Lecture 13

Classification and the EM algorithm

Last Time

- Logistic Regression, MLP, and Backprop
- Universal Approximation
- Learning Representations

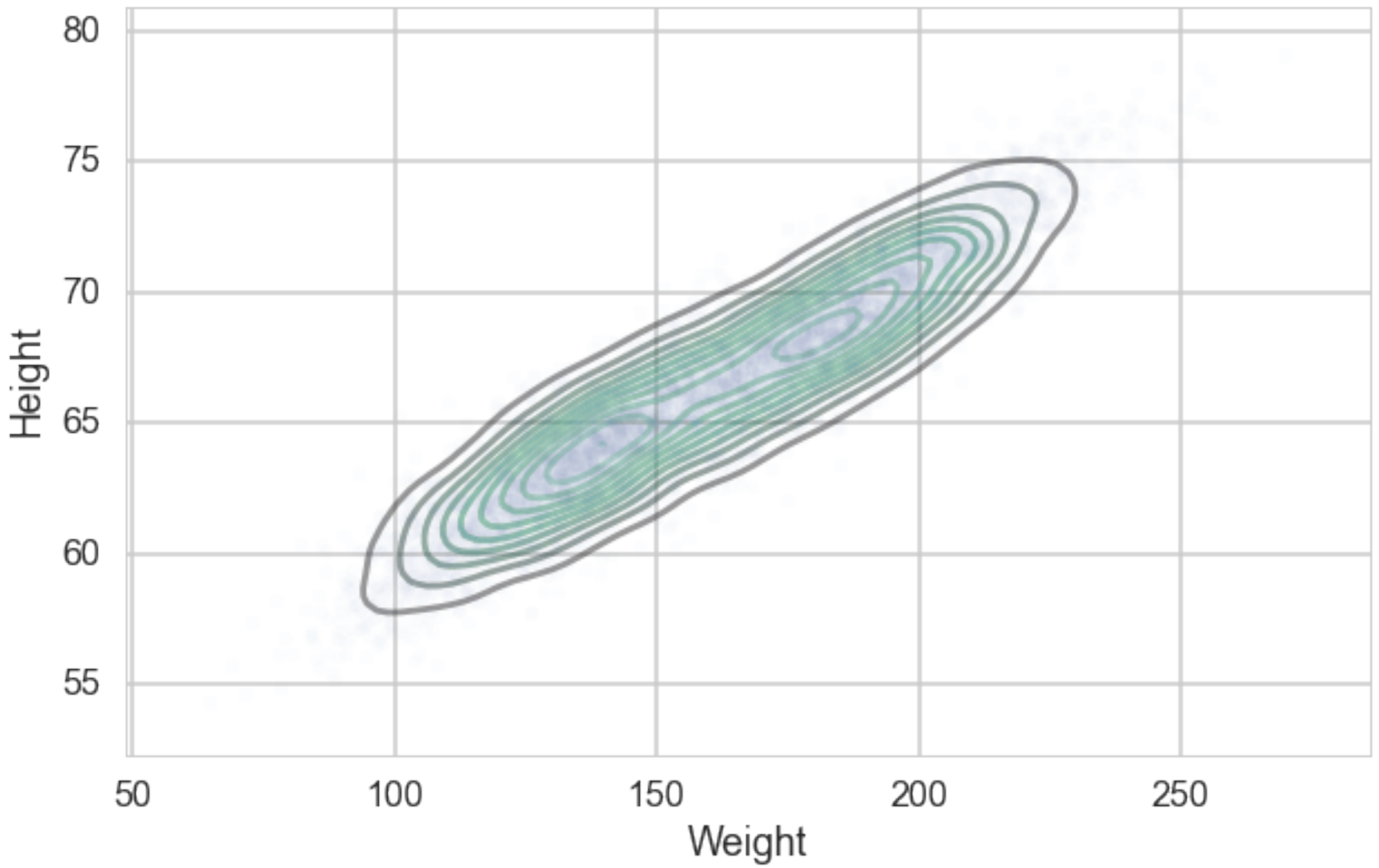
Today

- Latent Variables
- Mixture Models
- Supervised vs Unsupervised vs Semi-Supervised Learning
- Missing Data and the EM algorithm
- EM algorithm and the mixture model

PROBABILISTIC CLASSIFICATION

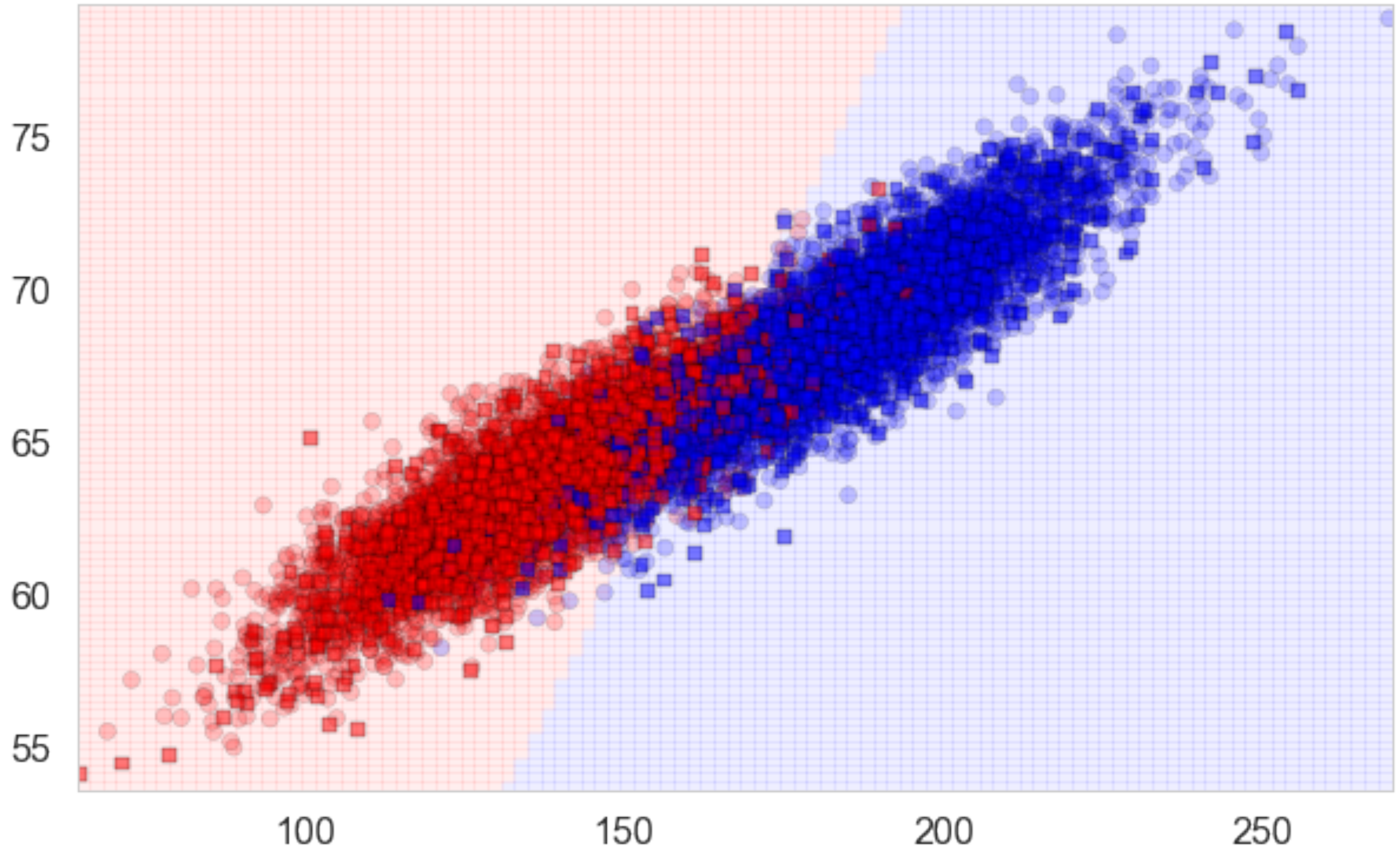
Model $P(y|x)$ $[P(c|x))]$

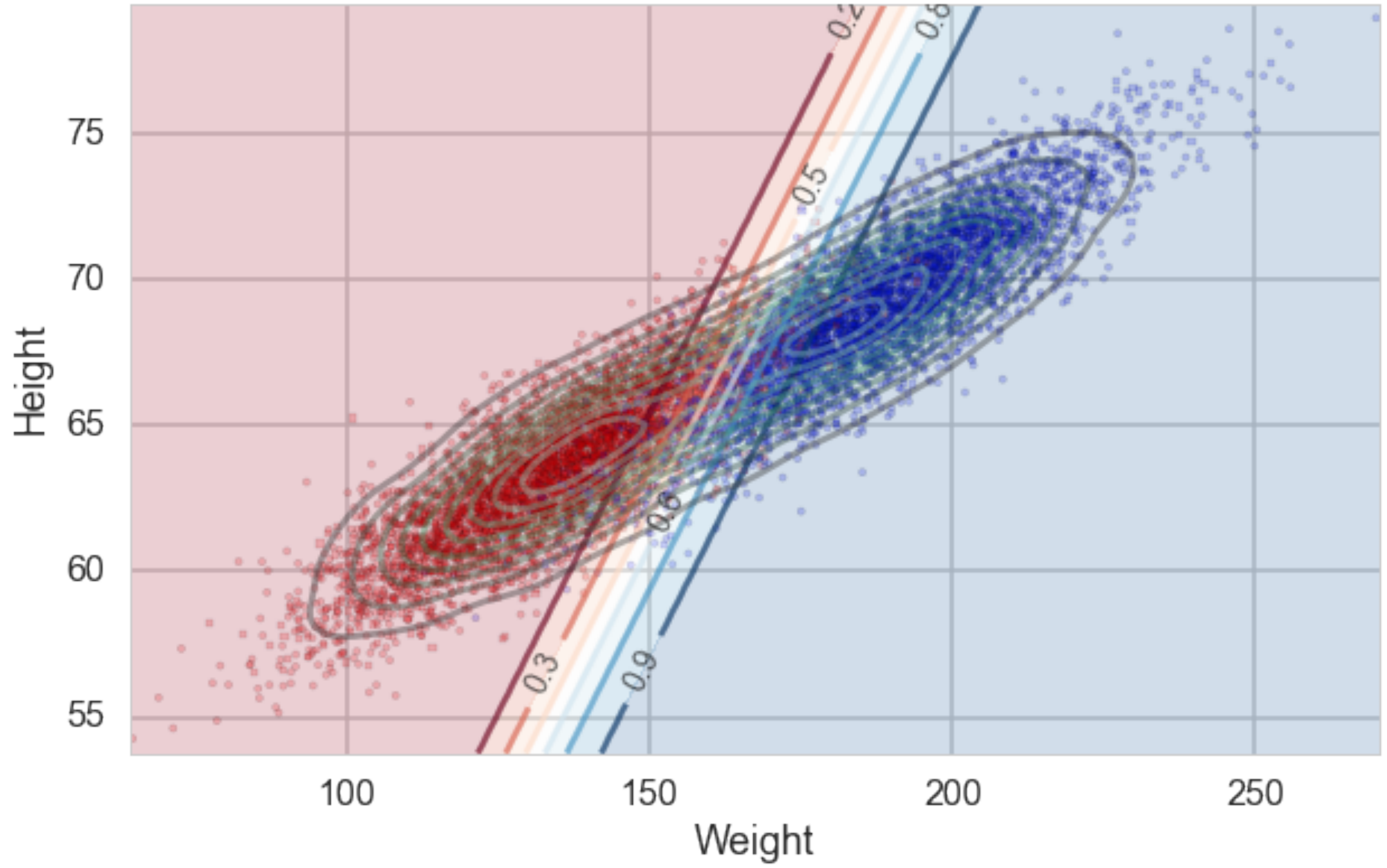
or $P(x|y)$ $[P(x|c)]$.



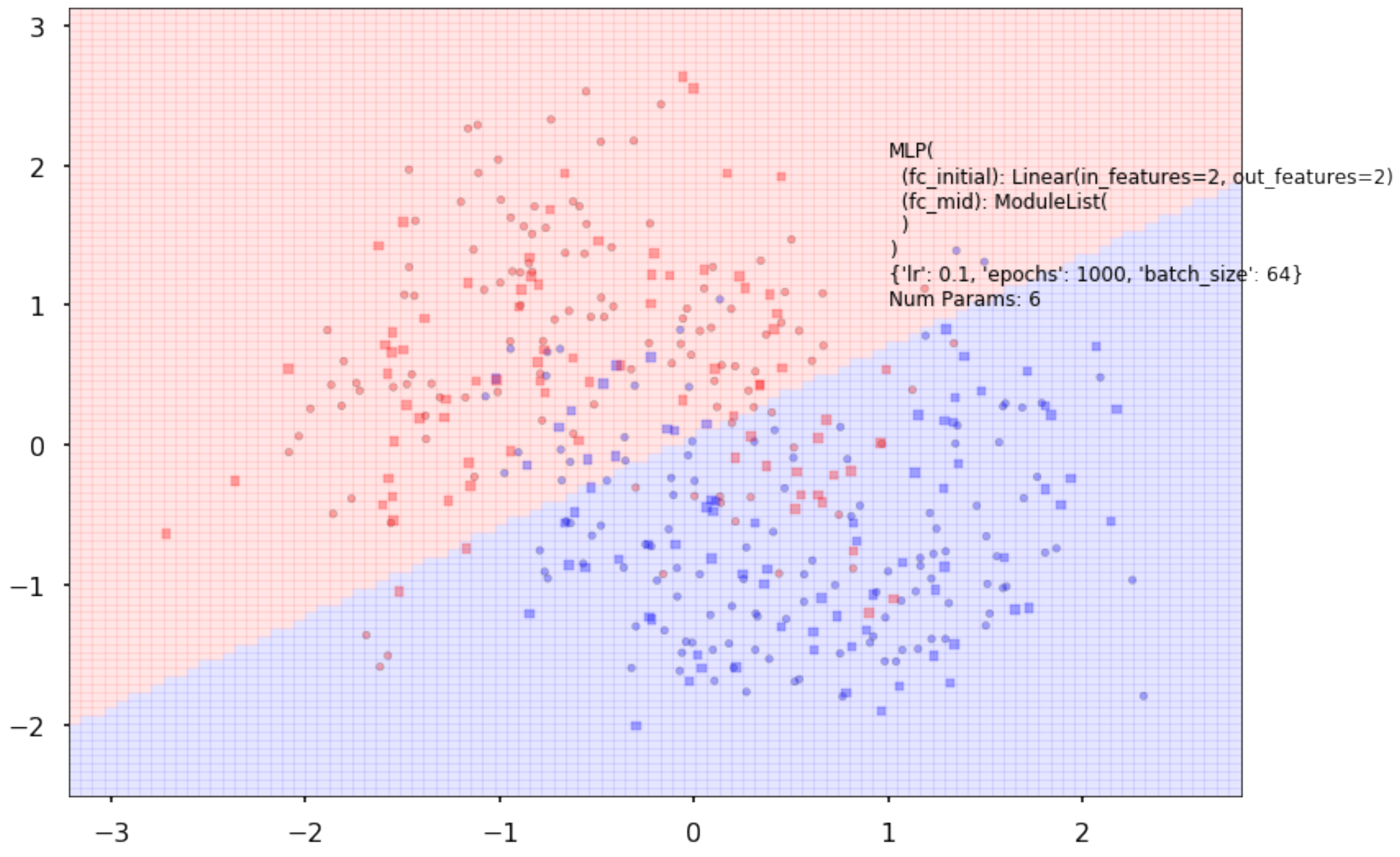
DISCRIMINATIVE CLASSIFIER

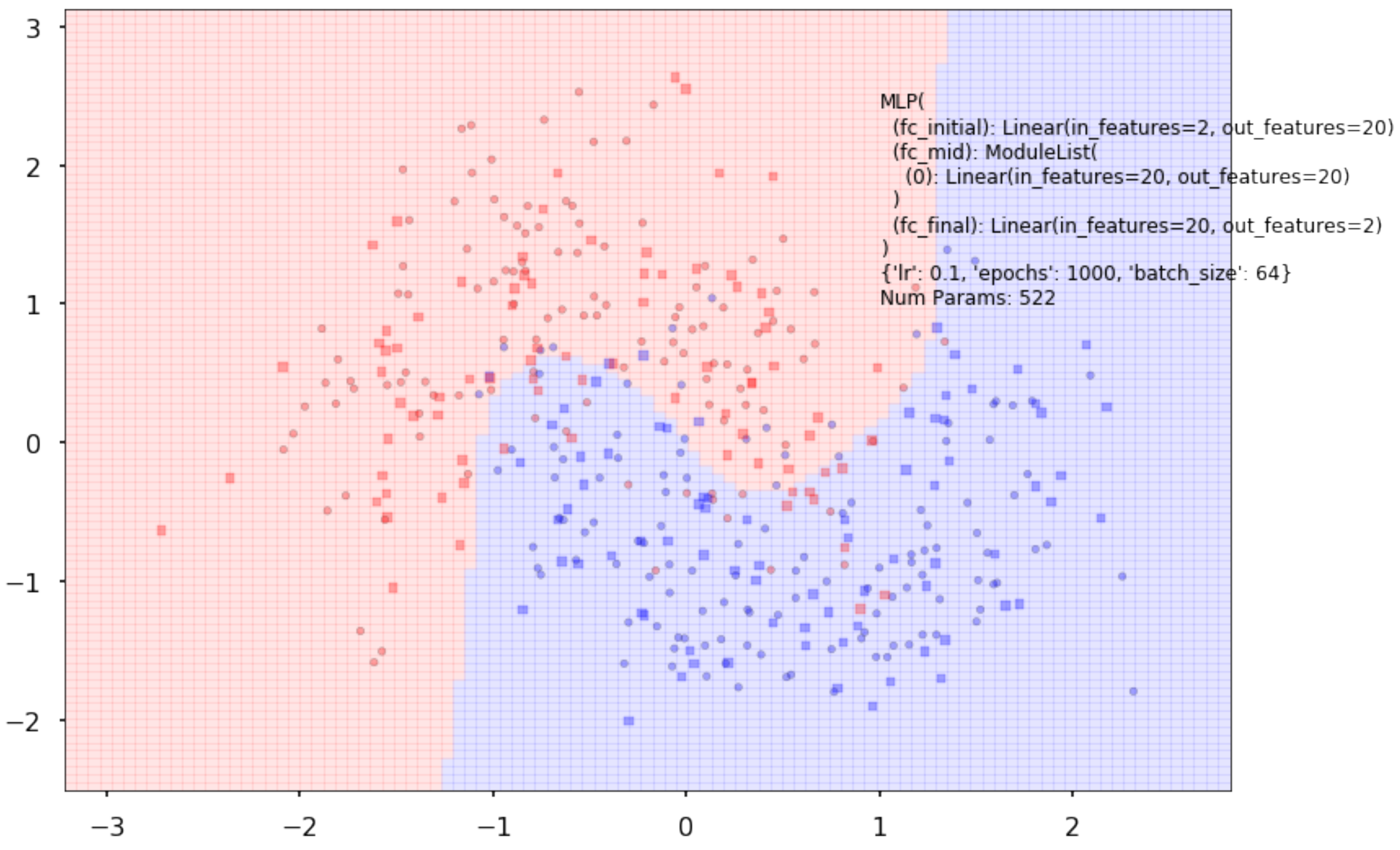
$$P(y|x) or P(c|x) : P(male|height, weight)$$





Half moon dataset (artificially GENERATED)



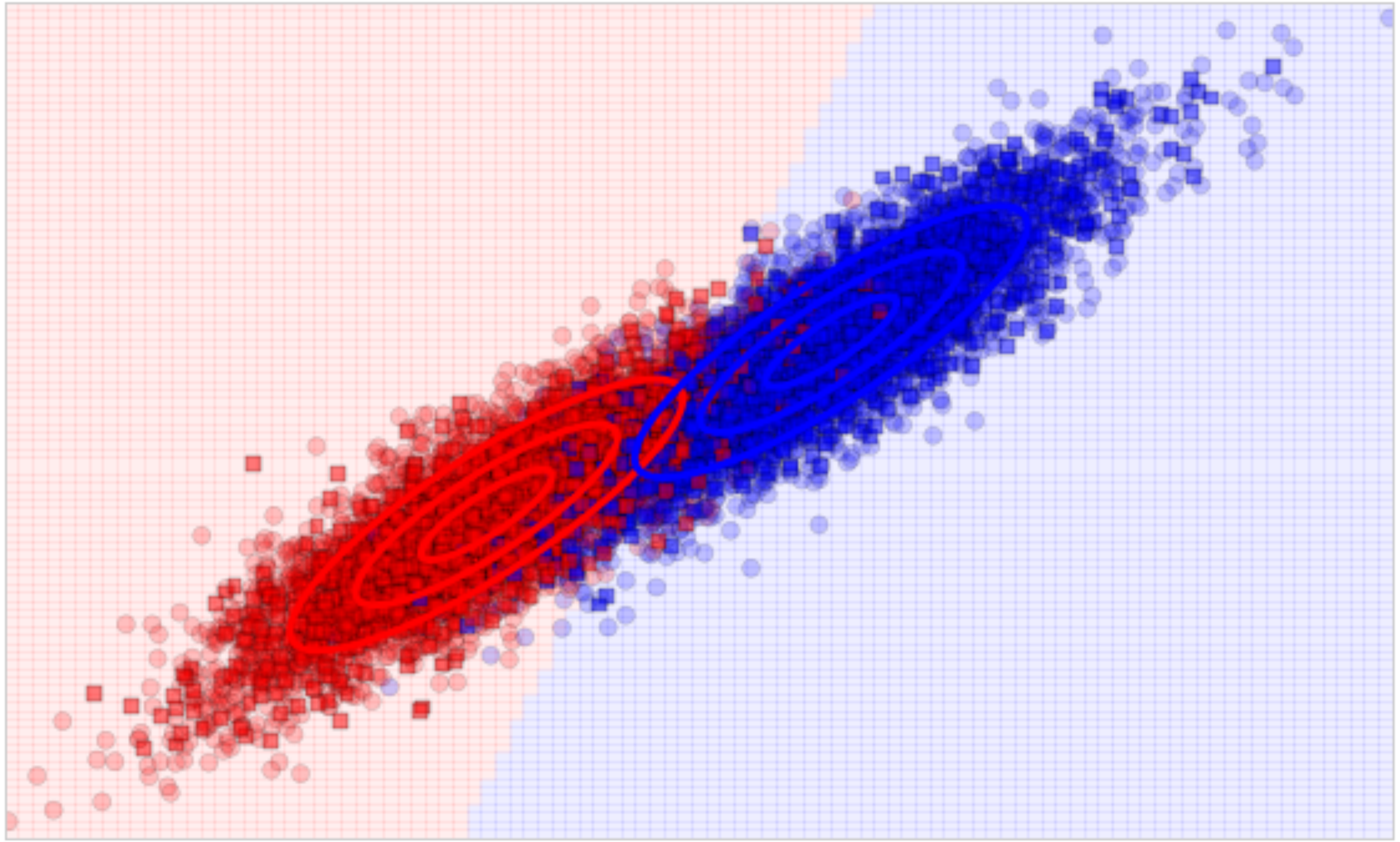


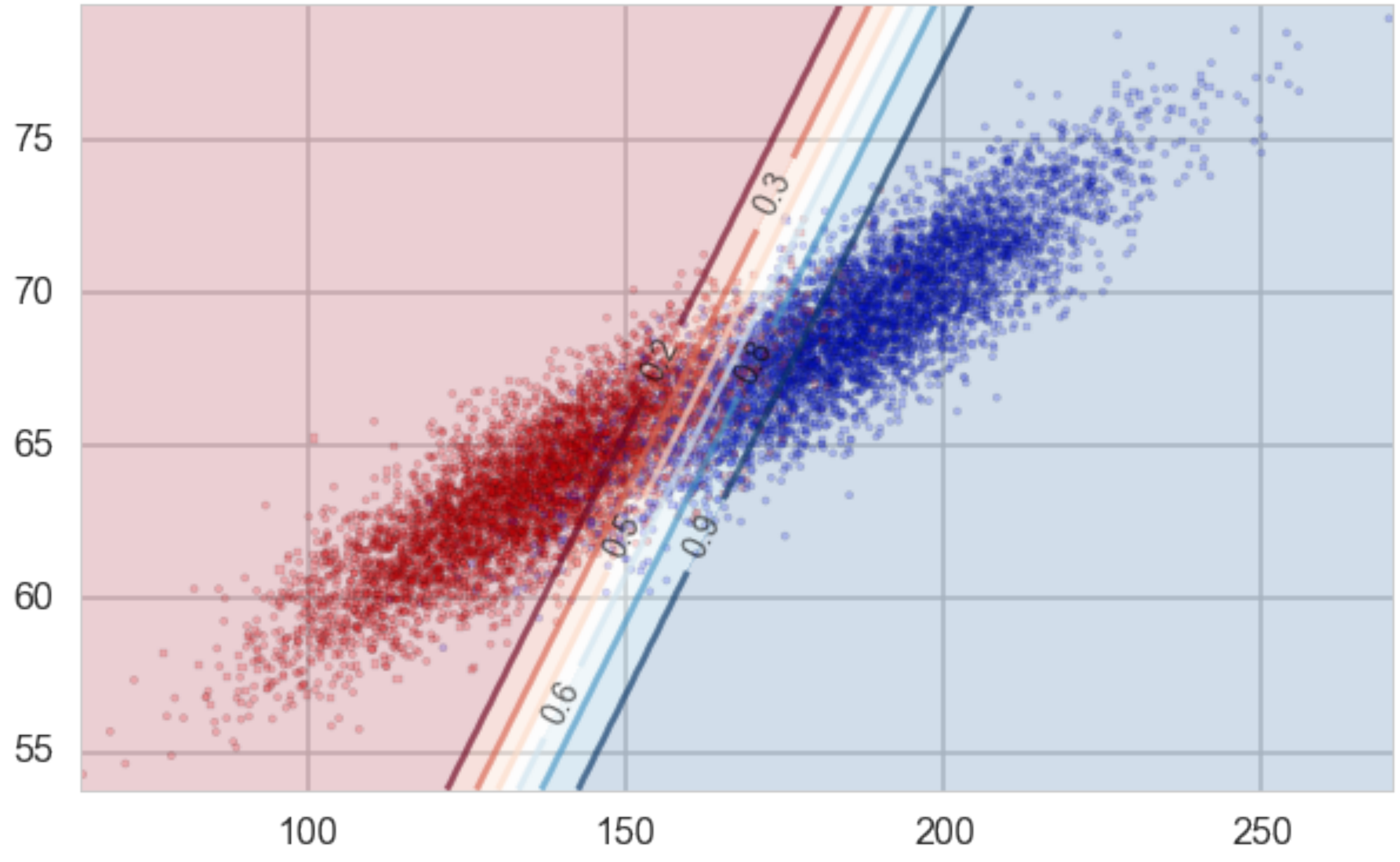
Discriminative Learning

- are these classifiers any good?
- they are discriminative and draw boundaries, but thats it
- they are cheaper to calculate but shed no insight
- would it not be better to have a classifier that captured the generative process

GENERATIVE CLASSIFIER

$$P(y|x) \propto P(x|y)P(x) : P(height, weight|male) \times P(male)$$





Representation Learning

- the idea of generative learning is to capture an underlying representation (compressed) of the data
- in the previous slide it was 2 normal distributions
- generally more complex, but the idea if to fit a "generative" model whose parameters represent the process
- wait, we've been doing this in our bayesian or conditional-on-data-marginalize-over-all-else paradigm
- besides gpus and autodiff on backprop, this is the third pillar of the AI rennaissance: the choice of better representations: e.g. convolutions

Ok, so how do we model (simple) representations. We've been doing it already....

Latent Variables

that we marginalize over!

- instead of bayesian vs frequentist, think hidden vs not hidden
- key concept: full data likelihood $p(\mathbf{x}, \mathbf{z})$ vs partial data likelihood $p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x}|\mathbf{z}) p(\mathbf{z})$
- with $\mathbf{z} = \theta$ we have the standard Bayesian scenario, but consider $\mathbf{z}$ now to be some hidden representation
- For regression/classification "$\mathbf{z} = y$ or c", full data likelihood is supervised learning with partial being unsupervised learning
- observed variables $\mathbf{x}$ correspond to data, and latent variables $\mathbf{z}$ to classes/parameters

From `edwardlib` docs: $p(\mathbf{x} \mid \mathbf{z})$

describes how any data $\mathbf{x}$ depend on the latent variables $\mathbf{z}$.

- **The likelihood posits a data generating process**, where the data $\mathbf{x}$ are assumed drawn from the likelihood conditioned on a particular hidden pattern described by $\mathbf{z}$.
- The *prior* $p(\mathbf{z})$ is a probability distribution that describes the latent variables present in the data. **The prior posits a generating process of the hidden structure**.

Mixture Models motivation

- $\mathbf{z}$ as "classes" in a classification problem leads to a generative classifier
- but in general, that identification is very strong, indeed $\mathbf{z}$ may just be a representation

Mixture Models

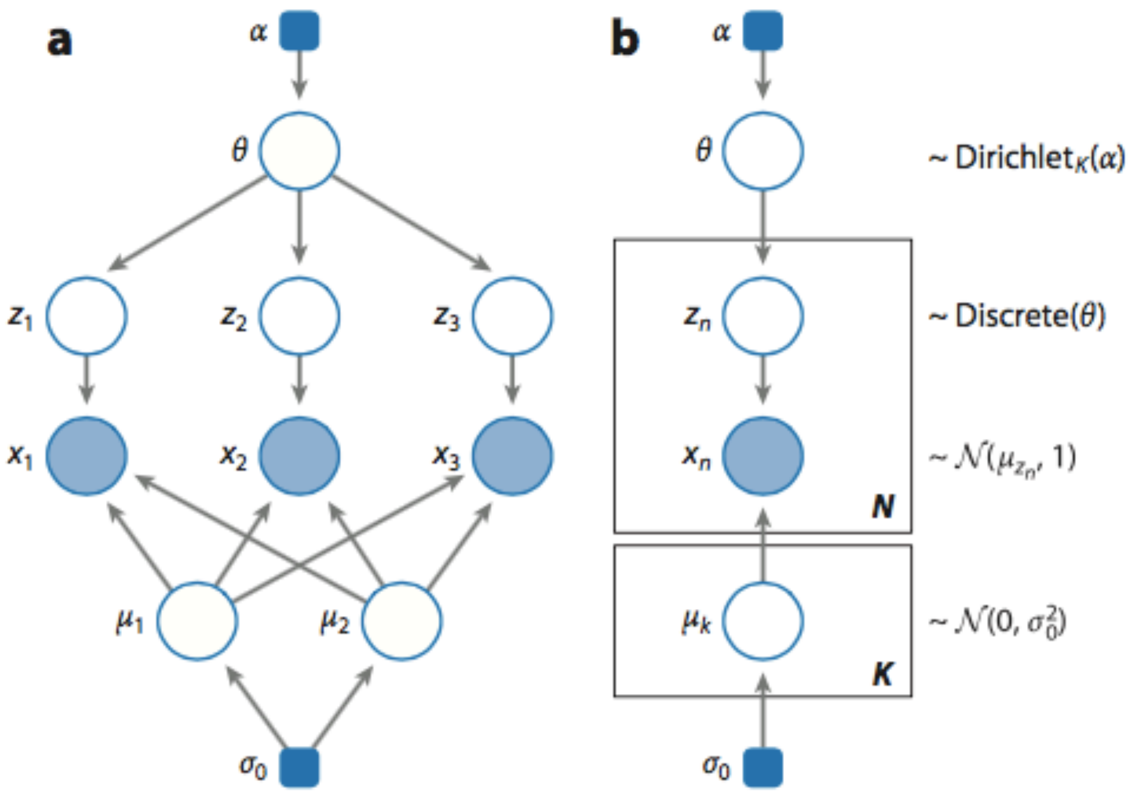


Figure 3

(*a*) A graphical model for a mixture of two Gaussians. There are three data points. The shaded nodes are observed variables, the unshaded nodes are hidden variables, and the blue square boxes are fixed hyperparameters (such as the Dirichlet parameters). (*b*) A graphical model for a mixture of *K* Gaussians with *N* data points.

A distribution $p(x|\{\theta_k\})$ is a mixture of K component distributions $p_1, p_2, \ldots p_K$ if:

$$p(x|\{\theta_k\}) = \sum_k \lambda_k p_k(x|\theta_k)$$

with the λ_k being mixing weights, $\lambda_k > 0$, $\sum_k \lambda_k = 1$.

Example: Zero Inflated Poisson

Generative Model: How to simulate from it?

$$Z \sim Categorical(\lambda_1, \lambda_2, \ldots, \lambda_K)$$

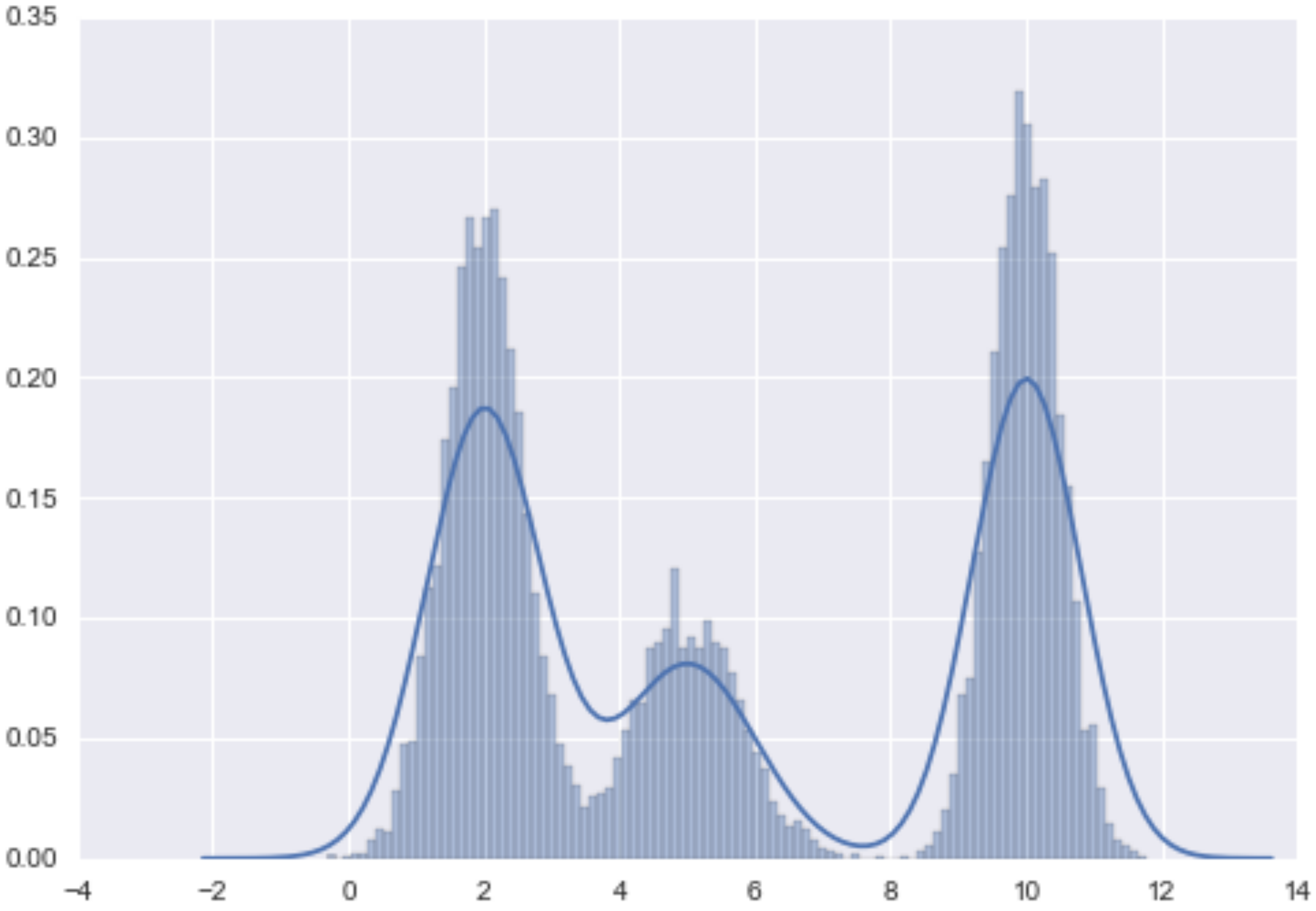
where Z says which component X is drawn from.

Thus λ_j is the probability that the hidden class variable $z = j$.

Then: $X \sim p_z(x|\theta_z)$ and general structure is:

$$p(x|\{\theta_z\}) = \sum_z p(x, z) = \sum_z p(z)p(x|z, \theta_z)\,.$$

Gaussian Mixture Model



$$p(x|\{\theta_k\}) = \sum_k \lambda_k N(x|\mu_k, \Sigma_k)$$

Generative:

```
mu_true = np.array([2, 5, 10])
sigma_true = np.array([0.6, 0.8, 0.5])
lambda_true = np.array([.4, .2, .4])
n = 10000

# Simulate from each distribution according to mixing proportion psi
z = multinomial.rvs(1, lambda_true, size=n) #categorical
x=np.array([np.random.normal(mu_true[i.astype('bool')][0],\
    sigma_true[i.astype('bool')][0]) for i in z])
```

```
multinomial.rvs(1,[0.6,0.1, 0.3], size=10)
array([[1, 0, 0],[0, 0, 1],...[1, 0, 0],[1, 0, 0]])
```

The two meanings of generative

Thus we **abuse** the world **generative** in two senses:

1. A way to generate data drom a data story. Here think of $\mathbf{z} = \theta$
2. A Model in which we try to figure $p(\mathbf{x}, \mathbf{z})$ or $p(\mathbf{x}|\mathbf{z})$. Here think of $\mathbf{z} = c$ or a class label.

Now lets focus on the latter. Suppose we believe their exists a "class" or representation $\mathbf{z}$. Then a dichotomy arises depending on whether $\mathbf{z}$ is observed or not.

Supervised vs Unsupervised Learning

In **Supervised Learning**, Latent Variables $\mathbf{z}$ are observed.

In other words, we can write the full-data likelihood $p(\mathbf{x}, \mathbf{z})$

In **Unsupervised Learning**, Latent Variables $\mathbf{z}$ are hidden.

We can only write the observed data likelihood:

$$p(\mathbf{x}) = \sum_{z} p(\mathbf{x}, \mathbf{z}) = \sum_{z} p(\mathbf{z}) p(\mathbf{x}|\mathbf{z})$$

GMM supervised formulation

$$Z \sim \text{Bernoulli}(\lambda)$$

$$X|z = 0 \sim \mathcal{N}(\mu_0, \Sigma_0),\, X|z = 1 \sim \mathcal{N}(\mu_1, \Sigma_1)$$

Full-data loglike: $l(x, z|\lambda, \mu_0, \mu_1, \Sigma) = -\sum_{i=1}^{m} \log((2\pi)^{n/2} |\Sigma|^{1/2})$

$$-\frac{1}{2}\sum_{i=1}^{m}(x - \mu_{z_i})^T \Sigma^{-1} (x - \mu_{z_i}) + \sum_{i=1}^{m} [z_i \; \log\lambda + (1 - z_i) \log(1 - \lambda)]$$

Solution to MLE

$$\lambda = \frac{1}{m}\sum_{i=1}^{m}\delta_{z_i,1}$$

$$\mu_0 = \frac{\sum_{i=1}^{m}\delta_{z_i,0}\, x_i}{\sum_{i=1}^{m}\delta_{z_i,0}}$$

$$\mu_1 = \frac{\sum_{i=1}^{m}\delta_{z_i,1}\, x_i}{\sum_{i=1}^{m}\delta_{z_i,1}}$$

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{z_i})(x_i - \mu_{z_i})^T$$

Classification

We can use the log likelihood at a given x as a classifier: assign class depending upon which probability $p(x_j|\lambda, z, \Sigma)$ is larger. (JUST x likelihood, as we want to compare probabilities at fixed zs).

$$log\, p(x_j|\lambda, z, \Sigma) = -\sum_{i=1}^{m} \log((2\pi)^{n/2} |\Sigma|^{1/2}) - \frac{1}{2} \sum_{i=1}^{m} (x - \mu_{z_i})^T\, \Sigma^{-1} (x - \mu_{z_i})$$

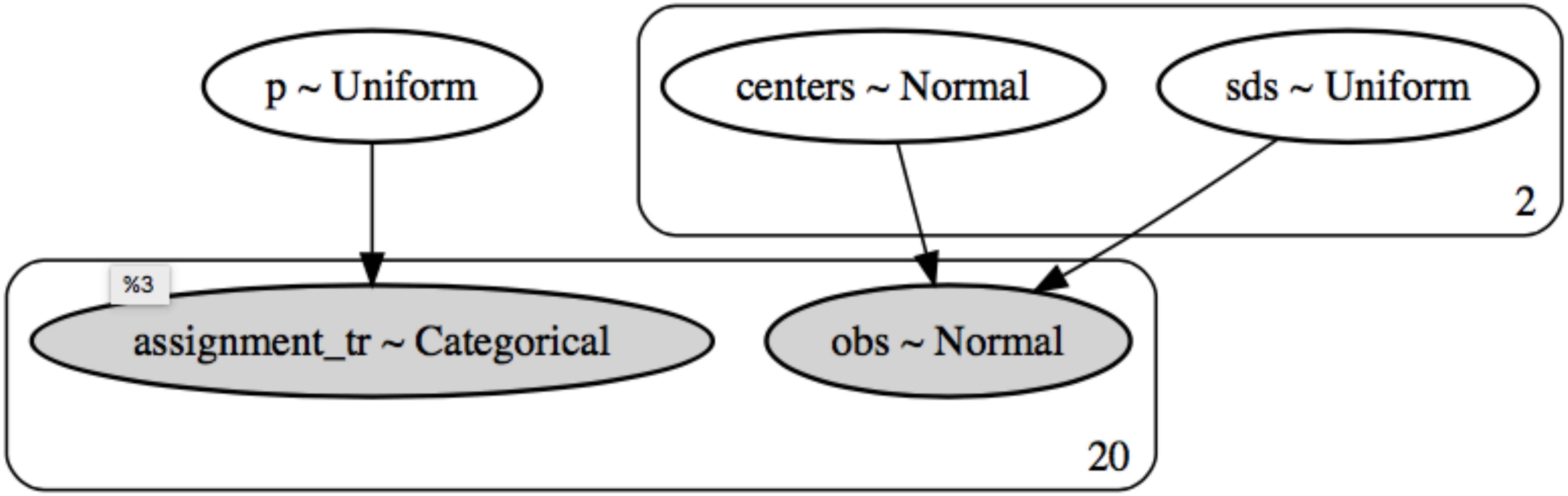
The first term of the likelihood does not matter since it is independent of z.

Bayesian Supervised

```
with pm.Model() as classmodel1:
    p1 = pm.Uniform('p', 0, 1)
    p2 = 1 - p1
    p = tt.stack([p1, p2])
    #Notice the "observed" below
    assignment_tr = pm.Categorical("assignment_tr", p,
                                   observed=ztr)
    sds = pm.Uniform("sds", 0, 100, shape=2)
    centers = pm.Normal("centers",
                        mu=np.array([130, 170]),
                        sd=np.array([20, 20]),
                        shape=2)
    p_min_potential = pm.Potential('lam_min_potential', tt.switch(tt.min(p) < .1, -np.inf, 0))
    order_centers_potential = pm.Potential('order_centers_potential',
                                           tt.switch(centers[1]-centers[0] < 0, -np.inf, 0))

    # and to combine it with the observations:
    observations = pm.Normal("obs", mu=centers[assignment_tr], sd=sds[assignment_tr], observed=xtr)
```

Supervised graph



Mixture Model as Generative Classifier

For a feature vector x, we use Bayes rule to express the posterior of the class-conditional or component-conditional ("c") as:

$$p(z = c|x, \theta) = \frac{p(z = c|\theta)p(x|z = c, \theta)}{\sum_{c'} p(z = c'|\theta)p(x|z = c', \theta)}$$

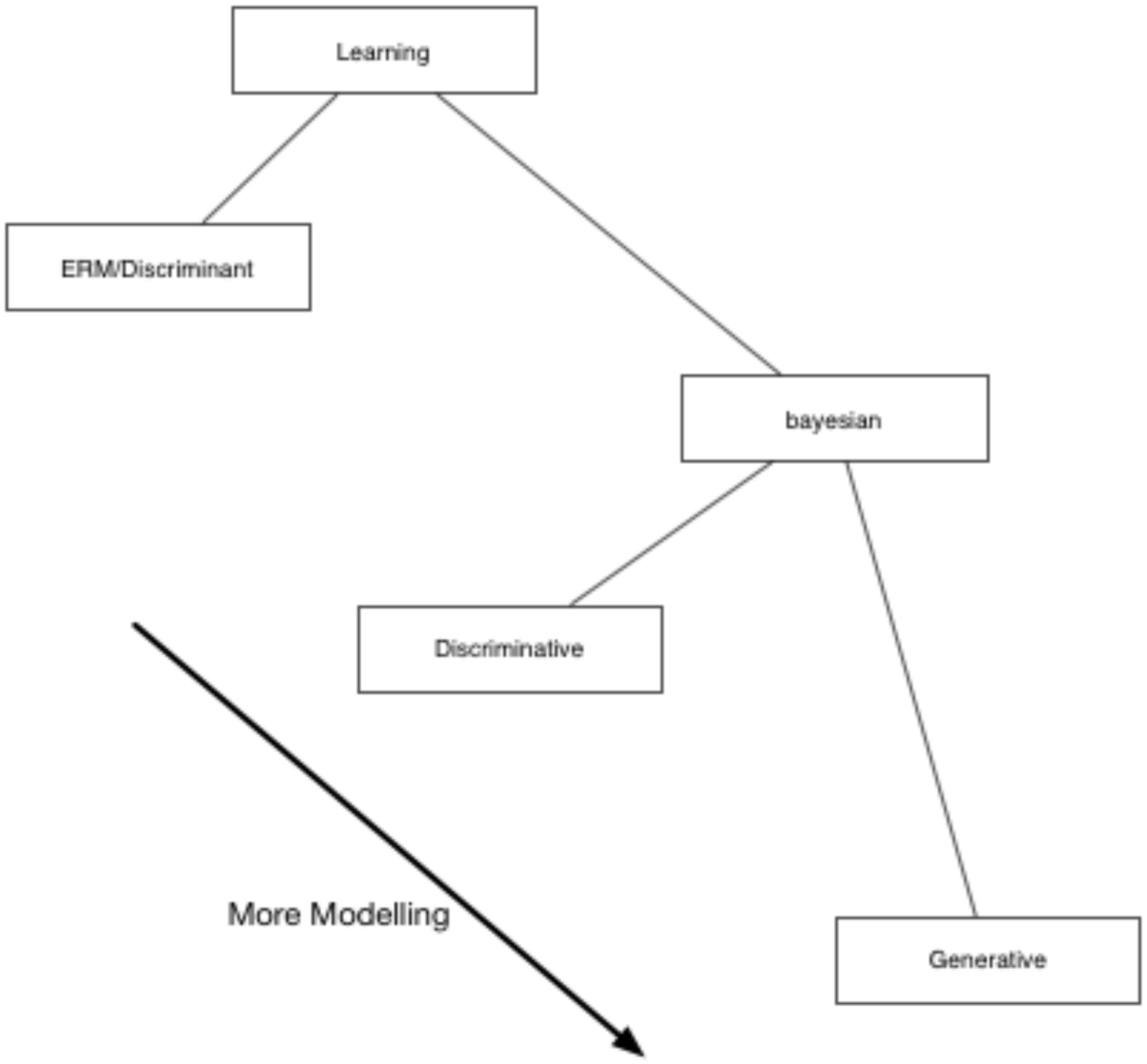
This is a **generative classifier**, since it specifies how to generate the data using the class-conditional density $p(x|z = c, \theta)$ and the class prior $p(z = c|\theta)$.

Generative vs Discriminative classifiers

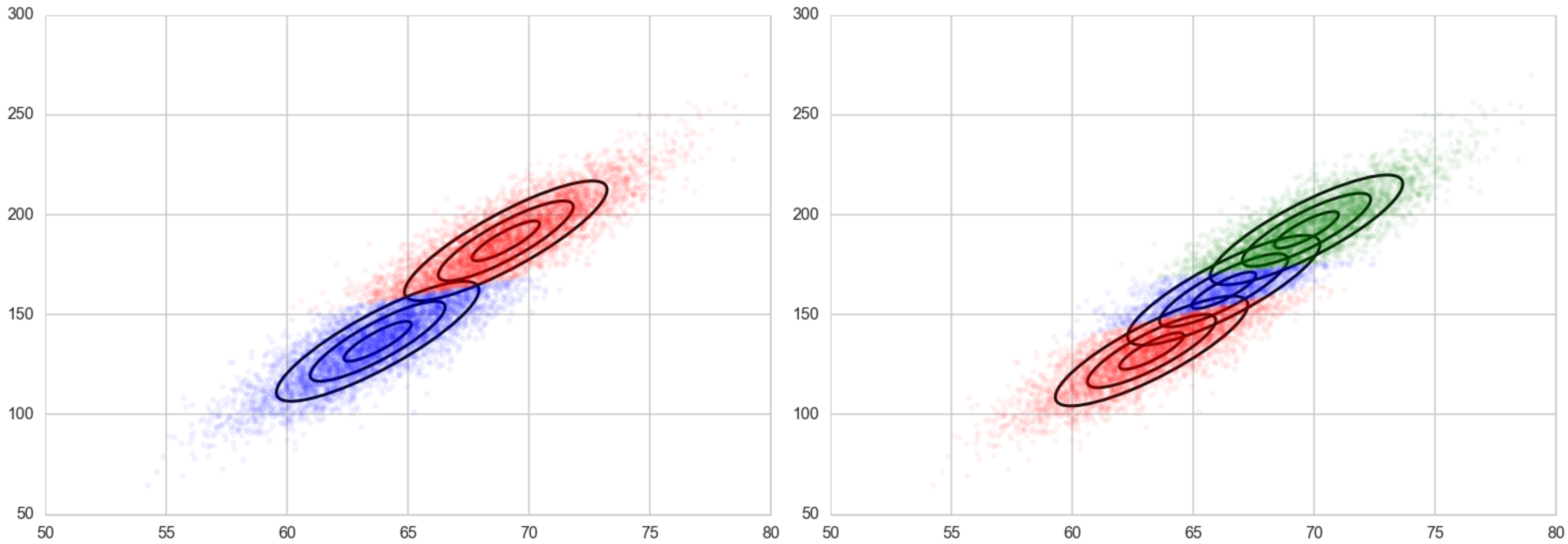
- LDA vs logistic respectively.
- Both have "generative" bayesian models: $p(c|x, \theta)$ or $p(y|x, \theta)$. Here think of $\mathbf{z} = \theta$
- LDA is generative as it models $p(x|c)$ while logistic models $p(c|x)$ directly. Here think of $\mathbf{z} = c$
- we do know c on the training set, so think of the unsupervised learning counterparts of these models where you dont know c

Generative vs Discriminative classifiers (contd)

- generative handles data asymmetry better
- sometimes generative models like LDA and Naive Bayes are easy to fit. Discriminative models require convex optimization via Gradient descent
- can add new classes to a generative classifier without retraining so better for online customer selection problems
- generative classifiers can handle missing data easily
- generative classifiers are better at handling unlabelled training data (semi-supervized learning)
- preprocessing data is easier with discriminative classifiers
- discriminative classifiers give generally better callibrated probabilities
- discriminative usually less expensive


Learning
ERM/Discriminant
bayesian
Discriminative
Generative
More Modelling

Unsupervised: How many clusters $\mathbf{z}$?



Concrete Formulation of unsupervised learning

Estimate Parameters by $\mathbf{x}$-MLE:

$$
\begin{aligned}
l(x|\lambda, \mu, \Sigma) &= \sum_{i=1}^{m} \log p(x_i|\lambda, \mu, \Sigma) \\
&= \sum_{i=1}^{m} \log \sum_{z} p(x_i|z_i, \mu, \Sigma)\, p(z_i|\lambda)
\end{aligned}
$$

Not Solvable analytically! EM and Variational. Or do MCMC.

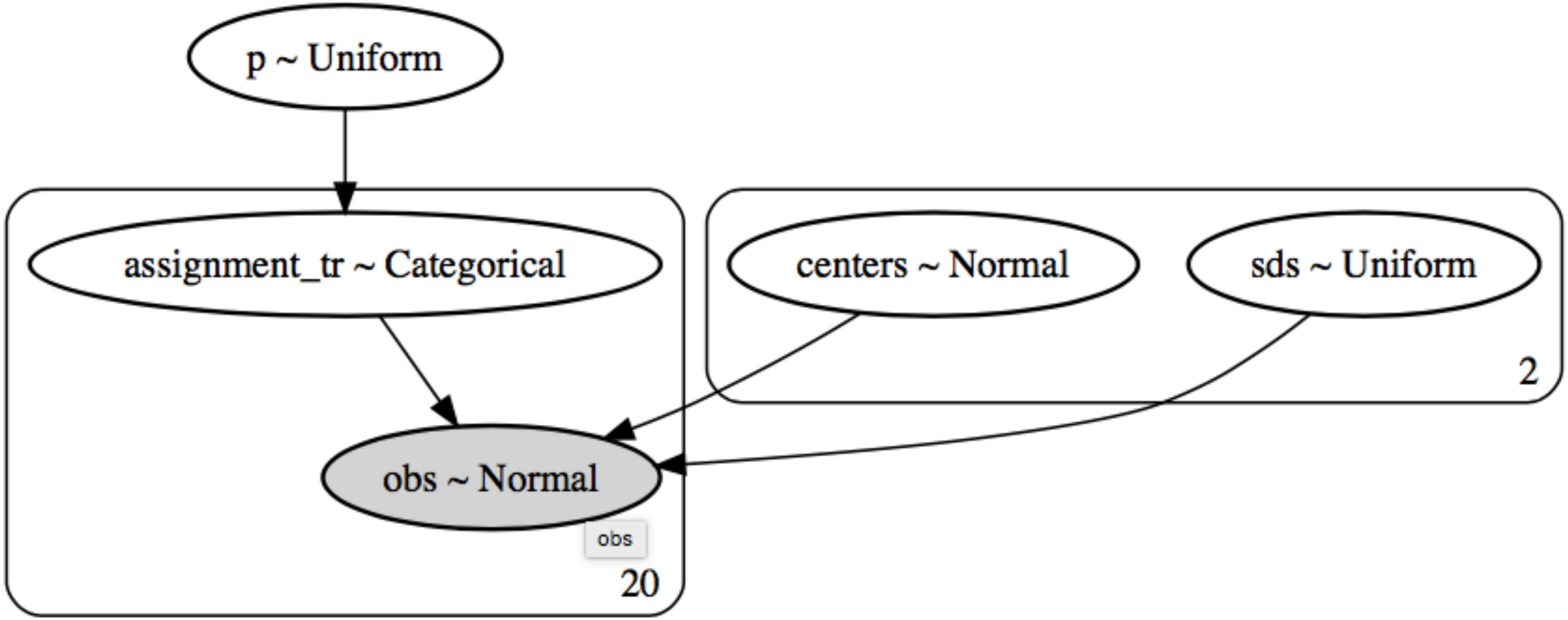
Bayesian Unsupervised

```python
with pm.Model() as ofmodel:
    p1 = pm.Uniform('p', 0, 1)
    p2 = 1 - p1
    p = tt.stack([p1, p2])
    assignment_tr = pm.Categorical("assignment_tr", p,
                    shape=ztr.shape[0])
    sds = pm.Uniform("sds", 0, 100, shape=2)

    centers = pm.Normal("centers",
                        mu=np.array([130, 170]),
                        sd=np.array([20, 20]),
                        shape=2)

    # and to combine it with the observations:
    observations = pm.Normal("obs", mu=centers[assignment_tr], sd=sds[assignment_tr], observed=xtr)
```

Unsupervised graph



Semi-supervised learning

We have some labels, but typically very few labels: not enough to form a good training set. Likelihood a combination.

$$l(\{x_i\},\{x_j\},\{z_i\}|\theta,\lambda) = \sum_i log\, p(x_i, z_i|\lambda,\theta) + \sum_j log\, p(x_j|\lambda,\theta)$$
$$= \sum_i log\, p(z_i|\lambda)p(x_i|z,\theta) + \sum_j log \sum_z p(z_j|\lambda)p(x_j|z_j,\theta)$$

Here i ranges over the data points where we have labels, and j over the data points where we dont.

Semi-supervised learning

Basic Idea: there is structure in $p(x)$ which might help us divine the conditionals, thus combine full-data and $\mathbf{x}$-likelihood.

Include x on the validation set in the likelihood, and x and z on the training set in the likelihood.

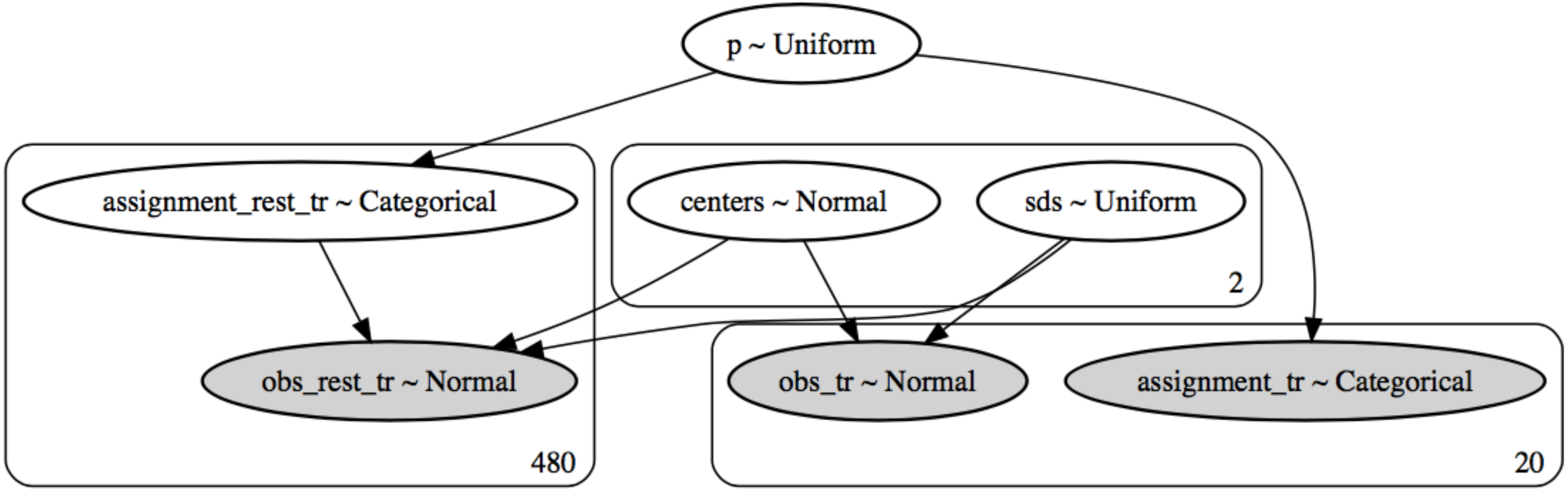
Has been very useful for Naive Bayes.

Bayesian Semi-Supervised

```python
with pm.Model() as classmodel2:
    p1 = pm.Uniform('p', 0, 1)
    p2 = 1 - p1
    p = tt.stack([p1, p2])
    assignment_tr = pm.Categorical("assignment_tr", p,
                                   observed=ztr)
    # we do not know the assignments for the rest
    assignment_rest_tr = pm.Categorical("assignment_rest_tr", p,
                                   shape=xte.shape[0])
    sds = pm.Uniform("sds", 0, 100, shape=2)
    centers = pm.Normal("centers",
                        mu=np.array([130, 170]),
                        sd=np.array([20, 20]),
                        shape=2)

    # and to combine it with the observations:
    observations_tr = pm.Normal("obs_tr", mu=centers[assignment_tr], sd=sds[assignment_tr], observed=xtr)
    observations_te = pm.Normal("obs_rest_tr", mu=centers[assignment_rest_tr], sd=sds[assignment_rest_tr], observed=xte)
```

Semi-supervised graph



EXPECTATION MAXIMIZATION

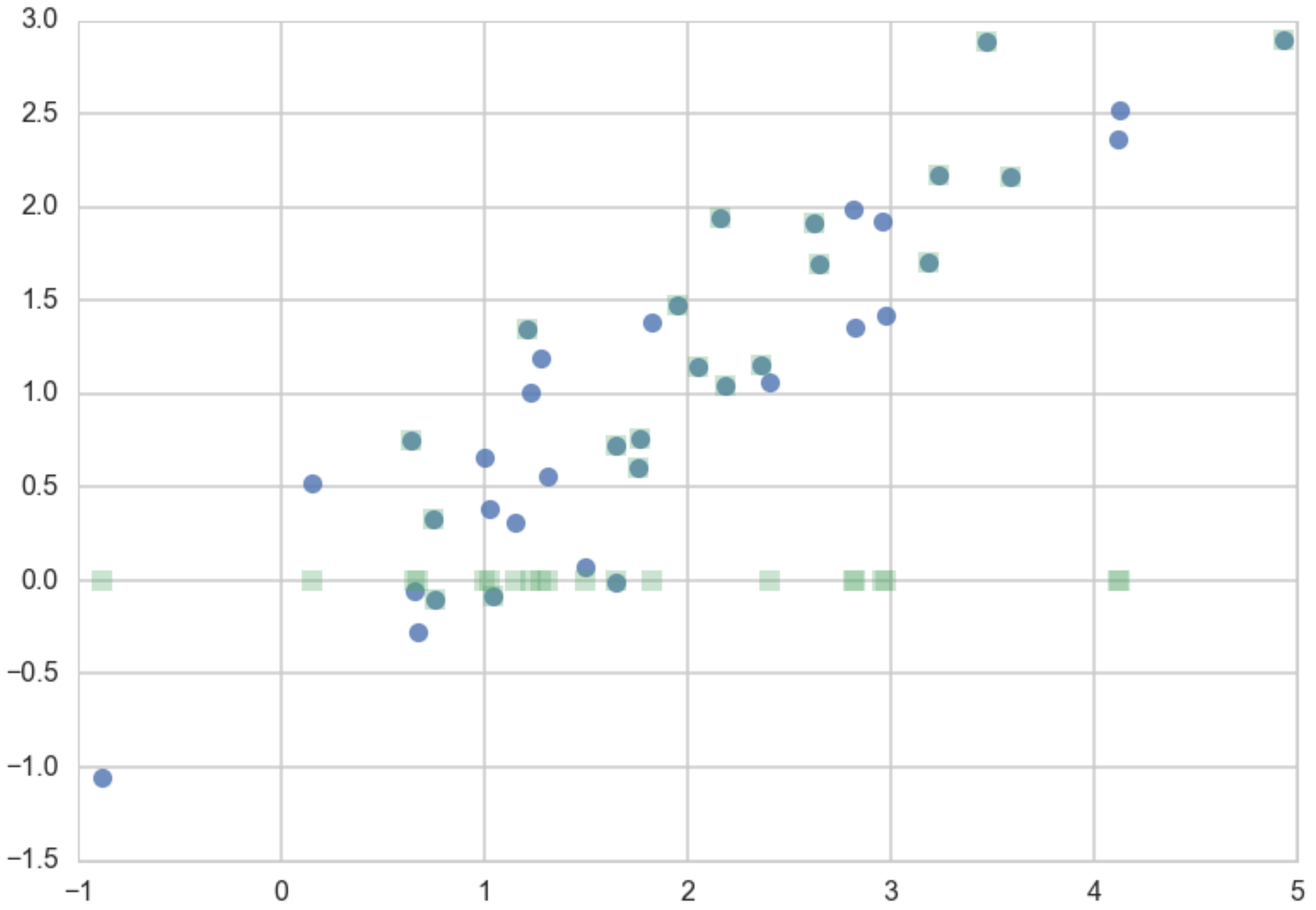
calculate MLE estimates for the incomplete data problem by using the complete-data likelihood. To create complete data, augment the observed data with manufactured data

Toy Example: 2D Gaussian

$$\begin{pmatrix} x_{1i} \\ x_{2i} \end{pmatrix} \overset{\text{ind}}{\sim} \mathcal{N}_2\left(\begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \begin{pmatrix} \sigma_1^2 & \sigma_1\sigma_2\rho \\ \sigma_1\sigma_2\rho & \sigma_2^2 \end{pmatrix}\right)$$

```
sig1=1
sig2=0.75
mu1=1.85
mu2=1
rho=0.82
means=np.array([mu1, mu2])
cov = np.array([
    [sig1**2, sig1*sig2*rho],
    [sig2*sig1*rho, sig2**2]
])
```

Lose z = 20 y-values. Set to 0.



MLE for full data problem

$$\hat{\mu}_1 = \sum_1^{40} x_{1i}/40, \quad \hat{\mu}_2 = \sum_1^{40} x_{2i}/40,$$

$$\hat{\sigma}_1 = \left[\sum_1^{40} (x_{1i} - \hat{\mu}_1)^2 /40\right]^{1/2}, \quad \hat{\sigma}_2 = \left[\sum_1^{40} (x_{2i} - \hat{\mu}_2)^2 /40\right]^{1/2},$$

$$\hat{\rho} = \left[\sum_1^{40} (x_{1i} - \hat{\mu}_1)(x_{2i} - \hat{\mu}_2)/40\right] \Big/ (\hat{\sigma}_1 \hat{\sigma}_2),$$

```
mu1 = lambda s: np.mean(s[:,0])
mu2 = lambda s: np.mean(s[:,1])
s1 = lambda s: np.std(s[:,0])
s2 = lambda s: np.std(s[:,1])
rho = lambda s: np.mean((s[:,0] - mu1(s))*(s[:,1]
    - mu2(s)))/(s1(s)*s2(s))
```

But we dont have full data.

Use Censored data with initial imputation

M-step: Maximizing full-data MLE

```
mu1s.append(mu1(samples_censored))
mu2s.append(mu2(samples_censored))
s1s.append(s1(samples_censored))
s2s.append(s2(samples_censored))
rhos.append(rho(samples_censored))
```

M-step done. Use these parameters let us calculate new y-values.

Replace the old-missing-y values (0s) with the means of these fixing the parameters of the multi-variate normal and the non-missing data.

E-step

Use expectation from hidden-data posterior distrib: $E_{p(z|\theta,x)}[z]$

This **posterior** distribution (in the sense of bayes theorem, not bayesian analysis) for the multi-variate gaussian is a gaussian..see wikipedia for the formulae

$$\bar{y}(t+1) - \hat{\mu_2}(t) = \hat{\rho}(t) \frac{\hat{\sigma_2}(t)}{\hat{\sigma_1}(t)} (\bar{x} - \hat{\mu_1}(t))$$

Iterate

```
def ynew(x, mu1, mu2, s1, s2, rho):
    return mu2 + rho*(s2/s1)*(x - mu1)

newys=ynew(samples_censored[20:,0], mu1s[0], mu2s[0], s1s[0], s2s[0], rhos[0])

for step in range(1,20):
    samples_censored[20:,1] = newys
    #M-step
    mu1s.append(mu1(samples_censored))
    mu2s.append(mu2(samples_censored))
    s1s.append(s1(samples_censored))
    s2s.append(s2(samples_censored))
    rhos.append(rho(samples_censored))
    #E-step
    newys=ynew(samples_censored[20:,0], mu1s[step], mu2s[step], s1s[step], s2s[step], rhos[step])
```

Voila. We converge to stable values of our parameters. Initials:

```
sig1=1
sig2=0.75
mu1=1.85
mu2=1
rho=0.82
```

But they may not be the ones we seeded the samples with. The EM algorithm is only good upto finding local minima, and a finite sample size also means that the minimum found can be slightly different.

	mu1	mu2	rho	s1	s2
0	1.966883	0.662900	0.522613	1.185731	0.889247
1	1.966883	0.949428	0.850340	1.185731	0.782217
2	1.966883	1.073320	0.926036	1.185731	0.811543
3	1.966883	1.126917	0.941491	1.185731	0.837711
4	1.966883	1.150122	0.945313	1.185731	0.851228
5	1.966883	1.160180	0.946476	1.185731	0.857421
6	1.966883	1.164547	0.946888	1.185731	0.860139
7	1.966883	1.166447	0.947048	1.185731	0.861307
8	1.966883	1.167277	0.947113	1.185731	0.861801
9	1.966883	1.167641	0.947139	1.185731	0.862008
10	1.966883	1.167802	0.947150	1.185731	0.862092
11	1.966883	1.167874	0.947154	1.185731	0.862125
12	1.966883	1.167907	0.947156	1.185731	0.862137
13	1.966883	1.167922	0.947156	1.185731	0.862141
14	1.966883	1.167929	0.947157	1.185731	0.862142
15	1.966883	1.167933	0.947157	1.185731	0.862142
16	1.966883	1.167934	0.947156	1.185731	0.862142
17	1.966883	1.167935	0.947156	1.185731	0.862141
18	1.966883	1.167936	0.947156	1.185731	0.862141
19	1.966883	1.167936	0.947156	1.185731	0.862141

The EM algorithm, conceptually

- iterative method for maximizing difficult likelihood (or posterior) problems, first introduced by Dempster, Laird, and Rubin in 1977
- Sorta like, just assign points to clusters to start with and iterate.
- Then, at each iteration, replace the augmented data by its conditional expectation given current observed data and parameter estimates. (E-step)
- Maximize the full-data likelihood (M-step).

Why does it work?

$$p(x|\theta) = \sum_z p(x, z|\theta)$$

where the x and z range over the multiple points in your data set.

Then x-data log-likelihood $\ell(x|\theta) = log\, p(x|\theta) = log \sum_z p(x, z|\theta)$.

Hard to maximize for us.

Assume z has some normalized distribution:

$$z \sim q(z).$$

We wish to compute conditional expectations of the type:

$$E_{p(z|x,\theta)}[z]$$

but we dont know this "posterior" (henceforth p).

Lets say we somehow know q.

Consider KL loss function

$$\mathrm{KL}(q||p) = D_{KL}(q,p) = E_q[log \frac{q}{p}] = -E_q[log \frac{p}{q}]$$

$$D_{KL}(q,p) = -E_q[log \frac{p(x,z|\theta)}{q\,p(x|\theta)}]$$

$$D_{KL}(q,p) = -\left(E_q[log \frac{p(x,z|\theta)}{q}] - E_q[log\, p(x|\theta)]\right)$$

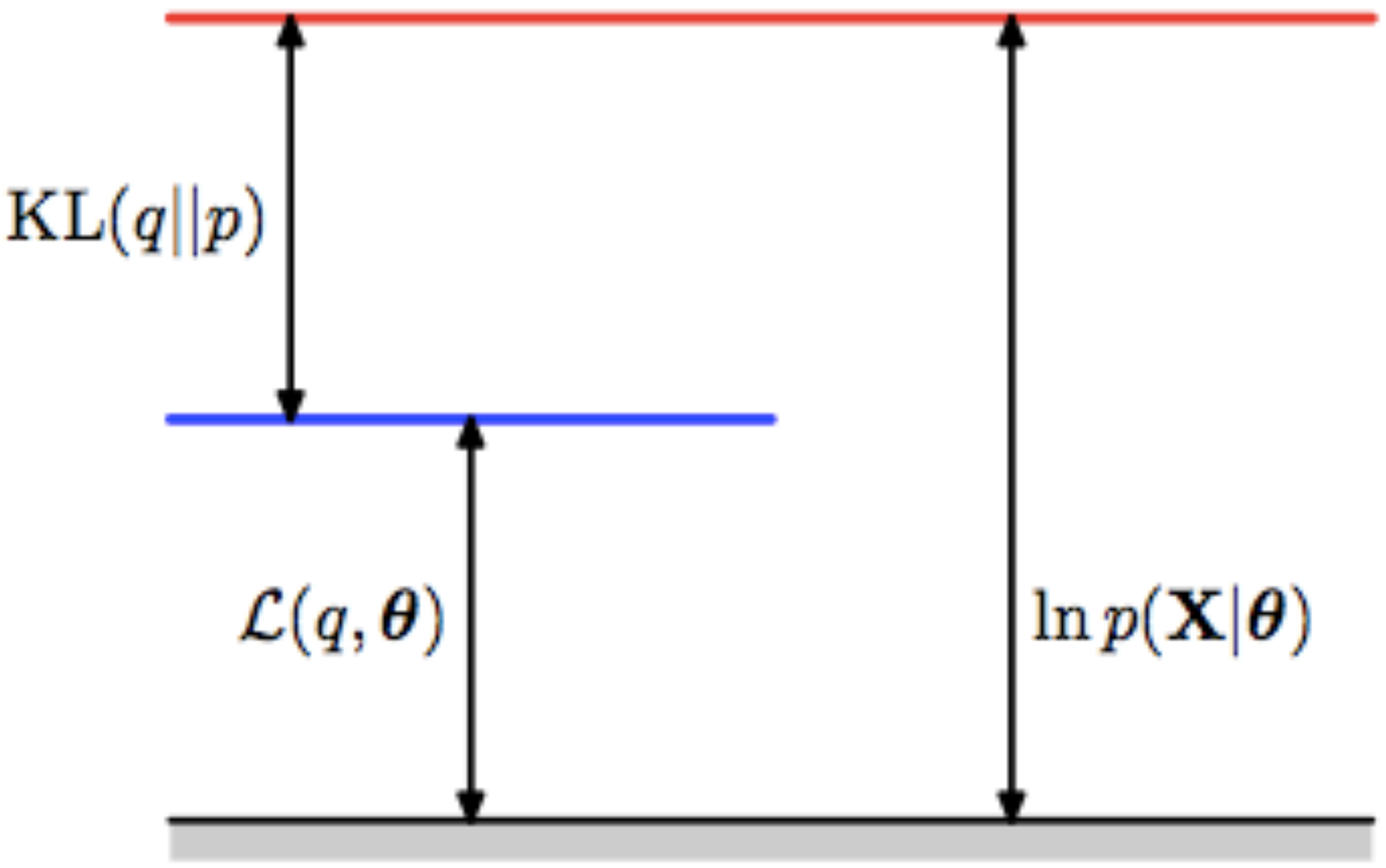
x-data likelihood

$$log\, p(x|\theta) = E_q[log \frac{p(x,z|\theta)}{q}] + D_{KL}(q,p)$$

If we define the ELBO or Evidence Lower bound as:

$$\mathcal{L}(q,\theta) = E_q[log \frac{p(x,z|\theta)}{q}]$$

then $log\, p(x|\theta)$ = ELBO + KL-divergence



- KL divergence only 0 when $p = q$ exactly everywhere
- minimizing KL means maximizing ELBO
- ELBO $\mathcal{L}(q, \theta)$ is a lower bound on the log-likelihood.
- ELBO is average full-data likelihood minus entropy of q:

$$\mathcal{L}(q, \theta) = E_q[log \frac{p(x, z|\theta)}{q}] = E_q[log p(x, z|\theta)] - E_q[log\, q]$$

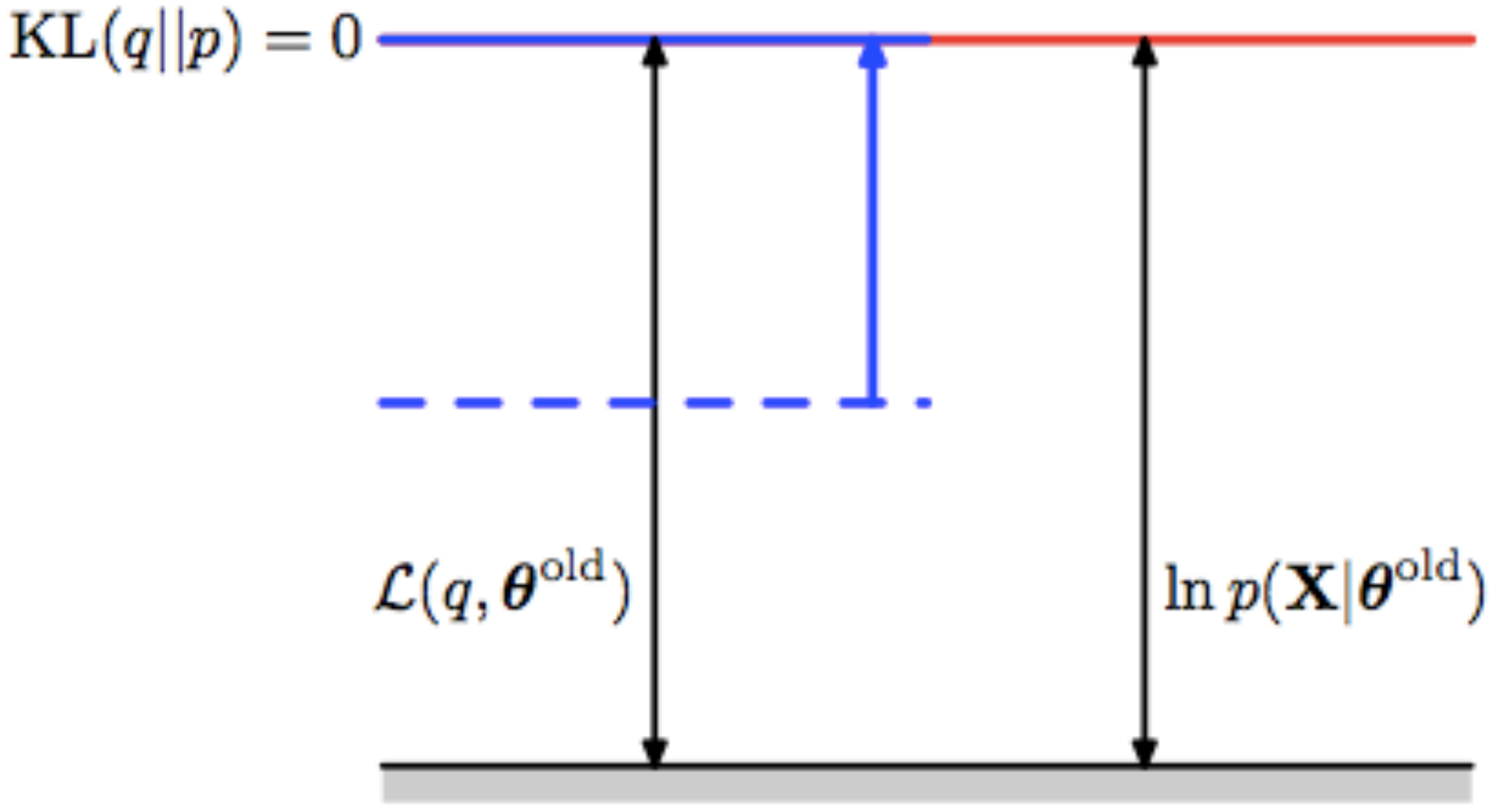
E-step conceptually

Choose at some (possibly initial) value of the parameters θ_{old},

$$q(z) = p(z|x, \theta_{old}),$$

then KL divergence = 0, and thus $\mathcal{L}(q, \theta)$ = log-likelihood at θ_{old}, maximizing the ELBO.

Conditioned on observed data, and θ_{old}, we use q to **conceptually** compute the expectation of the missing data.

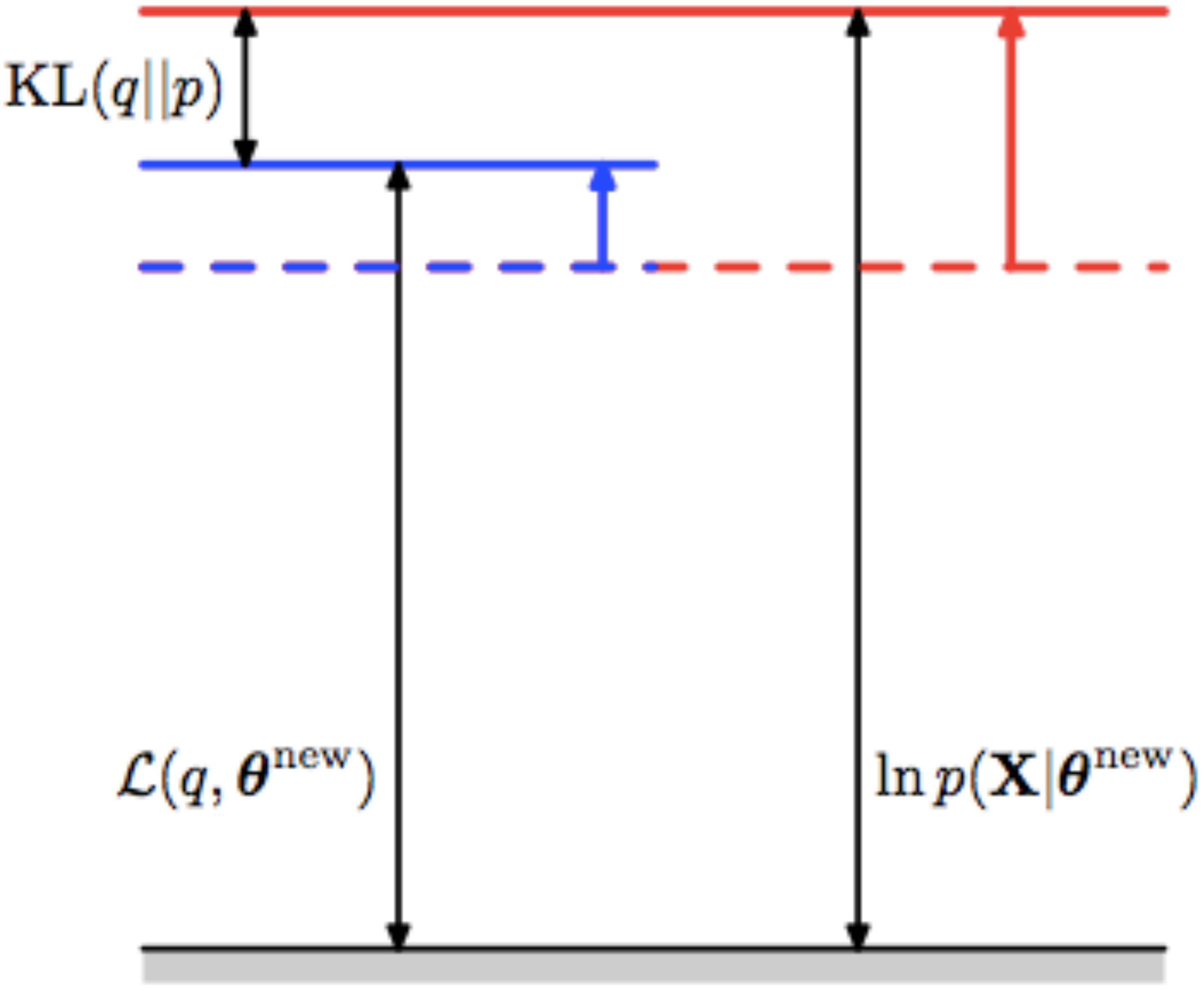


E-step: what we actually do

Compute the Auxilary function, $Q(\theta, \theta^{(t-1)})$, the expected complete(full) data log likelihood, defined by:

$$Q(\theta, \theta^{(t-1)}) = E_{Z|Y=y,\Theta=\theta^{t-1}}[logp(x, z|\theta)]$$

or the expectation of the ELBO instead of Q.



M-step

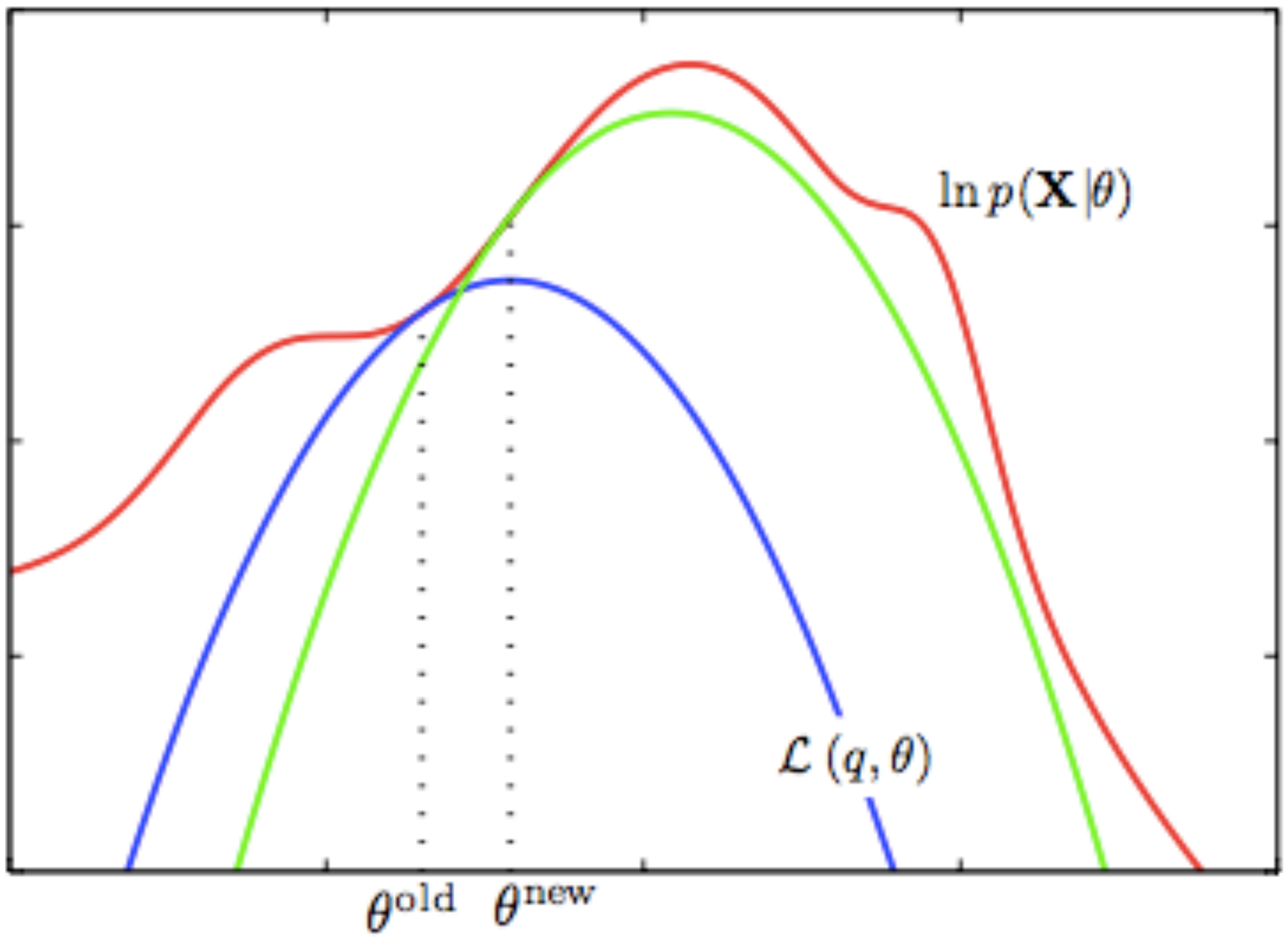
After E-step, ELBO touches $\ell(x|\theta)$, any maximization wrt θ will also “push up” on likelihood, thus increasing it.

Thus hold $q(z)$ fixed at the z-posterior calculated at θ_{old}, and maximize ELBO $\mathcal{L}(q, \theta, \theta_{old})$ or $Q(q, \theta, \theta_{old})$ wrt θ to obtain new θ_{new}.

In general $q(\theta_{old}0 \neq p(z|x, \theta_{new})$, hence KL $\neq 0$. Thus increase in $\ell(x|\theta) \geq$ increase in ELBO.

Process

1. Start with $p(x|\theta)$(red curve), θ_{old}.
2. Until convergence:
    1. E-step: Evaluate $q(z, \theta_{old}) = p(z|x, \theta_{old})$ which gives rise to $Q(\theta, \theta_{old})$ or $ELBO(\theta, \theta_{old})$(blue curve) whose value equals the value of $p(x|\theta)$ at θ_{old}.
    2. M-step: maximize Q or $ELBO$ wrt θ to get θ_{new}.
    3. Set $\theta_{old} = \theta_{new}$



An iteration:

$$\ell(\theta_{t+1}) \geq \mathcal{L}(q(z, \theta_t), \theta_{t+1}) \geq \mathcal{L}(q(z, \theta_t), \theta_t) = \ell(\theta_t)$$

The first equality follows since $\mathcal{L}$ is a lower bound on ℓ, the second from the M-step's maximization of $\mathcal{L}$, and the last from the vanishing of the KL-divergence after the E-step.

As a consequence, you **must** observe monotonic increase of the observed-data log likelihood ℓ across iterations. **This is a powerful debugging tool for your code**.

EM is local only!

Note that as shown above, since each EM iteration can only improve the likelihood, you are guaranteeing convergence to a local maximum. Because it IS local , you must try some different initial values of θ_{old} and take the one that gives you the largest ℓ.

GMM

E-step: Calculate $w_{i,j} = q_i(z_i = j) = p(z_i = j | x_i, \lambda, \mu, \Sigma)$

M-step: maximize: $\mathcal{L} = \sum_i \sum_{z_i} q_i(z_i) \log \frac{p(x_i, z_i | \lambda, \mu, \Sigma)}{q_i(z_i)}$

$$\mathcal{L} = \sum_i \sum_{j=i}^{k} q_i(z_i = j) \log \frac{p(x_i | z_i = j, \mu, \Sigma) p(z_i = j | \lambda)}{q_i(z_i = j)}$$

$$\mathcal{L} = \sum_{i=1}^{m} \sum_{j=i}^{k} w_{i,j} \log \left[\frac{\frac{1}{(2\pi)^{n/2} |\Sigma_j|^{1/2}} \exp\left(-\frac{1}{2} (x_i - \mu_j)^T \Sigma_j^{-1} (x_i - \mu_j) \right) \lambda_j}{w_{i,j}} \right]$$

M-step

Taking derivatives yields following updating formulas:

$$\lambda_j = \frac{1}{m}\sum_{i=1}^{m} w_{i,j}$$

$$\mu_j = \frac{\sum_{i=1}^{m} w_{i,j}\, x_i}{\sum_{i=1}^{m} w_{i,j}}$$

$$\Sigma_j = \frac{\sum_{i=1}^{m} w_{i,j}\, (x_i - \mu_j)(x_i - \mu_j)^T}{\sum_{i=1}^{m} w_{i,j}}$$

E-step: calculate responsibilities

We are basically calculating the posterior of the z's given the x's and the current estimate of our parameters. We can use Bayes rule

$$w_{i,j} = p(z_i = j | x_i, \lambda, \mu, \Sigma) = \frac{p(x_i | z_i = j, \mu, \Sigma)\, p(z_i = j | \lambda)}{\sum_{l=1}^{k} p(x_i | z_i = l, \mu, \Sigma)\, p(z_i = l | \lambda)}$$

Where $p(x_i | z_i = j, \mu, \Sigma)$ is the density of the Gaussian with mean μ_j and covariance Σ_j at x_i and $p(z_i = j | \lambda)$ is simply λ_j.

```python
def Estep(x, mu, sigma, lam):
    a = lam * norm.pdf(x, mu[0], sigma[0])
    b = (1. - lam) * norm.pdf(x, mu[1], sigma[1])
    return b / (a + b)

def Mstep(x, w):
    lam = np.mean(1.-w)

    mu = [np.sum((1-w) * x)/np.sum(1-w), np.sum(w * x)/np.sum(w)]

    sigma = [np.sqrt(np.sum((1-w) * (x - mu[0])**2)/np.sum(1-w)),
             np.sqrt(np.sum(w * (x - mu[1])**2)/np.sum(w))]

    return mu, sigma, lam
```

```
0.4 [2, 5] [0.6, 0.6]
Initials, mu: [-4.85176052  5.51133343]
Initials, sigma: [ 2.02807915  3.58912888]
Initials, lam: 0.5418931691319009
Iterations 71
A: N(2.0261, 0.5936)
B: N(5.0083, 0.6288)
lam: 0.5884

0.4 [2, 5] [0.6, 0.6]
Initials, mu: [ 11.09643621  -4.48315085]
Initials, sigma: [ 4.31750531  0.95518757]
Initials, lam: 0.5767814041950222
Iterations 103
A: N(5.0083, 0.6288)
B: N(2.0261, 0.5936)
lam: 0.4116
```

Compared to supervised classification and k-means

- M-step formulas vs GDA we can see that are very similar except that instead of using δ functions we use the w's.
- Thus the EM algorithm corresponds here to a weighted maximum likelihood and the weights are interpreted as the 'probability' of coming from that Gaussian
- Thus we have achieved a **soft clustering** (as opposed to k-means in the unsupervised case and classification in the supervised case).

- kmeans is HARD EM. Instead of calculating Q in e-step, use mode of z posterior. Also the case with classification
- finite mixture models suffer from multimodality, non-identifiability, and singularity. They are problematic but useful
- models can be singular if cluster has only one data point: overfitting
- add in prior to regularise and get MAP. Add log(prior) in M-step only