Mixture density estimation

Nuno Vasconcelos

ECE Department, UCSD

Recall

- last class, we will have "Cheetah Day"
- what:
  - 4 teams, average of 6 people
  - each team will write a report on the 4 cheetah problems
  - each team will give a presentation on one of the problems
- I am waiting to hear on the teams

Plan for today

- we have talked a lot about the BDR and methods based on density estimation
- practical densities are not well approximated by simple probability models
- last lecture: alternative way is to go non-parametric
  - kernel-based density estimates
  - “place a a pdf (kernel) on top of datapoint”
- today: mixture models
  - similar, but restricted number of kernels
  - likelihood evaluation significantly simpler
  - parameter estimation much more complex

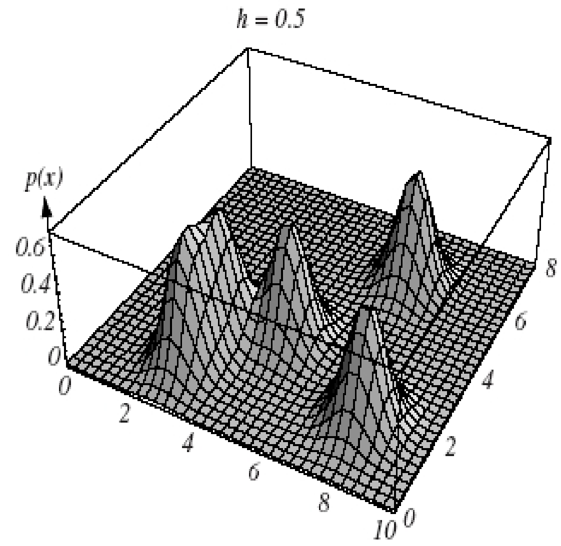
Kernel density estimates

- estimate density with

$$P_{\mathbf{X}}(\mathbf{x}) = \frac{1}{nh^d} \sum_{i=1}^{n} \phi\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right)$$

- where $\phi(x)$ is a kernel, the most popular is the Gaussian

$$\phi(\mathbf{x}) = \frac{1}{\sqrt{2\pi}^d} e^{-\frac{1}{2}\mathbf{x}^T\mathbf{x}}$$



- sum of n Gaussians centered at X_i
- Gaussian kernel density estimate:
  - *"approximate the pdf of X with a sum of Gaussian bumps"*

Kernel bandwidth

- back to the generic model

$$P_{\mathbf{X}}(\mathbf{x}) = \frac{1}{nh^d} \sum_{i=1}^{n} \phi\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right)$$

- what is the role of *h* (bandwidth parameter)?
- defining

$$\delta(\mathbf{x}) = \frac{1}{h^d} \phi\left(\frac{\mathbf{x}}{h}\right)$$

- we can write

$$P_{\mathbf{X}}(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \delta\left(\mathbf{x} - \mathbf{x}_i\right)$$

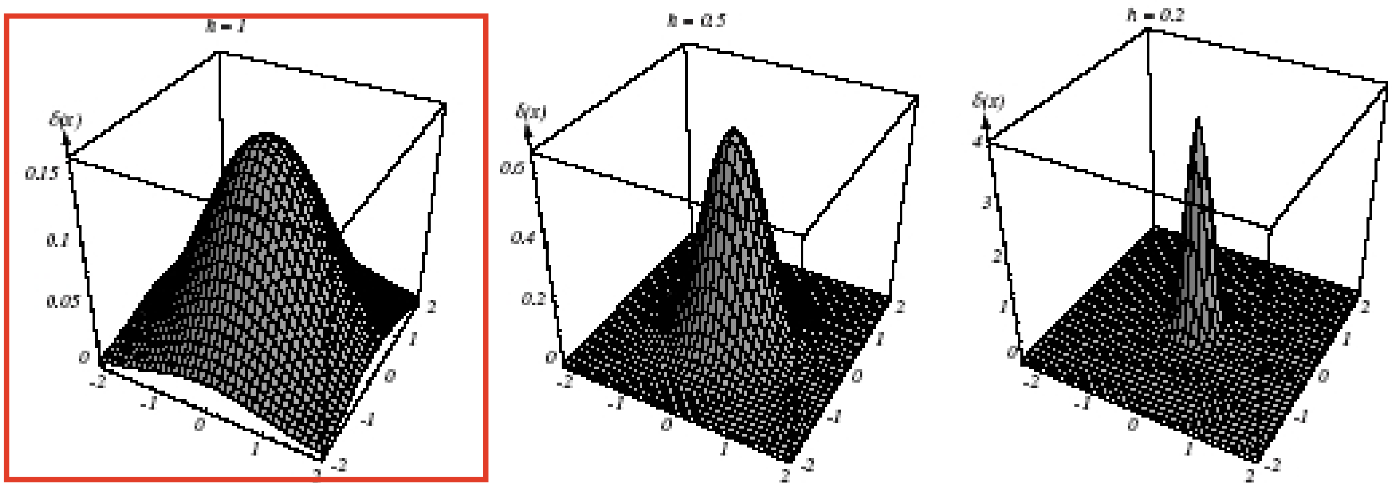
- i.e. a sum of translated replicas of $\delta(x)$

Kernel bandwidth

- *h* has two roles:
  1. rescale the *x*-axis
  2. rescale the amplitude of $\delta(x)$

$$\delta(\mathbf{x}) = \frac{1}{h^d}\phi\left(\frac{\mathbf{x}}{h}\right)$$

- this implies that for large *h*:
  1. $\delta(x)$ has low amplitude
  2. iso-contours of *h* are quite distant from zero
     (*x* large before $\phi(x/h)$ changes significantly from $\phi(0)$)

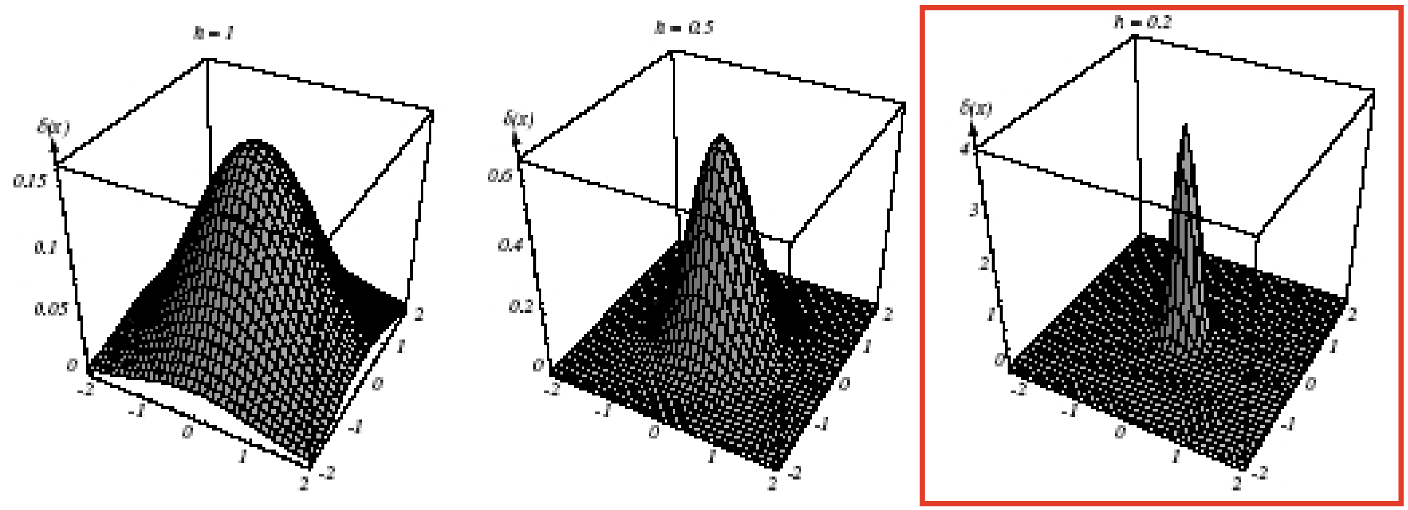


Kernel bandwidth

- for small *h*:

$$\delta(\mathbf{x}) = \frac{1}{h^d}\phi\left(\frac{\mathbf{x}}{h}\right)$$

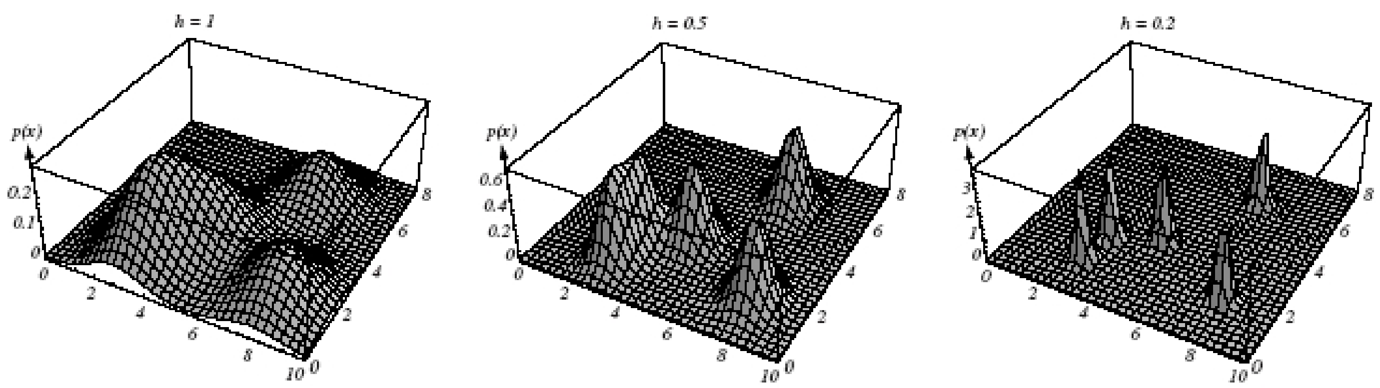
  1. $\delta(x)$ has large amplitude
  2. iso-contours of *h* are quite close to zero
     (*x* small before $\phi(x/h)$ changes significantly from $\phi(0)$)



- what is the impact of this on the quality of the density estimates?

Kernel bandwidth

- it controls the smoothness of the estimate
  - as h goes to zero we have a sum of delta functions (very "spiky" approximation)
  - as h goes to infinity we have a sum of constant functions (approximation by a constant)
  - in between we get approximations that are gradually more smooth



Bias and variance

- the bias and variance are given by

$$E_{\mathbf{X}_1,\ldots\mathbf{X}_n}[\widehat{P}_{\mathbf{X}}(\mathbf{x})] \;=\; \delta(\mathbf{x}) \odot P_{\mathbf{X}}(\mathbf{x})$$

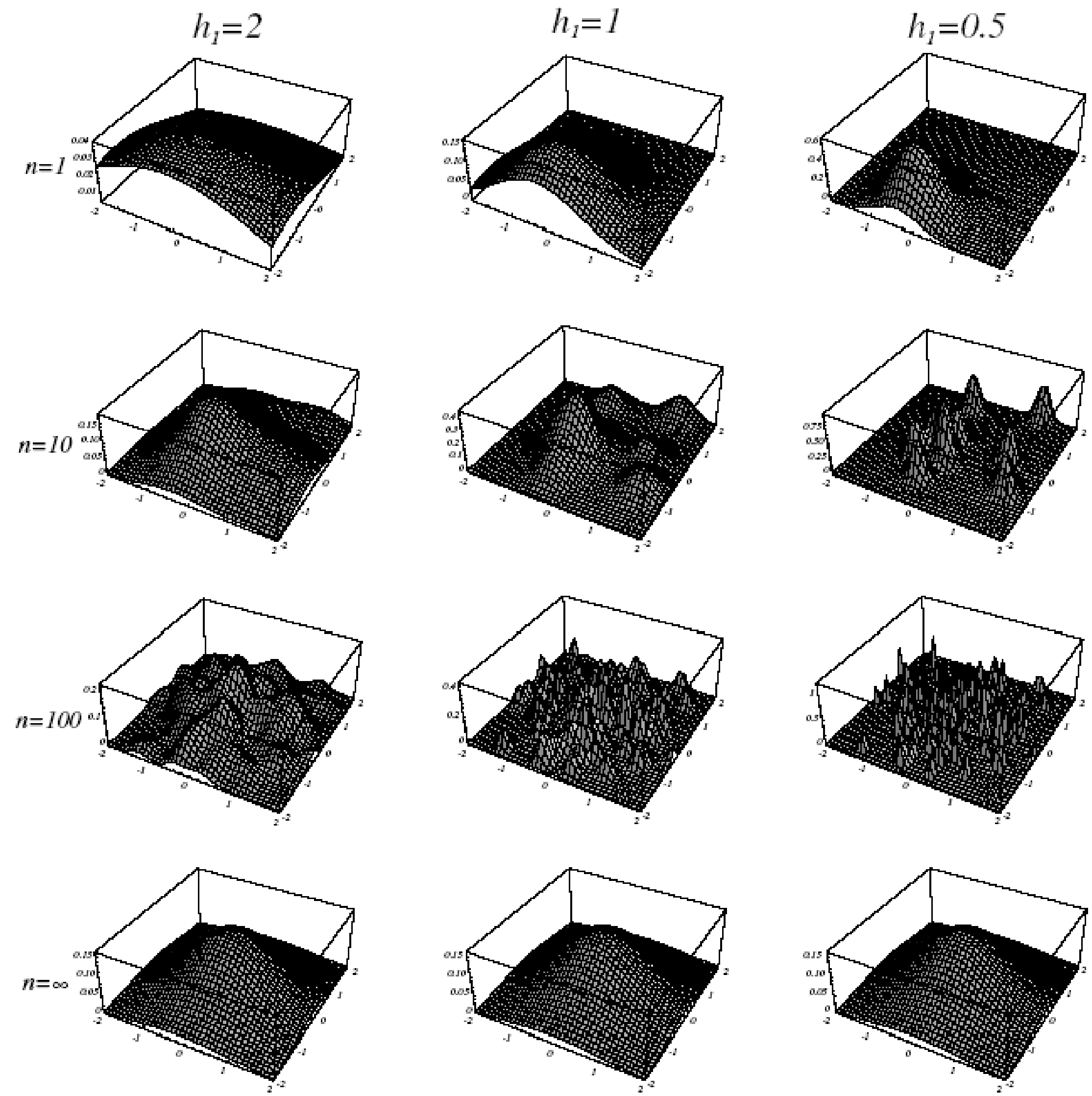
$$var_{\mathbf{X}_1,\ldots\mathbf{X}_n}[\widehat{P}_{\mathbf{X}}(\mathbf{x})] =$$

$$\leq \; \frac{1}{nh^d} \sup\left[\phi\left(\frac{\mathbf{x}}{h}\right)\right] E_{\mathbf{X}_1,\ldots\mathbf{X}_n}[\widehat{P}_{\mathbf{X}}(\mathbf{x})]$$

- this means that:
  - to obtain small bias we need *h ~ 0*
  - to obtain small variance we need *h* infinite

Example

- example: fit to N(0,I) using $h = h_1/n^{1/2}$
- small h: spiky
- need a lot of points to converge (variance)
- large h: approximate N(0,I) with a sum of Gaussians of larger covariance
- will never have zero error (bias)



Optimal bandwidth

- we would like
  - *h ~ 0* to guarantee zero bias
  - zero variance as *n* goes to infinity
- solution:
  - make h a function of n that goes to zero
  - since variance is $O(1/nh^d)$ this is fine if nh^d goes to infinity
- hence, we need

$$\lim_{n\to\infty} h(n) = 0 \text{ and } \lim_{n\to\infty} nh(n) = \infty$$

- optimal sequences exist, e.g.

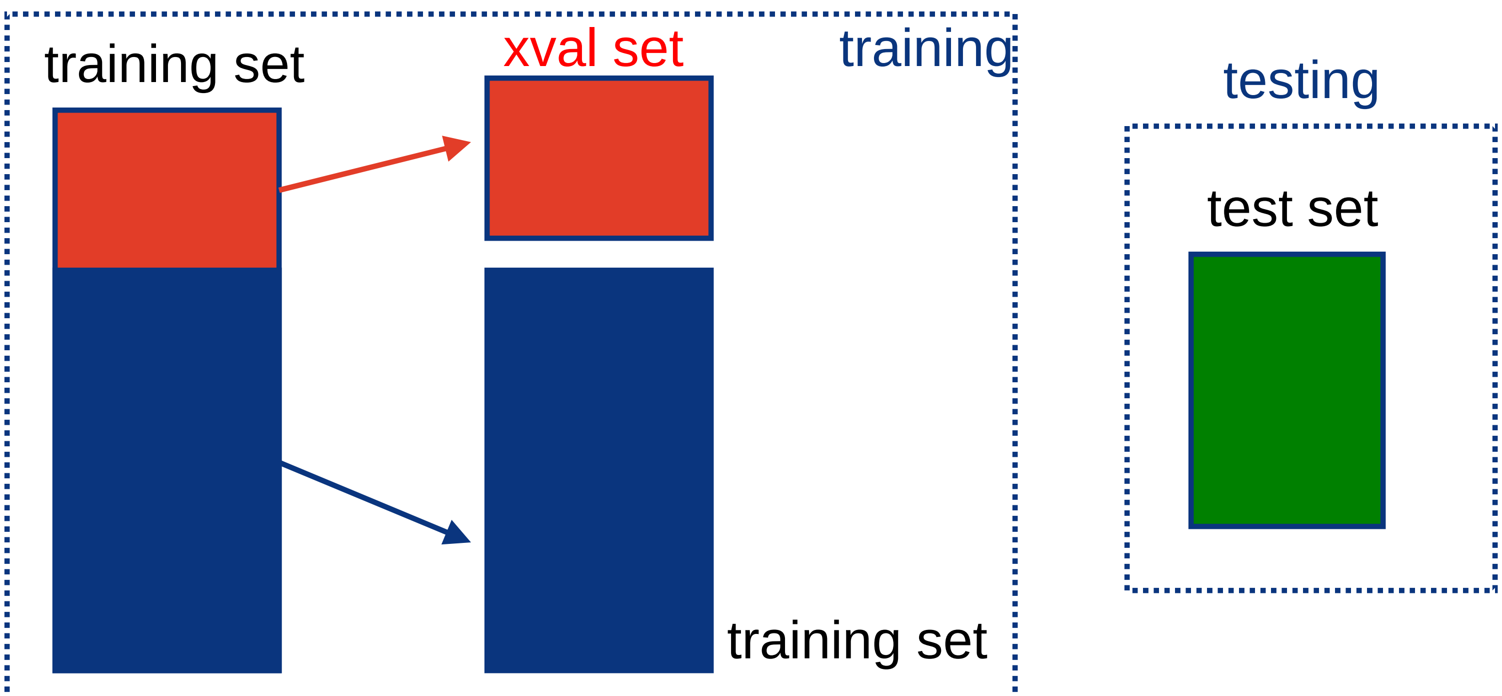
$$h(n) = \frac{k}{\sqrt{n}} \text{ or } h(n) = \frac{k}{\log n}$$

Optimal bandwidth

- in practice this has limitations
  - does not say anything about the finite data case (the one we care about)
  - still have to find the best k
- usually we end up using trial and error or techniques like cross-validation

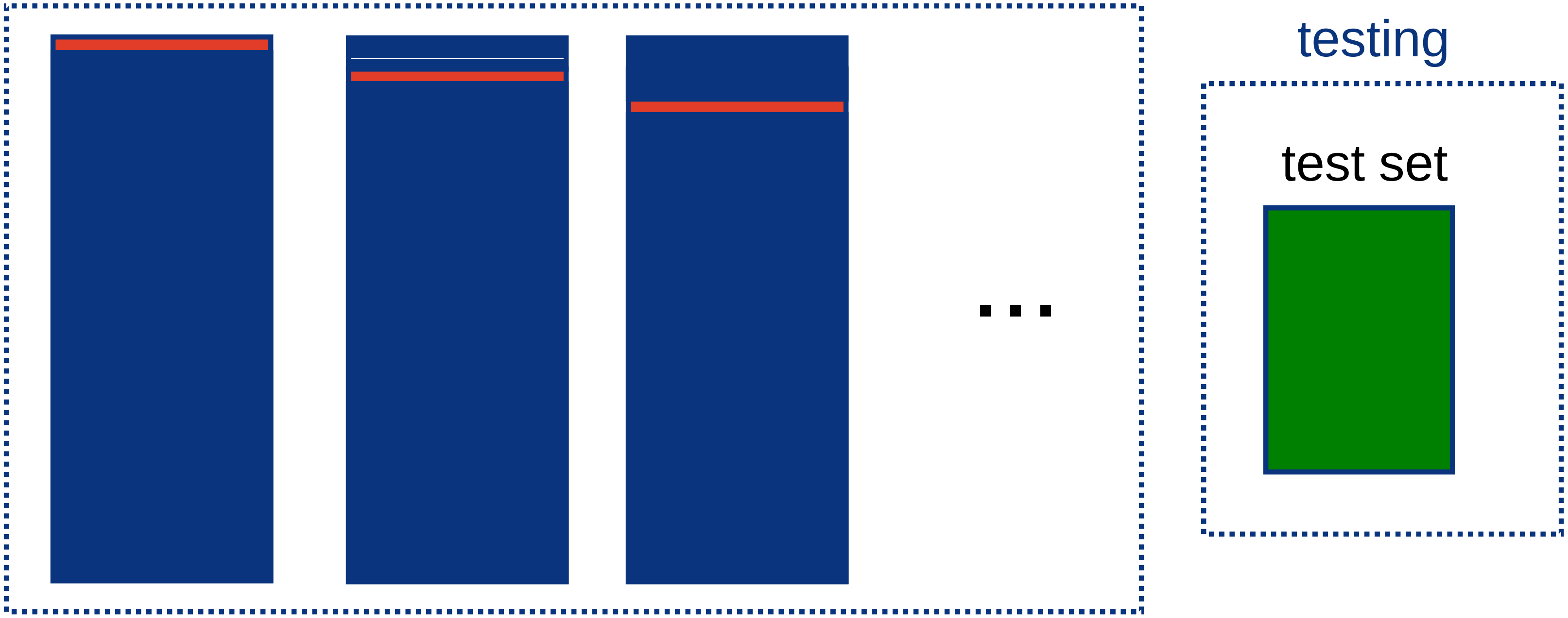
Cross-validation

- basic idea:
  - leave some data out of your training set (cross validation set)
  - train with different parameters
  - evaluate performance on cross validation set
  - pick best parameter configuration



Leave-one-out cross-validation

- many variations
- leave-one-out CV:
  - compute n estimators of $P_X(x)$ by leaving one X_i out at a time
  - for each $P_X(x)$ evaluate $P_X(X_i)$ on the point that was left out
  - pick $P_X(x)$ that maximizes this likelihood

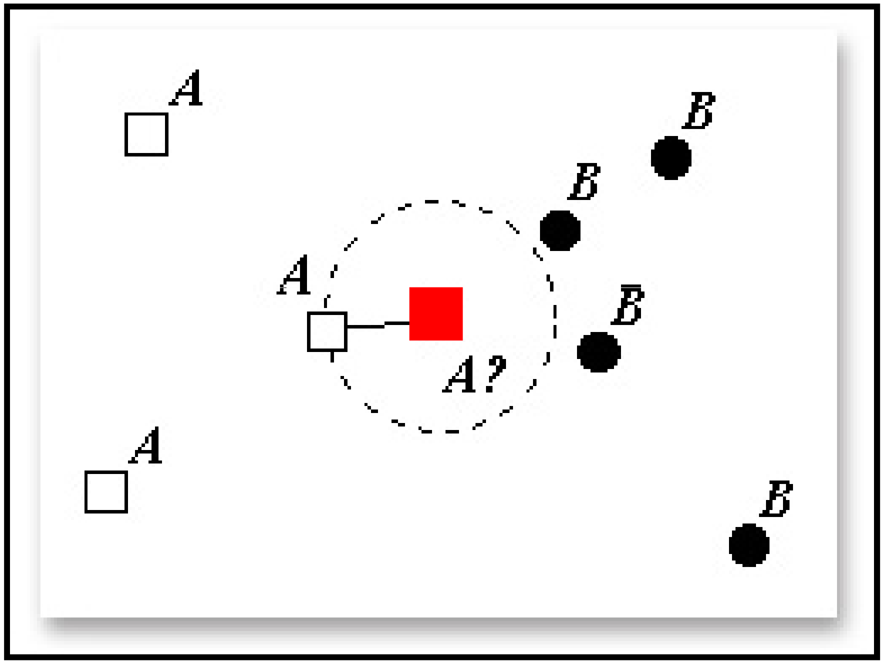


Non-parametric classifiers

- given kernel density estimates for all classes we can compute the BDR
- since the estimators are non-parametric the resulting classifier will also be non-parametric
- this term is general and applies to any learning algorithm
- a very simple example is the nearest neighbor classifier

Nearest neighbor classifier

- is the simplest possible classifier that one could think of:
  - it literally consists of assigning to the vector to classify the label of the closest vector in the training set
  - to classify the red point:
    - measure the distance to all other points
    - if the closest point is a square, assign to "square" class
    - otherwise assign to "circle" class



- it works a lot better than what one might predict

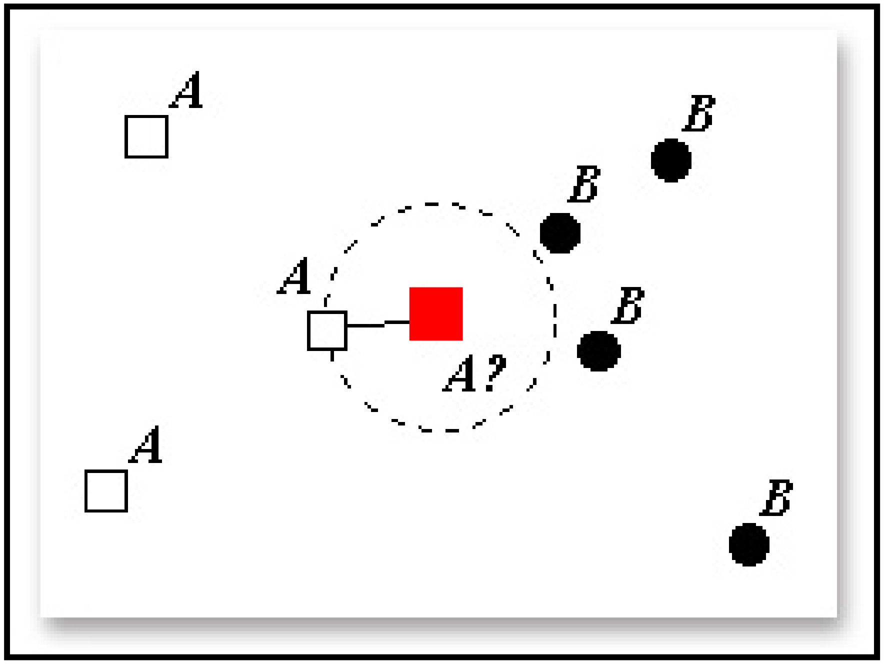
Nearest neighbor classifier

- to define it mathematically we need to define
  - a training set $\mathcal{D} = \{(x_1,y_1), \ldots, (x_n,y_n)\}$
  - x_i is a vector of observations, y_i is the label
  - a vector x to classify
- the "decision rule" is

$$\text{set} \quad y = y_{i^*}$$

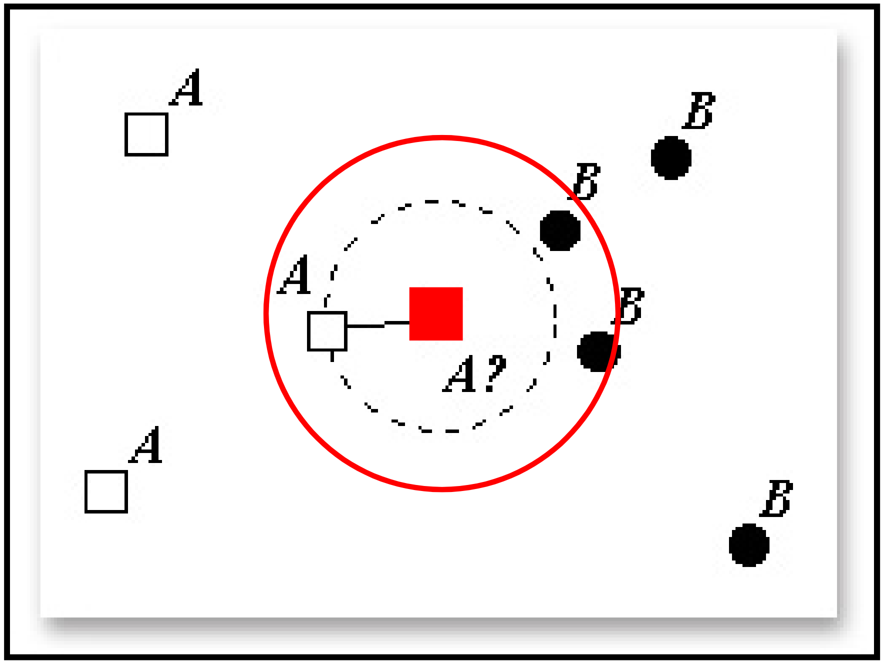
where

$$i^* = \arg\min_{i \in \{1,\ldots,n\}} d(x, x_i)$$



k-nearest neighbors

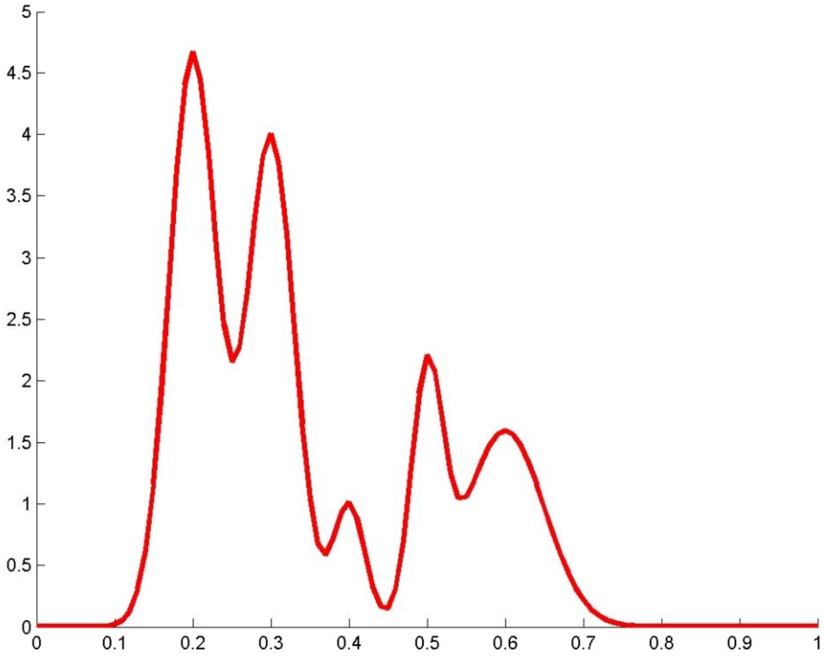
- instead of the NN, assigns to the majority vote of the k nearest neighbors
- in this example
  - NN rule says “A”
  - but 3-NN rule says “B”



- for x away from the border does not make much difference
- usually best performance for k > 1, but there is no universal number
- k large: performance degrades (no longer neighbors)
- k should be odd, to prevent ties

Mixture density estimates

- back to BDR-based classifiers
- consider the bridge traffic analysis problem
- summary:
  - want to classify vehicles into commercial/private
  - measure vehicle weight
  - estimate pdf
  - use BDR

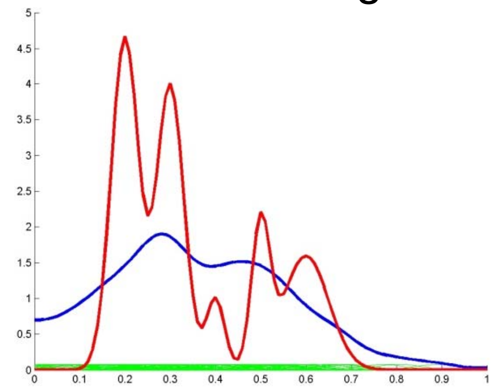


- clearly this is not Gaussian
- possible solution: use a kernel-based model

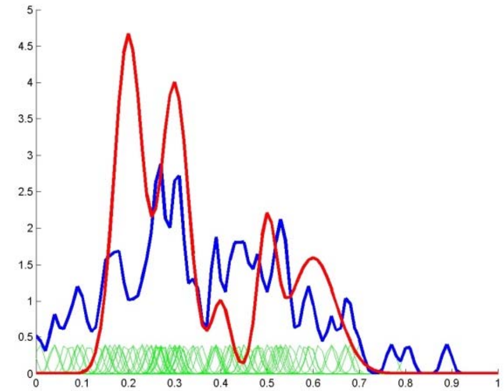
Kernel-based estimate

- simple learning procedure
  - measure car weights x_i
  - place a Gaussian on top of each measurement
- can be overkill
  - spending all degrees of freedom (# of training points) just to get the Gaussian means
  - cannot use the data to determine variances
- handpicking of bandwidth can lead to too much bias or variance

bandwidth too large: bias

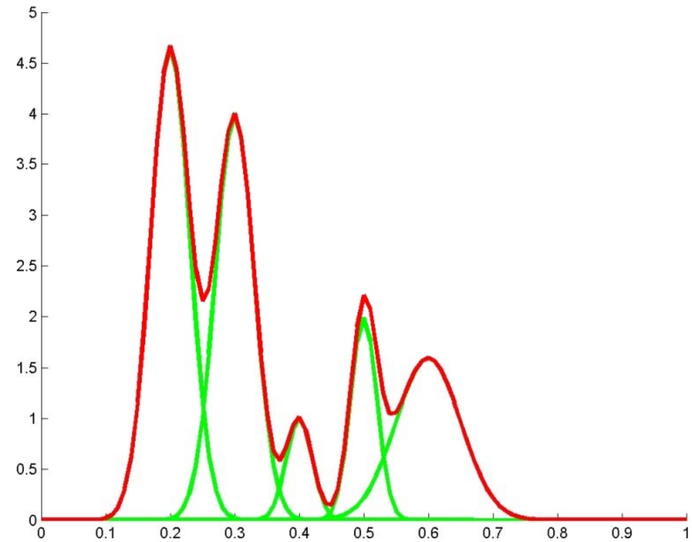


bandwidth too small: variance



mixture density estimate

- it looks like we could do better by just picking the right # of Gaussians
- this is indeed a good model:
  - density is multimodal because there is a hidden variable Z
  - Z determines the type of car



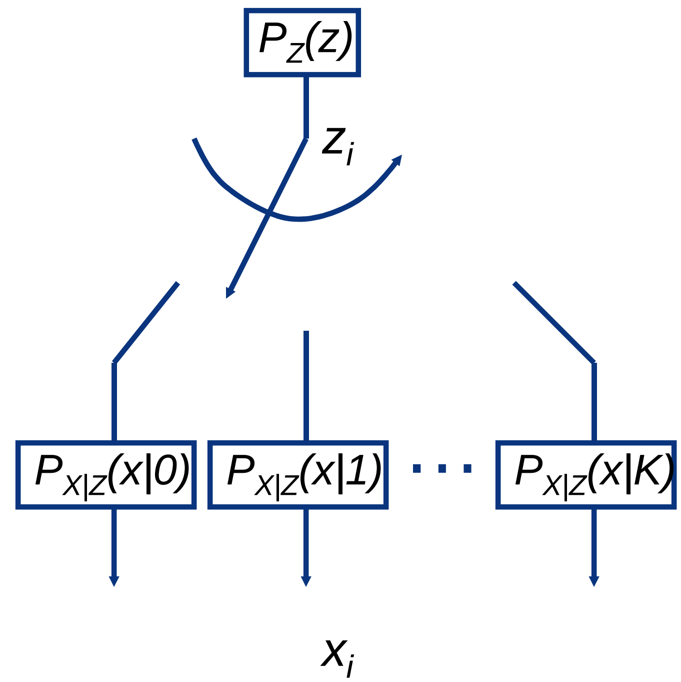
$z \in$ *{compact, sedan, station wagon, pick up, van}*

  - for a given car type the weight is approximately Gaussian (or has some other parametric form)
  - the density is a "mixture of Gaussians"

mixture model

- two types of random variables
  - Z – hidden state variable
  - X – observed variable
- observations sampled with a two-step procedure
  - a state (class) is sampled from the distribution of the hidden variable

$$P_Z(z) \rightarrow z_i$$



  - an observation is drawn from the class conditional density for the selected state

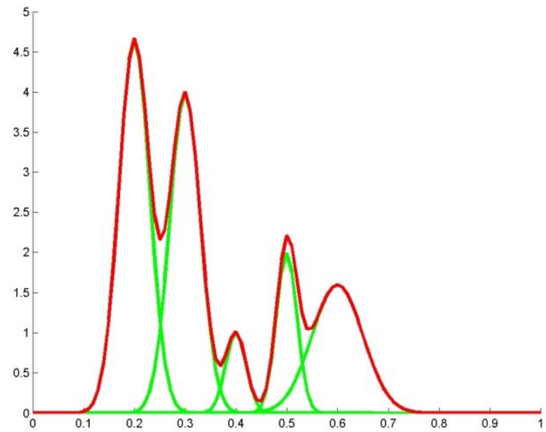
$$P_{X|Z}(x|z_i) \rightarrow x_i$$

mixture model

- the sample consists of pairs (x_i,z_i)

$$D = \{(x_1,z_1), \ldots, (x_n,z_n)\}$$

  but we never get to see the z_i



- e.g. bridge example:
  - sensor only registers weight
  - the car class was certainly there, but it is lost by the sensor
  - for this reason Z is called hidden
- the pdf of the observed data is

$$P_{\mathbf{X}}(\mathbf{x}) = \sum_{c=1}^{C} P_{\mathbf{X}|Z}(\mathbf{x}|c) P_Z(c)$$

$$= \sum_{c=1}^{C} P_{\mathbf{X}|Z}(\mathbf{x}|c) \pi_c$$

# of mixture components

component "weight"

c^{th} "mixture component"

mixtures vs kernels

- the mixture model can be rewritten as

$$P_{\mathbf{X}}(\mathbf{x}) \;=\; \sum_{c=1}^{C} \phi_c(\mathbf{x})\pi_c$$

  where $\phi_c(\mathbf{x}) > 0, \forall \mathbf{x}$ and $\int \phi_c(\mathbf{x})d\mathbf{x} = 1$.
- this looks a lot like the kernel density estimate

$$P_{\mathbf{X}}(\mathbf{x}) \;=\; \frac{1}{nh^d} \sum_{i=1}^{n} \phi\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right)$$

- the kernel density estimate is a mixture estimate of n components

  - mixture components are $\frac{1}{h^d}\phi\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right)$

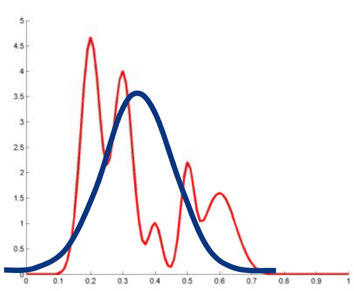
  - mixture weights are uniform $\pi_c = 1/n$.

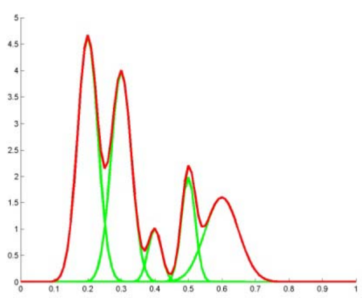
mixtures vs parametric models

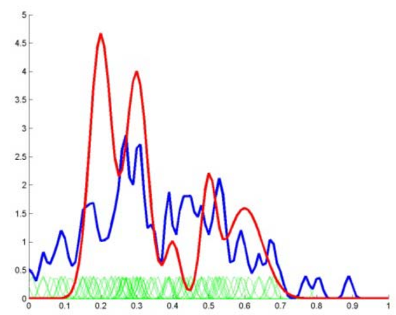
- any parametric model is a mixture of 1 component
  - the weight is 1
  - the mixture component is the parametric density itself
- mixtures provide a connection between these two extreme models

parametric | mixture of *C* components | kernel-based

C=1 | | *C=n*

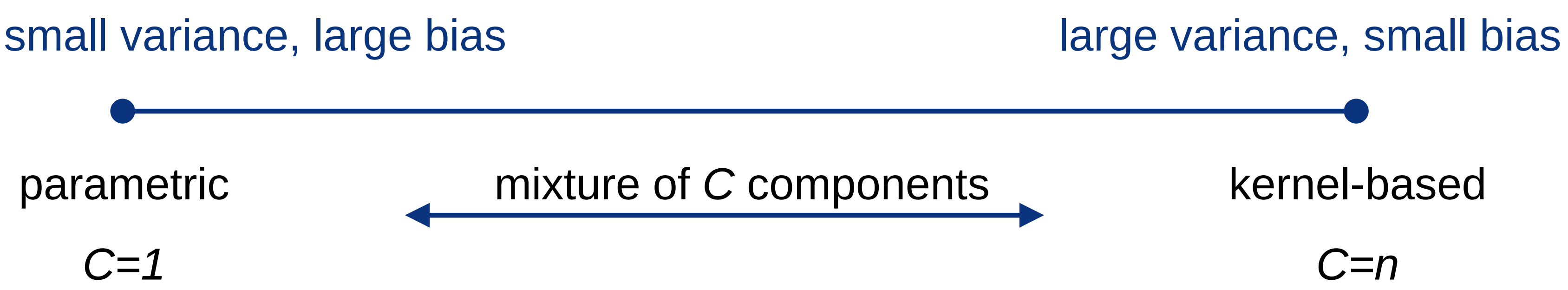






mixture advantages

- with respect to parametric estimates
  - more degrees of freedom (parameters) $\Rightarrow$ less bias
- with respect to kernel estimates
  - much smaller # of components $\Rightarrow$ less parameters, less variance



- for the mixture we can learn both means and covariances (or whatever parameters) from the data
- this usually leads to a better fit!

mixture disadvantages

- main disadvantage is learning complexity
- non-parametric estimates
  - simple: store the samples (NN); place a kernel on top of each point (kernel-based)
- parametric estimates
  - small amount of work: if ML equations have closed-form
  - substantial amount of work: otherwise (numerical solution)
- mixtures:
  - there is usually no closed-form solution
  - always need to resort to numerical procedures
- standard tool is the expectation-maximization (EM) algorithm

The basics of EM

- as usual, we start from an iid sample $D = \{x_1, \ldots, x_n\}$
- two types of random variables
  - X observed random variable
  - Z hidden random variable
- joint density of X and Z is parameterized by Ψ

$$P_{XZ}(x, z; \Psi)$$

- goal is to find parameters Ψ^* that maximize likelihood with respect to D

$$\begin{aligned} \Psi^\star &= \arg\max_\Psi P_{\mathbf{X}}(\mathcal{D}; \Psi) \\ &= \arg\max_\Psi \int P_{\mathbf{X}|Z}(\mathcal{D}|z; \Psi) P_Z(z; \Psi) dz \end{aligned}$$

Complete vs incomplete data

- the set

$$D_c = \{(x_1,z_1), \ldots, (x_n,z_n)\}$$

  is called the complete data

- the set

$$D = \{x_1, \ldots, x_n\}$$

  is called the incomplete data

- in general, the problem would be trivial if we had access to the complete data
- to see this let's consider a specific example
  - Gaussian mixture of C components
  - parameters $\Psi = \{(\pi_1,\mu_1,\Sigma_1), \ldots,(\pi_C,\mu_C,\Sigma_C)\}$

Learning with complete data

- given the complete data D_c, we only have to split the training set according to the labels z_i

$$D^1 = \{x_i|z_i=1\}, \quad D^2 = \{x_i|z_i=2\}, \quad \ldots \quad , D^C = \{x_i|z_i=C\}$$

- the likelihood of the complete data is

$$\begin{aligned}
P_{\mathbf{X},Z}(\mathcal{D},\mathbf{z};\Psi) &= \prod_{c=1}^{C} P_{\mathbf{X},Z}(\mathcal{D}^c,c;\Psi) \\
&= \prod_{c=1}^{C} P_{\mathbf{X}|Z}(\mathcal{D}^c|c;\Psi)P_Z(c;\Psi) \\
&= \prod_{c=1}^{C} \mathcal{G}(\mathcal{D}^c,\mu_c,\Sigma_c)\pi_c
\end{aligned}$$

Learning with complete data

- the optimal parameters are

$$\Psi^\star = \arg\max_{\Psi} \prod_{c=1}^{C} \mathcal{G}(\mathcal{D}^c, \mu_c, \Sigma_c)\pi_c$$

- since each term only depends on D^c and (π_c, μ_c, Σ_c) this can be simplified into

$$(\pi_c^\star, \mu_c^\star, \Sigma_c^\star) = \arg\max_{\pi,\mu,\Sigma} \mathcal{G}(\mathcal{D}^c, \mu, \Sigma)\pi$$

- and we have a collection of C very familiar maximum likelihood problems (HW 2)
  - ML estimate of the Gaussian parameters
  - ML estimate of the class probabilities

Learning with complete data

- the solution is

$$\pi_c^\star = \frac{|\{\mathbf{x}_i \in \mathcal{D}^c\}|}{n}$$

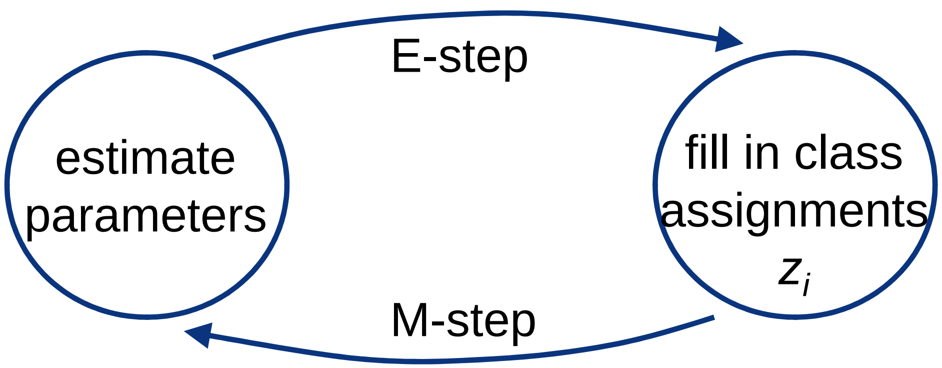
$$\mu_c^\star = \frac{1}{|\{\mathbf{x}_i \in \mathcal{D}^c\}|} \sum_{i|\mathbf{x}_i \in \mathcal{D}^c} \mathbf{x}_i$$

$$\Sigma_c^\star = \frac{1}{|\{\mathbf{x}_i \in \mathcal{D}^c\}|} \sum_{i|\mathbf{x}_i \in \mathcal{D}^c} (\mathbf{x}_i - \mu_c^\star)(\mathbf{x}_i - \mu_c^\star)^T$$

- hence, all the hard work seems to be in figuring out what the z_i are
- the EM algorithm does this iteratively

Learning with incomplete data (EM)

- the basic idea is quite simple
  1. start with an initial parameter estimate $\Psi^{(0)}$
  2. **E-step:** given current parameters $\Psi^{(i)}$ and observations in D, "guess" what the values of the z_i are
  3. **M-step:** with the new z_i, we have a complete data problem, solve this problem for the parameters, i.e. compute $\Psi^{(i+1)}$
  4. go to 2.
- this can be summarized as



Any Questions?