CSC2515 Winter 2015
Introduction to Machine Learning

Lecture 5: Clustering, mixture models, and EM

All lecture slides will be available as .pdf on the course website:
http://www.cs.toronto.edu/~urtasun/courses/CSC2515/CSC2515_Winter15.html

Overview

- Clustering with K-means, and a proof of convergence
- Clustering with K-medians
- Clustering with a mixture of Gaussians
- The EM algorithm, and a proof of convergence
- Variational inference

Unsupervised Learning

- Supervised learning algorithms have a clear goal: produce desired outputs for given inputs
- Goal of unsupervised learning algorithms (no explicit feedback whether outputs of system are correct) less clear:
  - Reduce dimensionality
  - Find clusters
  - Model data density
  - Find hidden causes
- Key utility
  - Compress data
  - Detect outliers
  - Facilitate other learning

Major types

Primary problems, approaches in unsupervised learning fall into three types:

1. Dimensionality reduction: represent each input case using a small number of variables (e.g., principal components analysis, factor analysis, independent components analysis)

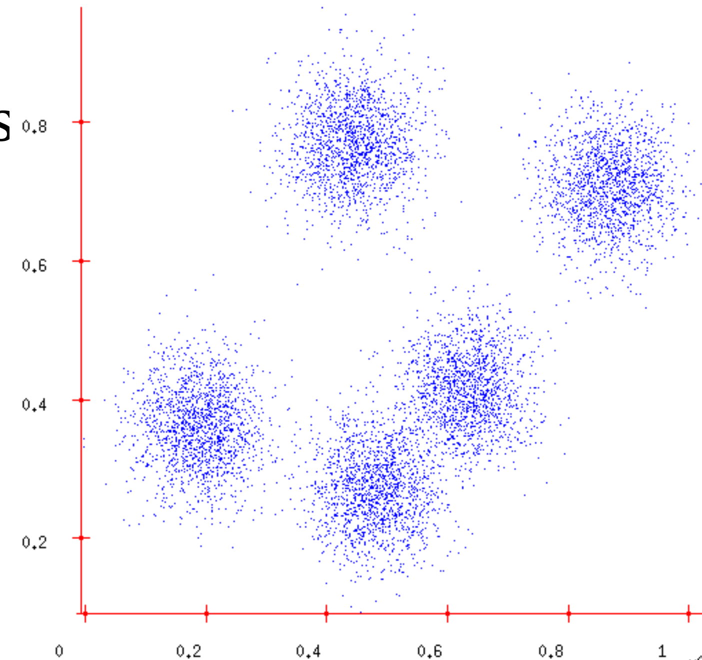
2. Clustering: represent each input case using a prototype example (e.g., k-means, mixture models)

3. Density estimation: estimating the probability distribution over the data space

Clustering

- Grouping N examples into K clusters one of canonical problems in unsupervised learning
- Motivations: prediction; lossy compression; outlier detection

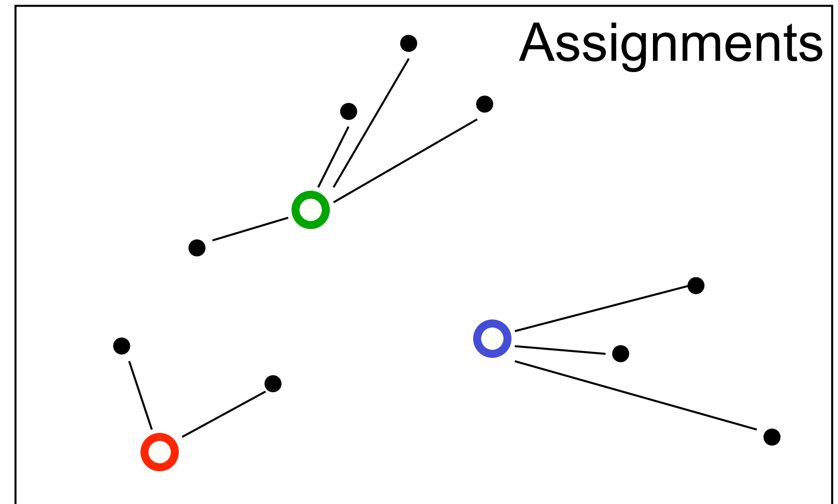
- We assume that the data was generated from a number of different classes. The aim is to cluster data from the same class together.
  - How many classes?
  - Why not put each datapoint into a separate class?
- What is the objective function that is optimized by sensible clusterings?



The K-means algorithm

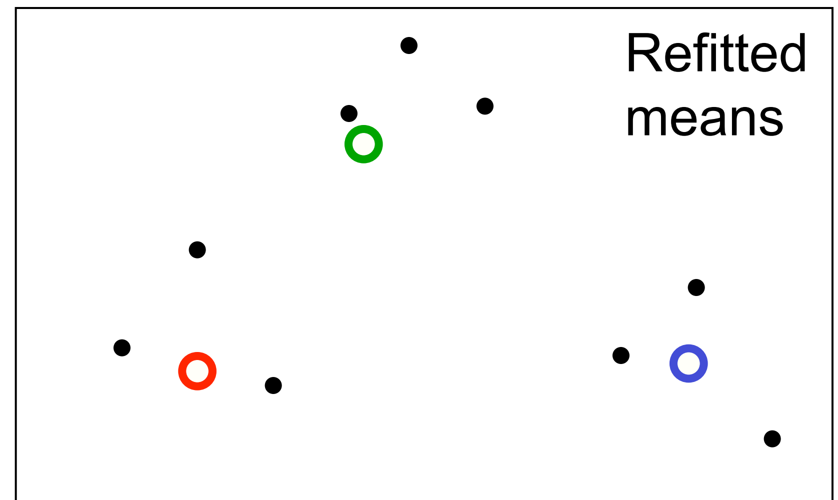
- Assume the data lives in a Euclidean space.
- Assume we want K classes.

- Initialization: randomly located cluster centers



The algorithm alternates between two steps:

Assignment step: Assign each datapoint to the closest cluster.



Refitting step: Move each cluster center to the center of gravity of the data assigned to it.

K-means algorithm

Initialization: Set K means $\{\mathbf{m}_k\}$ to random values

Assignment: Each datapoint n assigned to nearest mean

$$\hat{k}^{(n)} = \text{argmin}_k \{d(\mathbf{m}_k, \mathbf{x}^{(n)})\}$$

responsibilities:

$$r_k^{(n)} = 1 \Leftrightarrow \hat{k}^{(n)} = k$$

Update: Model parameters, means, are adjusted to match sample means of datapoints they are responsible for:

$$\mathbf{m}_k = \frac{\sum_n r_k^{(n)} \mathbf{x}^{(n)}}{\sum_n r_k^{(n)}}$$

Repeat assignment and update steps until assignments do not change

Questions about K-means

- Why does update set $\mathbf{m}_k$ to mean of assigned points?
- Where does distance d come from?
- What if we used a different distance measure?
- How can we choose best distance?
- How to choose K?
- How can we choose between alternative clusterings?
- Will it converge?

Hard cases – unequal spreads, non-circular spreads, inbetween points

Why K-means converges

- Whenever an assignment is changed, the sum squared distances of datapoints from their assigned cluster centers is reduced.
- Whenever a cluster center is moved the sum squared distances of the datapoints from their currently assigned cluster centers is reduced.
- Test for convergence: If the assignments do not change in the assignment step, we have converged (to at least a local minimum).

K-means: Details

- Objective: minimize sum squared distance of datapoints to their assigned cluster centers

$$E(\{\mathbf{m}\},\{\mathbf{r}\}) = \sum_n \sum_k r_k^{(n)} \left\| \mathbf{m}_k - \mathbf{x}^{(n)} \right\|^2$$

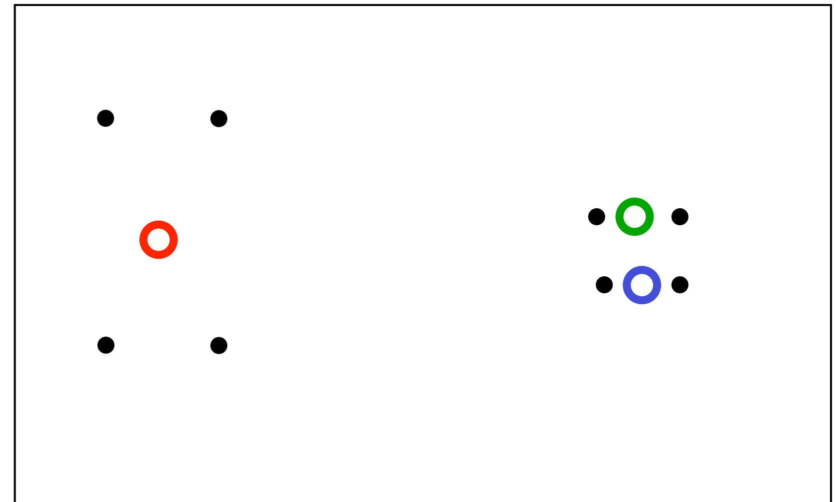
$$\text{s.t.} \sum_k r_k^{(n)} = 1, \forall n; \quad r_k^{(n)} \in \{0,1\}, \forall k, n$$

- Optimization method is a form of coordinate descent ("block coordinate descent")
  - Fix centers, optimize assignments (choose cluster whose mean is closest)
  - Fix assignments, optimize means (average of assigned datapoints)

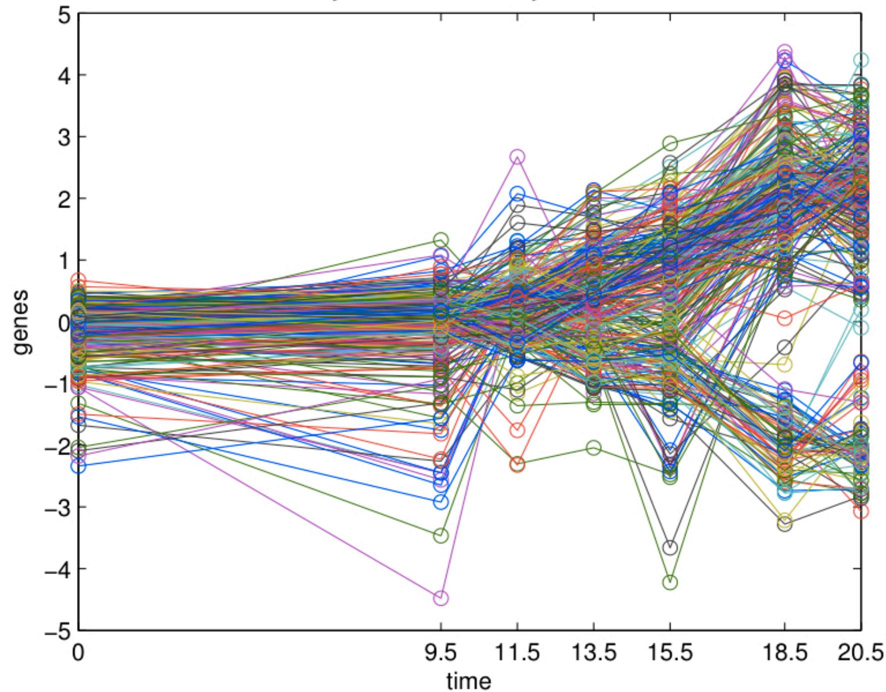
Local minima

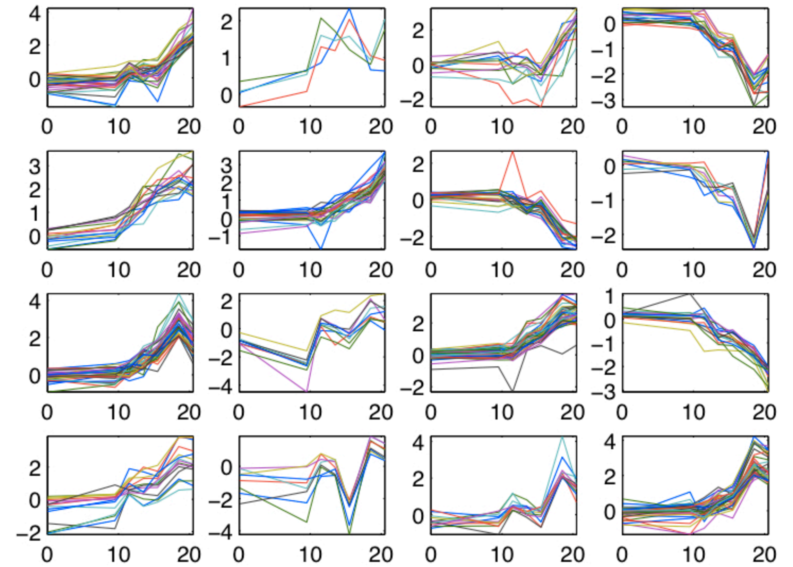
- There is nothing to prevent k-means getting stuck at local minima.
- We could try many random starting points
- We could try non-local split-and-merge moves: Simultaneously merge two nearby clusters and split a big cluster into two.

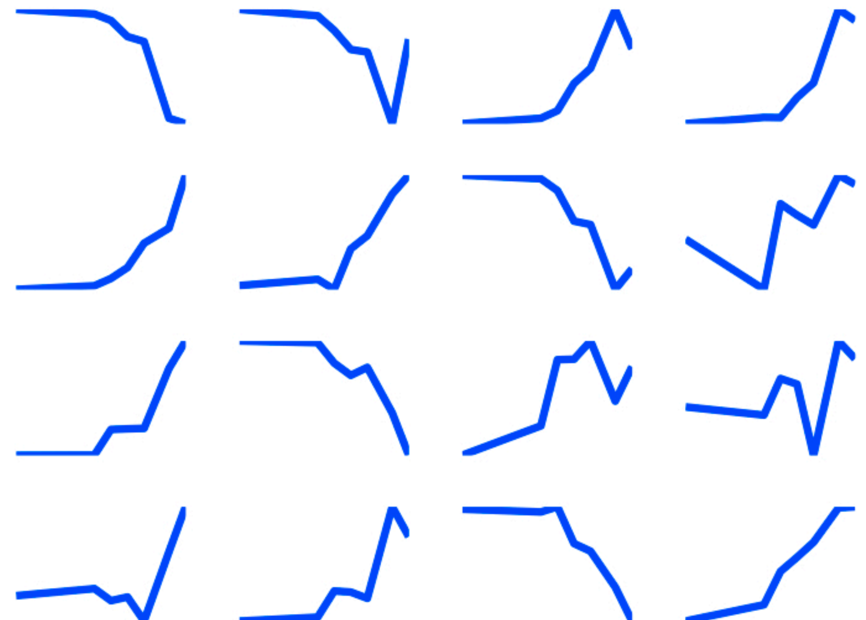
A bad local optimum



Application of K-Means Clustering







K-medioids

- K-Means: Choose number of clusters K; algorithm's primary aim is to strategically position these K means

- Alternative: allow each datapoint to potentially act as a cluster representative; algorithm assigns each point to one of these representatives (exemplars)

K-medioids algorithm

Initialization: Set random set K of datapoints as medioids

Assignment: Each datapoint c assigned to nearest medioid

$$\hat{k}^{(c)} = \operatorname{argmin}_k \{d(\mathbf{x}_k, \mathbf{x}^{(c)})\} \qquad r_k^{(c)} = 1 \Leftrightarrow \hat{k}^{(c)} = k$$

Update: For each medioid k, for each datapoint c, swap k and c and compute total cost J

Select configuration with lowest cost

$$J(\{\mathbf{x}\}, \{\mathbf{r}\}) = \sum_c \sum_k r_k^c d(\mathbf{x}^{(c)}, \mathbf{x}_k)$$

Repeat assignment and update steps until assignments do not change

K-medioids vs. K-means

- Both partition data into K partitions
- Both can utilize various distance functions, but K-medioids can pre-compute pairwise distances

- K-medioids chooses datapoints as centers (medioids/exemplars), while K-means allows the means to be arbitrary locations → discrete vs. continuous optimization

- K-medioids more robust to noise and outliers

Soft k-means

Instead of making hard assignments of datapoints to clusters, we can make soft assignments. One cluster may have a responsibility of .7 for a datapoint and another may have a responsibility of .3.

- Allows a cluster to use more information about the data in the refitting step.
- What happens to our convergence guarantee?
- How do we decide on the soft assignments?

Soft K-means algorithm

Initialization: Set K means $\{\mathbf{m}_k\}$ to random values

Assignment: Each datapoint n given soft 'degree of assignment' to each cluster mean k, based on responsibilities

$$r_k^{(n)} = \frac{\exp[-\beta d(\mathbf{m}_k, \mathbf{x}^{(n)})]}{\sum_{k'} \exp[-\beta d(\mathbf{m}_{k'}, \mathbf{x}^{(n)})]}$$

Update: Model parameters, means, are adjusted to match sample means of datapoints they are responsible for:

$$\mathbf{m}_k = \frac{\sum_n r_k^{(n)} \mathbf{x}^{(n)}}{\sum_n r_k^{(n)}}$$

Repeat assignment and update steps until assignments do not change

Questions about soft K-means

- How to set β?
- What about problems with elongated clusters?
- Clusters with unequal weight and width

Latent variable models

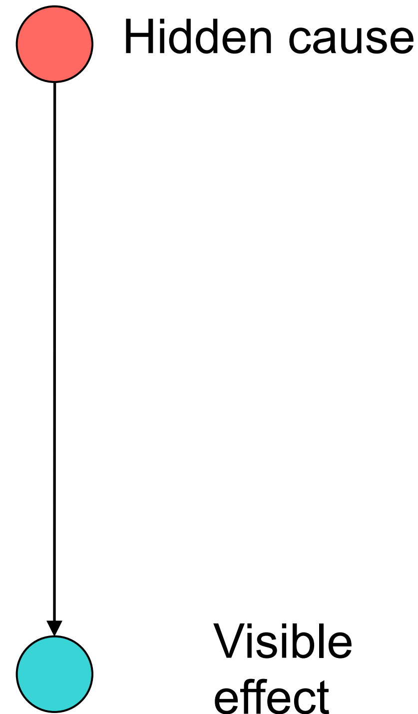
- Adopt a different view of clustering, in terms of a model with *latent variables*: variables that are always unobserved

- We may want to intentionally introduce latent variables to model complex dependencies between variables without looking at dependencies between them directly -- this can actually simplify the model
- Form of divide-and-conquer: use simple parts to build complex models (e.g., multimodal densities, or piecewise-linear regression)

Mixture models

- Most common example is a mixture model: most basic form of latent variable model, with single discrete latent variable

- We have defined the hidden cause to be discrete: a multinomial random variable
- And the observation is Gaussian
- The model allows for other distributions
- Example: Bernoulli observations
- Another example: Continuous hidden (latent) variable – see next lecture



- But these are only the simplest models: can add many hidden & visible nodes, layers

Learning is harder with latent variables

- In fully observed settings, probability model is a product, so the log likelihood is a sum, terms decouple:

$$\ell(\theta; D) = \sum_{c} \log p(\mathbf{x}^c, \mathbf{y}^c \mid \theta)$$

$$= \sum_{c} \log p(\mathbf{x}^c \mid \theta_x) + \sum_{c} \log p(\mathbf{y}^c \mid \mathbf{x}^c, \theta_y)$$

- With latent variables, probability contains a sum so the log-likelihood has all parameters coupled together

$$\ell(\theta; D) = \sum_{c} \log \sum_{\mathbf{z}} p(\mathbf{x}^c, \mathbf{z} \mid \theta)$$

$$= \sum_{c} \log \sum_{\mathbf{z}} p(\mathbf{z} \mid \theta_z) p(\mathbf{x}^c \mid \mathbf{z}, \theta_x)$$

Direct learning in mixtures of Gaussians

- We can treat likelihood as an objective function and try to optimize it as a function of θ by taking gradients (as we did before, for example in neural networks)
- If you work thru the gradients, you'll find that

$$\frac{\partial \ell(\theta)}{\partial \theta} = \sum_{\mathrm{k}} \alpha_k r_k \frac{\partial \ell_k(\theta)}{\partial \theta_k} = \sum_k \alpha_k r_k \frac{\partial \log p_k(\mathbf{x} \mid \theta_k)}{\partial \theta}$$

- In a mixture of Gaussians for example:

$$\frac{\partial \ell(\theta)}{\partial \boldsymbol{\mu}_k} = -\sum_{\mathrm{k}} \alpha_k r_k (\mathbf{x} - \boldsymbol{\mu}_k) / \sigma_k^2$$

- To use optimization methods (e.g., conjugate gradient), have to ensure that parameters respect constraints – reparametrize in terms of unconstrained values

EM: An alternative learning approach

- Use the posterior weightings to softly label the data
- Then solve for parameters given these current weightings, and recalculate posteriors (weights) given new parameters
- Expectation-Maximization is a form of bound optimization, as opposed to gradient descent
- With respect to latent variables, guessing their values makes the learning fully-observed

$$\ell(\theta; D) = \sum_{c} \log \sum_{\mathbf{z}} p(\mathbf{z} \mid \theta_z) p(\mathbf{x}^c \mid \mathbf{z}, \theta_x)$$

$$\ell(\theta; D) = \sum_{c} \log p(\mathbf{x}^c, \mathbf{z} \mid \theta) = \sum_{c} \log p(\mathbf{z} \mid \theta_z) + \log p(\mathbf{x}^c \mid \mathbf{z}, \theta_x)$$

- Note: EM is not a cost function, such as cross-entropy; and EM is not a model, such as a mixture-of-Gaussians
- With latent variables, probability contains a sum so the log-likelihood has all parameters coupled together

Graphical model view of mixture models

- Each node is a random variable
- Blue node: observed variable (data, aka visibles)
- Red node: hidden variable [cluster assignment]
- The model defines a probability distribution over all the nodes
- The model generates data by picking state for hidden node based on prior
- The distribution over leaf node (data) is defined by its parent(s).

The mixture of Gaussians generative model

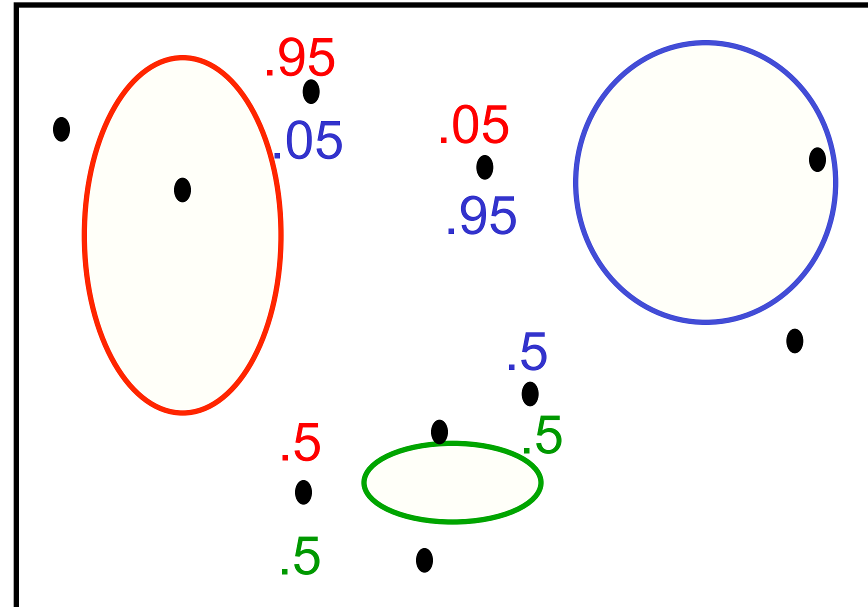
- First pick one of the k Gaussians with a probability that is called its "mixing proportion".
- Then generate a random point from the chosen Gaussian.
- The probability of generating the exact data we observed is zero, but we can still try to maximize the probability density.
  - Adjust the means of the Gaussians
  - Adjust the variances of the Gaussians on each dimension.
  - Adjust the mixing proportions of the Gaussians.

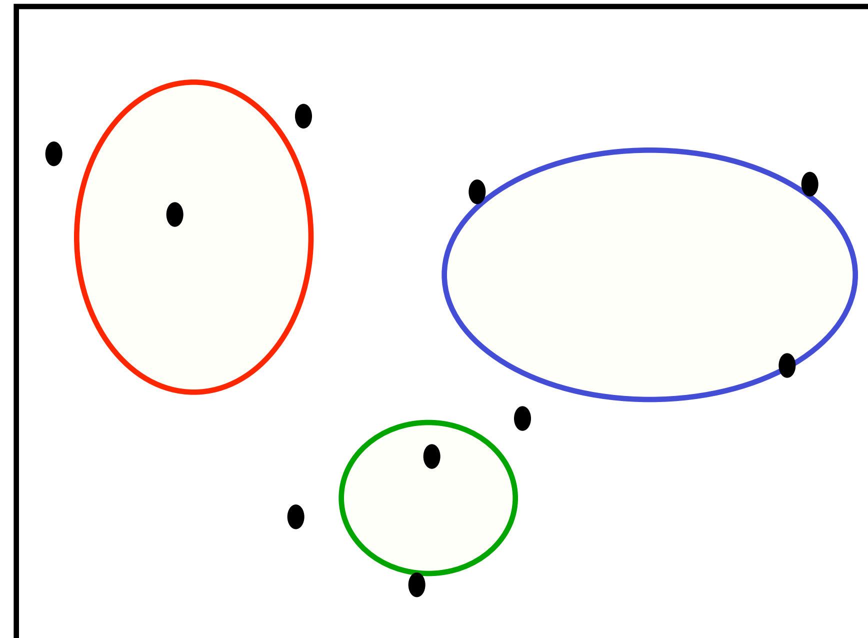
Fitting a mixture of Gaussians

Optimization uses the Expectation Maximization algorithm, which alternates between two steps:

E-step: Compute the posterior probability that each Gaussian generates each datapoint.

M-step: Assuming that the data really was generated this way, change the parameters of each Gaussian to maximize the probability that it would generate the data it is currently responsible for.





The E-step: Computing responsibilities

- In order to adjust the parameters, we must first solve the inference problem: Which Gaussian generated each datapoint?
  - We cannot be sure, so it's a distribution over all possibilities.
- Use Bayes theorem to get posterior probabilities

Posterior for Gaussian i

Prior for Gaussian i

$$p(i \mid \mathbf{x}^{(c)}) = \frac{p(i)p(\mathbf{x}^{(c)} \mid i)}{p(\mathbf{x}^{(c)})}$$

Bayes theorem

$$p(\mathbf{x}^{(c)}) = \sum_j p(j)p(\mathbf{x}^{(c)} \mid j)$$

$$p(i) = \pi_i$$

Mixing proportion

$$p(\mathbf{x}^{(c)} \mid i) = \prod_{d=1}^{d=D} \frac{1}{\sqrt{2\pi}\sigma_{i,d}} e^{-\frac{\| x_d^{(c)} - \mu_{i,d} \|^2}{2\sigma_{i,d}^2}}$$

Product over all data dimensions

The M-step: Computing new mixing proportions

- Each Gaussian gets a certain amount of posterior probability for each datapoint.
- The optimal mixing proportion to use (given these posterior probabilities) is just the fraction of the data that the Gaussian gets responsibility for.

Posterior for Gaussian i

Data for training case c

$$\pi_i^{new} = \frac{\sum_{c=1}^{c=N} p(i \mid \mathbf{x}^{(c)})}{N}$$

Number of training cases

More M-step: Computing the new means

- We just take the center-of gravity of the data that the Gaussian is responsible for.
  - Just like in K-means, except the data is weighted by the posterior probability of the Gaussian.
  - Guaranteed to lie in the convex hull of the data
    - Could be big initial jump

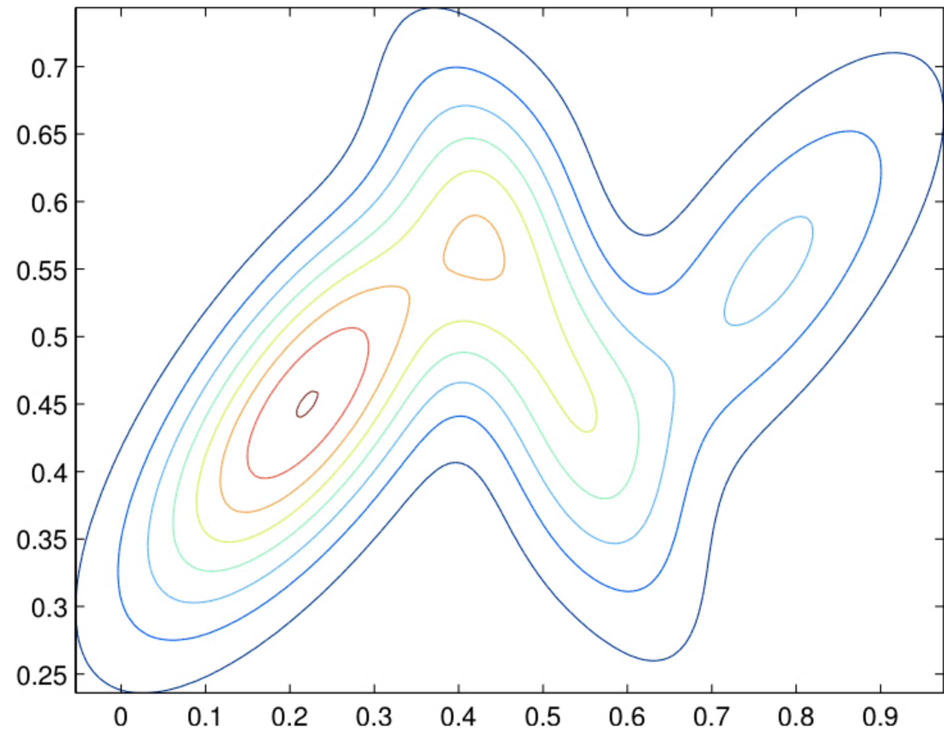
$$\mu_i^{new} = \frac{\sum_c p(i \mid \mathbf{x}^{(c)})\, \mathbf{x}^{(c)}}{\sum_c p(i \mid \mathbf{x}^{(c)})}$$

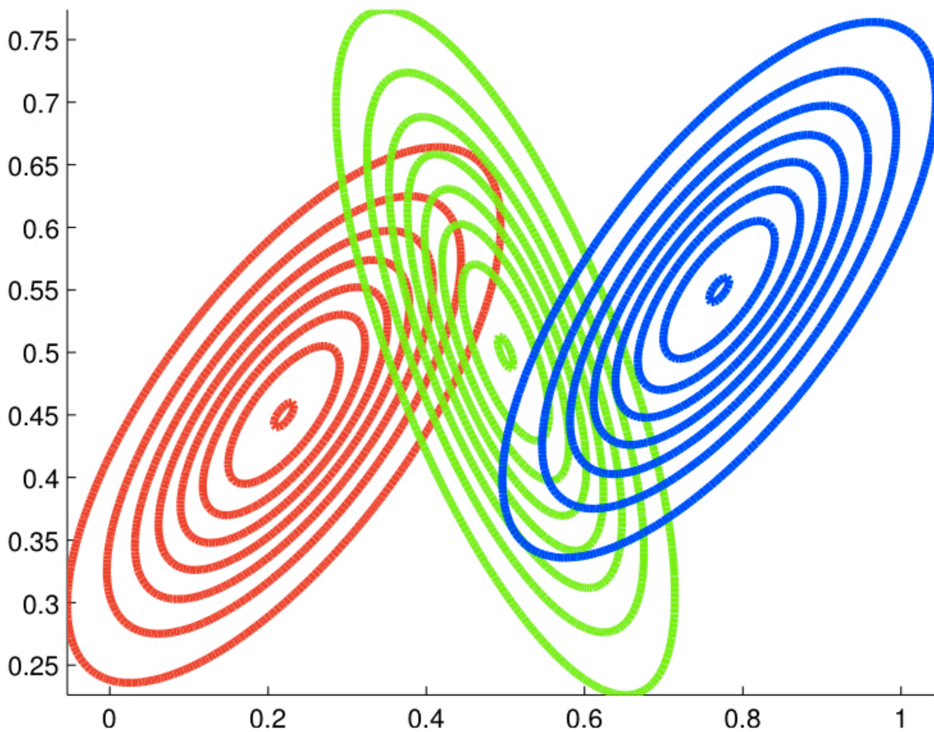
More M-step: Computing the new variances

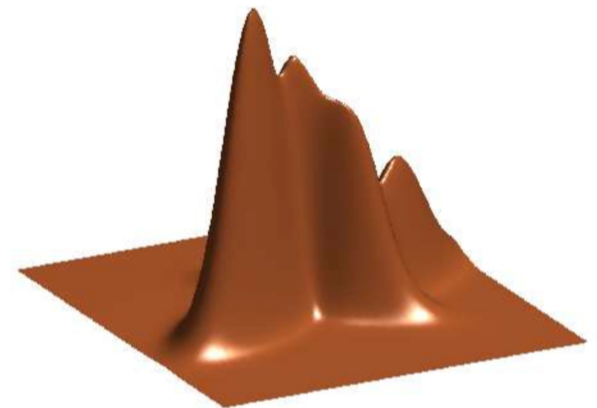
- We fit the variance of each Gaussian, i, on each dimension, d, to the posterior-weighted data
  - Its more complicated if we use a full-covariance Gaussian that is not aligned with the axes.

$$\sigma_{i,d}^2 = \frac{\sum_c p(i \mid \mathbf{x}^{(c)}) \, \| x_d^{(c)} - \mu_{i,d}^{new} \|^2}{\sum_c p(i \mid \mathbf{x}^{(c)})}$$

Visualizing a Mixture of Gaussians







Mixture of Gaussians vs. K-means

- EM for mixtures of Gaussians is just like a soft version of K-means, with fixed priors and covariance

- Instead of hard assignments in the E-step, we do soft assignments based on the softmax of the squared distance from each point to each cluster.

- Each center moved by weighted means of the data, with weights given by soft assignments

- In K-means, weights are 0 or 1

How do we know that the updates improve things?

- Updating each Gaussian definitely improves the probability of generating the data if we generate it from the same Gaussians after the parameter updates.
  - But we know that the posterior will change after updating the parameters.
- A good way to show that this is OK is to show that there is a single function that is improved by both the E-step and the M-step.
  - The function we need is called Free Energy.

Deriving variational free energy

- We can derive variational free energy as the objective function that is minimized by both steps of the Expectation and Maximization algorithm (EM).

Why EM converges

- Free energy F is a cost function that is reduced by both the E-step and the M-step.

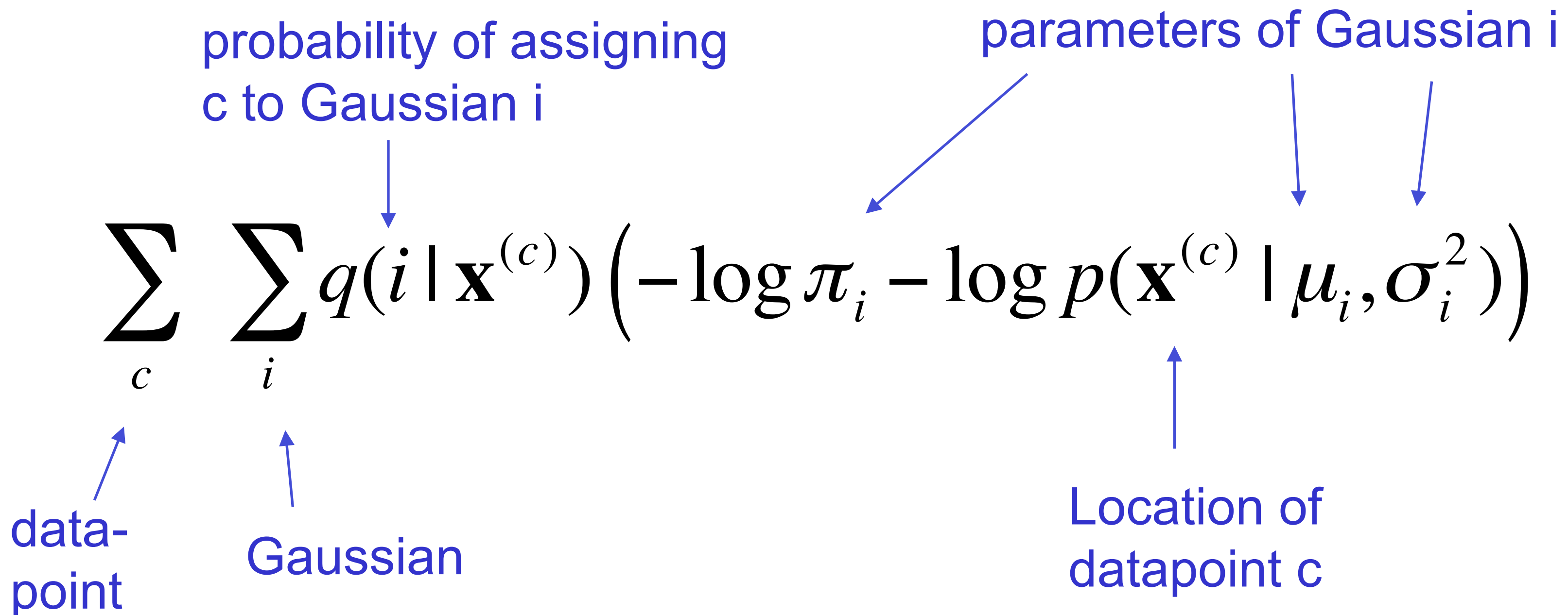
  Cost = F = expected energy – entropy

- The expected energy term measures how difficult it is to generate each datapoint from the Gaussians it is assigned to. It would be happiest assigning each datapoint to the Gaussian that generates it most easily (as in K-means).

- The entropy term encourages "soft" assignments. It would be happiest spreading the assignment probabilities for each datapoint equally between all the Gaussians.

The expected energy of a datapoint

- The expected energy of datapoint c is the average negative log probability of generating the datapoint
  - The average is taken using the probabilities of assigning the datapoint to each Gaussian.
  - Can use any probabilities we like – use some distribution q (more on this soon)



The entropy term

- This term wants the assignment probabilities to be as uniform as possible (max. entropy)
- It fights the expected energy term.

$$\text{entropy} = -\sum_{c} \sum_{i} q(i \mid \mathbf{x}^{(c)}) \log q(i \mid \mathbf{x}^{(c)})$$

log probabilities are always negative

The E-step chooses assignment probabilities that minimize F (with parameters of Gaussians fixed)

- How do we find assignment probabilities for a datapoint that minimize the cost and sum to 1?
- The optimal solution to the trade-off between expected energy and entropy is to make the probabilities be proportional to the exponentiated negative energies:

$$\text{energy of assigning } c \text{ to } i \;=\; -\log \pi_i - \log p(\mathbf{x}^{(c)} \mid \mu_i, \sigma_i^2)$$

$$\text{optimal value of } q(i \mid \mathbf{x}^{(c)}) \;\propto\; \exp(-\text{energy(c,i)}) \propto\; \pi_i \; p(\mathbf{x}^{(c)} \mid i)$$

- So using the posterior probabilities as assignment probabilities minimizes the cost function!

M-step chooses parameters that minimize F (with the assignment probabilities held fixed)

- This is easy. We just fit each Gaussian to the data weighted by the assignment probabilities that the Gaussian has for the data.
  - The entropy term is unaffected (since it only depends on the assignment probabilities)
  - When you fit a Gaussian to data you are maximizing the log probability of the data given the Gaussian. This is the same as minimizing the energies of the datapoints that the Gaussian is responsible for.
  - If a Gaussian is assigned a probability of 0.7 for a datapoint the fitting treats it as 0.7 of an observation.

- Since both the E-step and the M-step decrease the same cost function, EM converges.

Summary: EM is coordinate descent in Free Energy

$$F(\mathbf{x}^{(c)}) = \sum_i q(i \mid \mathbf{x}^{(c)})\Big(-\log \pi_i - \log p(\mathbf{x}^{(c)} \mid i)\Big) \quad - \quad \sum_i q(i \mid \mathbf{x}^{(c)})\Big(-\log q(i \mid \mathbf{x}^{(c)})\Big)$$

- Think of each different setting of the hidden and visible variables as a "configuration". The energy of the configuration has two terms:
  - The log prob of generating the hidden values
  - The log prob of generating the visible values from the hidden ones
- The E-step minimizes F by finding the best distribution over hidden configurations for each data point.
- The M-step holds the distribution fixed and minimizes F by changing the parameters that determine the energy of a configuration.

Recap: EM algorithm

- A way of maximizing likelihood for latent variable models
- EM is general algorithm: finds ML parameters when the original hard problem can be broken up into two easier pieces:
  - Infer distribution over hidden variables (given current parameters)
  - Using this complete data, find the maximum likelihood parameter estimates
- Allows constraints to be enforced easily (versus Lagrange multipliers in gradient descent)
- Works fine if distribution over hidden variables easy to compute

The advantage of using F to understand EM

- There is clearly no need to use the optimal distribution over hidden configurations.
  - We can use any distribution that is convenient so long as:
    - we always update the distribution in a way that improves F
    - We change the parameters to improve F given the current distribution.
- This is very liberating. It allows us to justify all sorts of weird algorithms**: variational inference**

A trade-off between how well the model fits the data and the accuracy of inference

parameters

data

approximating posterior distribution

true posterior distribution

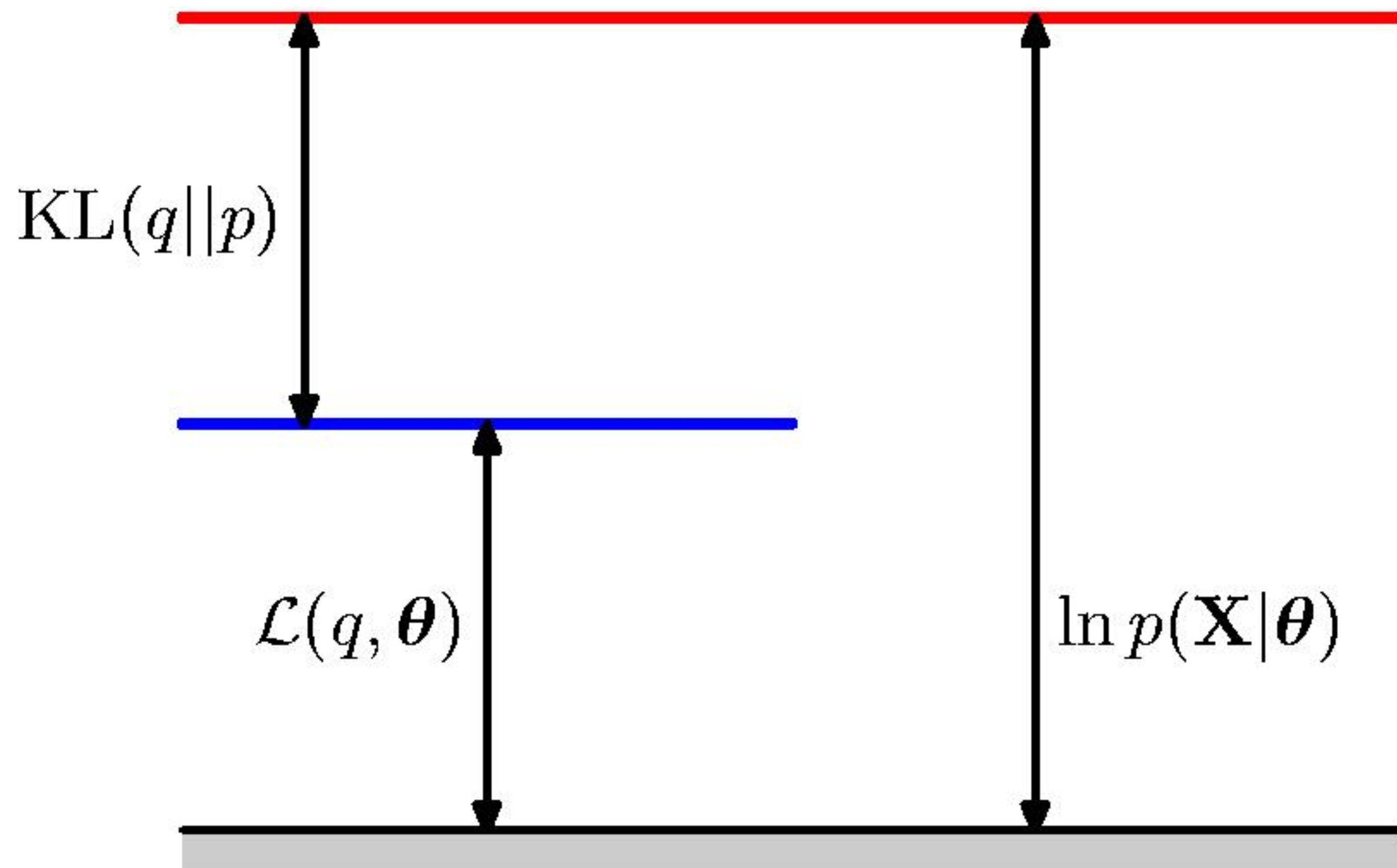
$$-F(q,\theta) = \sum_{d} \log p(d \mid \theta) - KL\big(q(d) \,\|\, p(d)\big)$$

new objective function

How well the model fits the data

The inaccuracy of inference

This makes it feasible to fit very complicated models, but the approximations that are tractable may be poor.


$\mathrm{KL}(q||p)$
$\mathcal{L}(q, \boldsymbol{\theta})$
$\ln p(\mathbf{X}|\boldsymbol{\theta})$