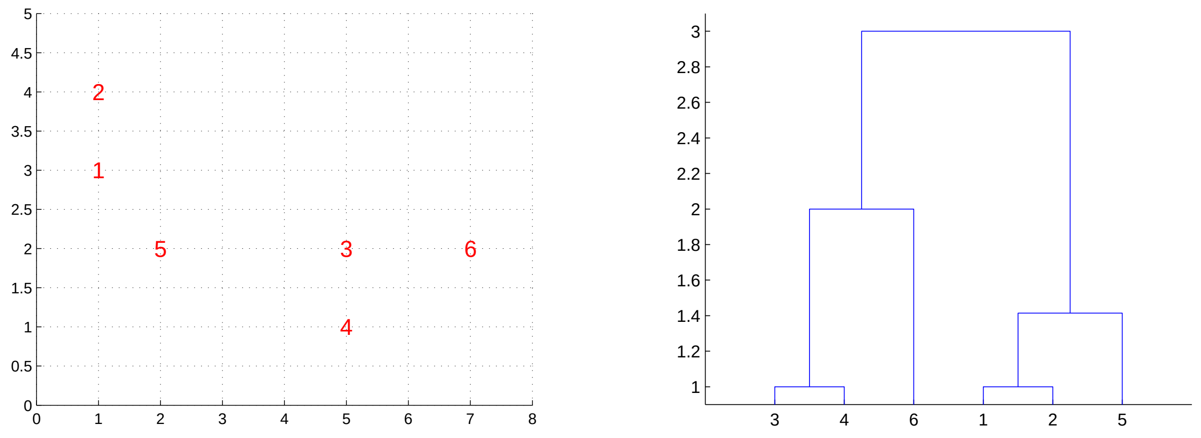

Figure 11: An example of single link clustering. Figure generated using `agglomDemo.m`.

In **complete link clustering**, the distance between two groups is defined as the distance between the two most distant pairs:

$$D_{CL}(G_i, G_j) = \max_{x^r \in G_i, x^s \in G_j} D(x^r, x^s) \tag{48}$$

Another option is **average link clustering**, which measures the average distance between all pairs.

The order in which groups are merged can be represented as a **dendrogram**. For example, in Figure 11, we see an example of single link clustering (using Euclidan distance) applied to a 2D dataset. Initially each group contains one item. The closest groups are $1, 2$ and $3, 4$ (both distance 1 apart), which get merged at step 1. At the next iteration, the $1, 2$ group gets merged with the 5 group (distance $\sqrt{2} = 1.41$). Then the $3, 4$ group merges with 6 (distance 2). The order of these events is shown in the tree on the right. One can then "cut" the tree at any height to get a desired number of groups. (Note that the tree built using single link clustering is the **minimal spanning tree** of the data.)

In Matlab, you can perform agglomerative lustering using the statistics toolbox using the `linkage` command (which defaults to single link). For example, you can reproduce Figure 11 using the code below.

```
% agglomDemo

X = [1 3;
     1 4;
     5 2;
     5 1;
     2 2;
     7 2];

figure;clf
axis on
grid on
N = size(X,1);
for i=1:N
  hold on
  h=text(X(i,1)-0.1, X(i,2), sprintf('%d', i));
  set(h,'fontsize',15,'color','r')
end
axis([0 8 0 5])

Y= pdist(X); % Euclidean distance
Z = linkage(Y); % single link
dendrogram(Z)
```

Note that agglomerative clustering is an algorithm, not a model, whereas GMM is a model, not an algorithm. **Model based clustering** has many advantages: the objective function (likelihood) is clearly defined (and can can be optimized via various algorithms, such as EM, gradient ascent); the model can be easily combined with other probabilistic models, such as PCA; one can compute the predictive distribution and do model comparison, etc. Surprisingly, there has been very little work on probabilistic hierarchical clustering models (but see [HG05] for a recent Bayesian approach).

References

[Bis06] C. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[CPW03] M. Carreira-Perpinan and C. Williams. An isotropic gaussian mixture can have more modes than components. Technical Report EDI-INF-RR-0185, School of Informatics, U. Edinburgh, 2003.

[Elk03] C. Elkan. Using the triangle inequality to accelerate k-means. In *Intl. Conf. on Machine Learning*, 2003.

[HG05] K. Heller and Z. Ghahramani. Bayesian Hierarchical Clustering. In *Intl. Conf. on Machine Learning*, 2005.

[Mac03] D. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[Moo98] Andrew W. Moore. Very fast EM-based mixture model clustering using multiresolution kd-trees. In *NIPS-11*, 1998.

[Ras00] C. Rasmussen. The infinite gaussian mixture model. In *NIPS*, 2000.