Lecture 25

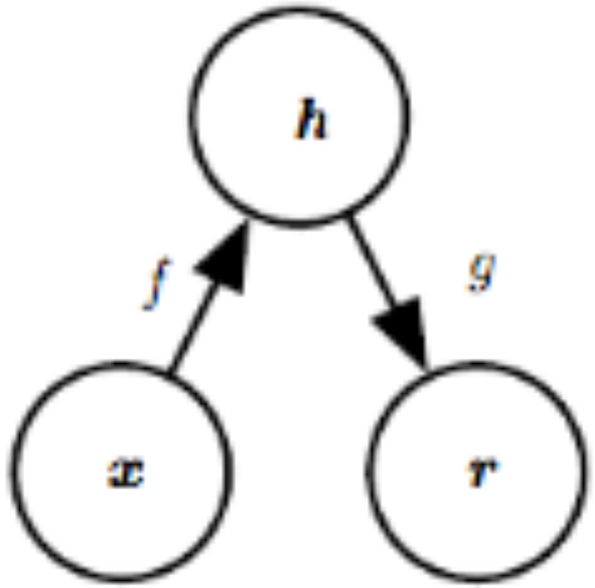
Variational Inference

Autoencoders and Mixture Models

- we have not talked about learning θ, the parametrization of $p_\theta(x, z)$.
- also, what if there are data-point specific parameters in out model. In other words we have a ϕ_i, where i indexes the data-points
- an example of this is topic modeling or generative image modeling
- we can do this by fitting a ϕ_i as a regression model on the x_i. And we can make this model non-linear by considering an ANN!

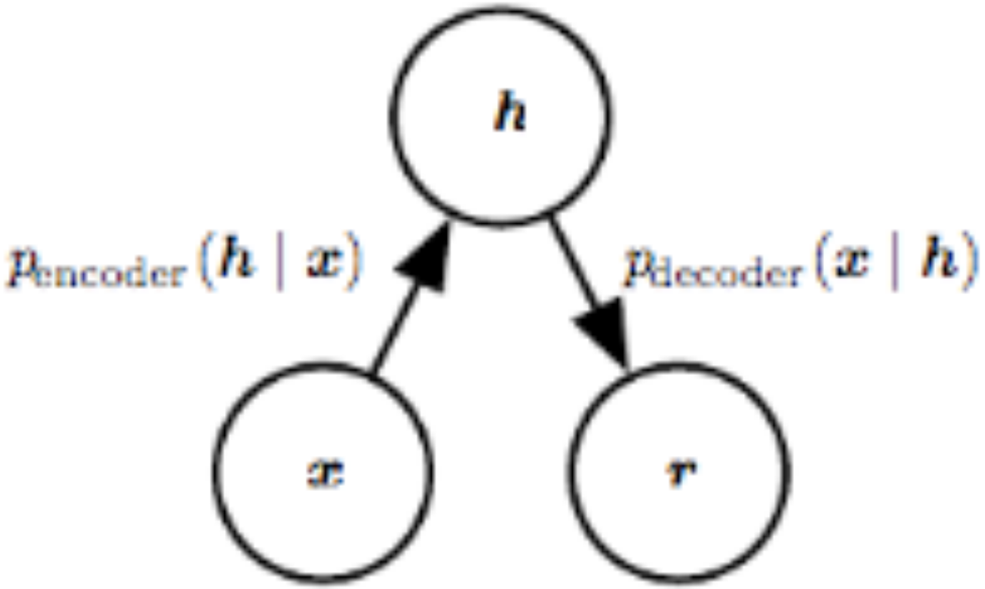
Variational Autoencoders

Autoencoders: basic idea



- **h** is the representation. An *undercomplete* autoencoder makes **h** of smaller dimension than x
- f is the encoder and g the decoder
- simplest idea: minimize $L(x, g(f(x)))$

- can think of an autoencoder as a way of approximately training a generative model.
- the features of the autoencoder describe the latent variables that explain the input
- can go deep!
- generalize to a stochastic autoencoder. The standard autoencoder then is a specific hidden state h or z



Variational Autoencoder

- just as in ADVI, we want to learn an approximate "encoding posterior" $p(z|x)$
- note that we have now again gone back to thinking of z as a (possibly) deep latent variable, or "representation".

We know how to do this:

ELBO maximization

Basic Setup in VAE

$KL + ELBO = log(p(x))$: ELBO bounds log(evidence)

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

$$\implies ELBO(q) = E_{q(z)}[(log(p(x|z))] - KL(q(z)||p(z))$$

(likelihood-prior balance)

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

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

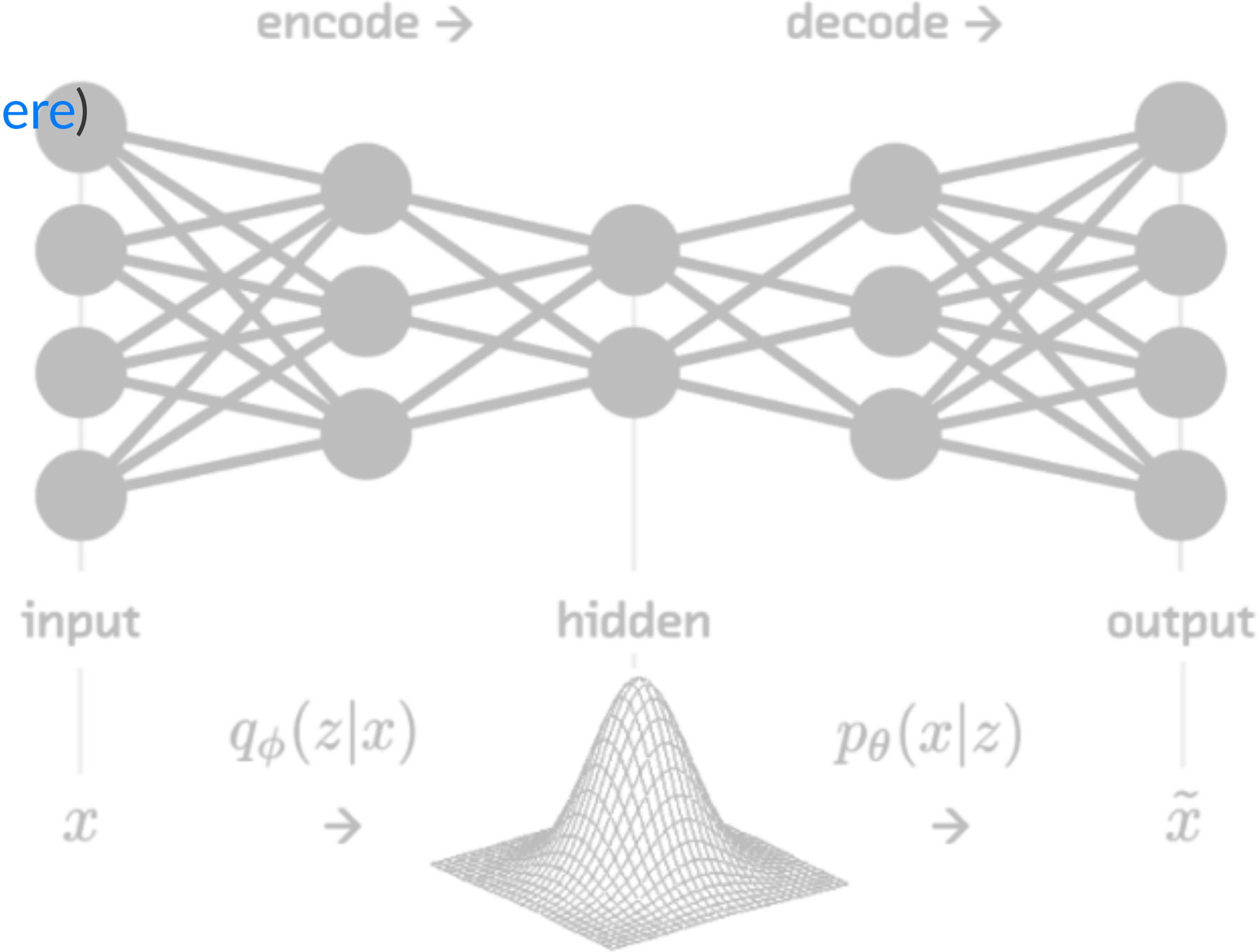
- **The likelihood posits a data generating process**, where the data $\mathbf{x}$ are assumed drawn from the likelihood conditioned on a particular hidden pattern described by $\mathbf{z}$.
- The *prior* $p(\mathbf{z})$ is a probability distribution that describes the latent variables present in the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

The Game

$$ELBO(q) = E_{q(z|x)}[(log(p(x|z))] - KL(q(z|x)||p(z))$$

- get z samples coming for fixed x, $q(z|x)$- to be close to some prior, $p(z)$, typically chosen as an isotropic gaussian...the regularization term
- first term is called "reconstruction loss", or "capacity of model to generate something like the data".

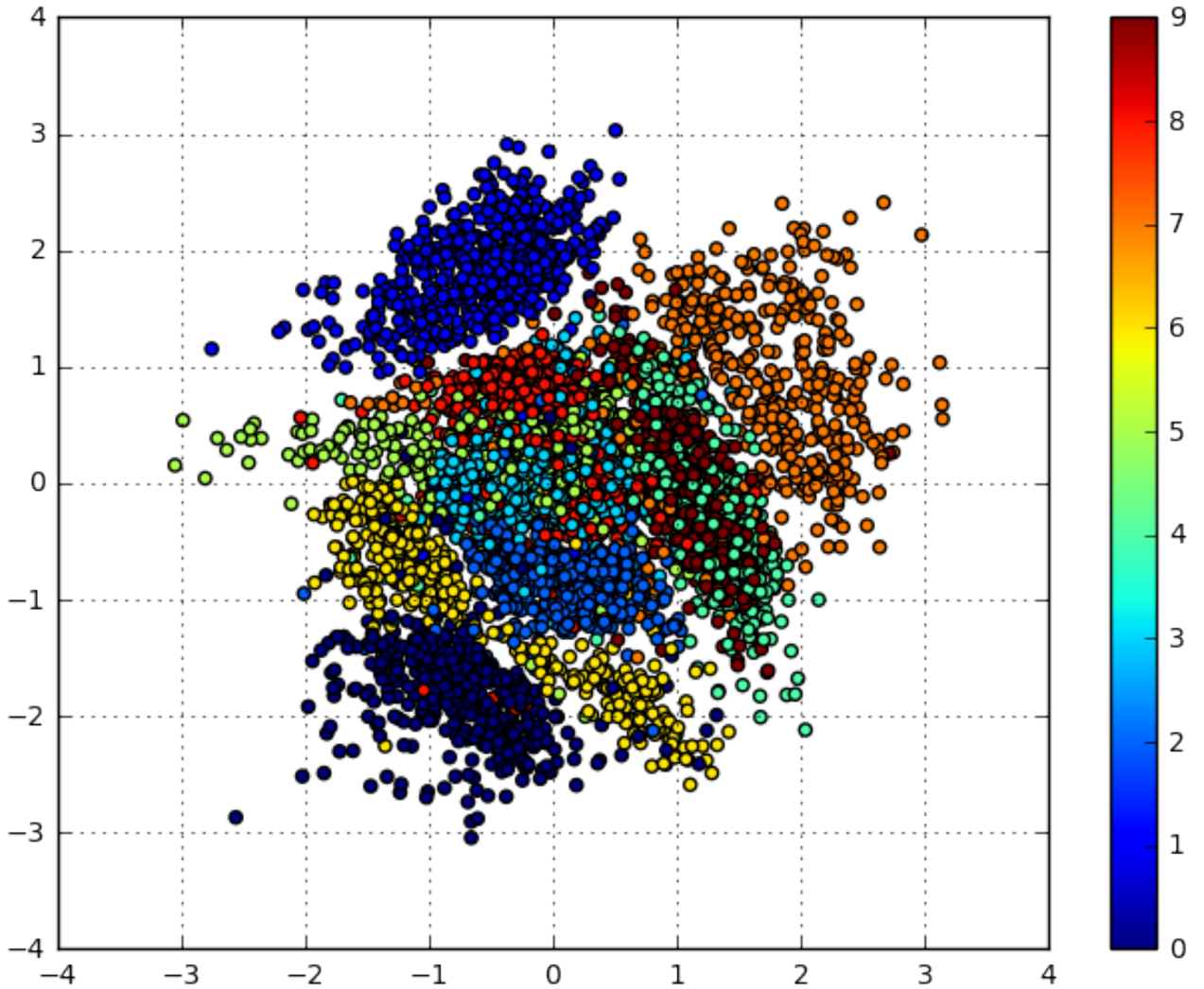
(from here)

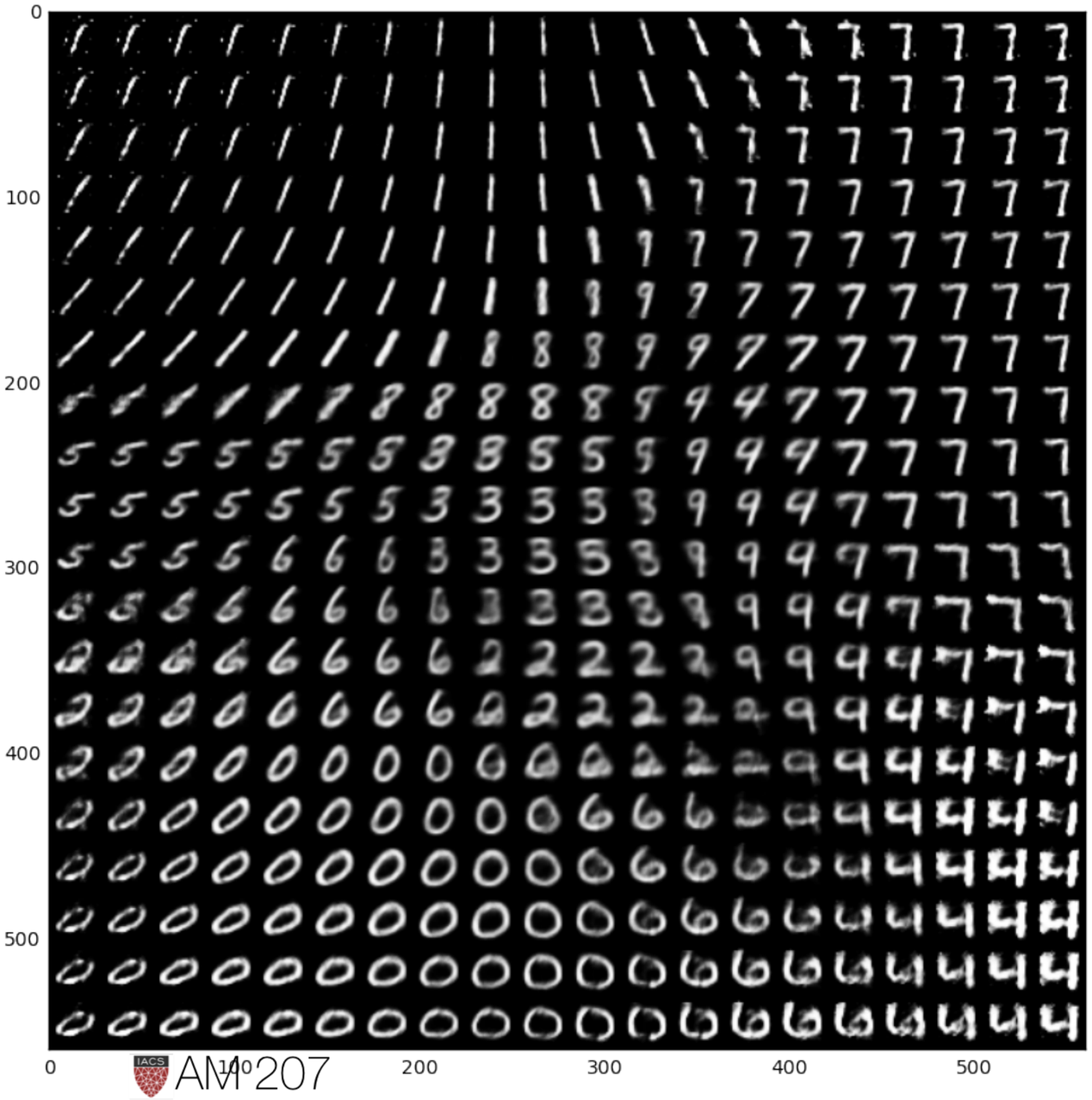

encode →
decode →
input
hidden
output
$q_\phi(z|x)$
$p_\theta(x|z)$
x
→
→
$\tilde{x}$

VAE steps for MNIST

- details in original paper and notebook
- linear encoder for both μ and $log(\sigma^2)$
- then transformation to $N(0,1)$ to be able to take gradient inside expectation as in ADVI
- then decode using a loss: binary cross-entropy $p(x|z)$ (for images) minus KL

```python
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def encode(self, x):
        h1 = self.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparameterize(self, mu, logvar):
        if self.training:
            std = logvar.mul(0.5).exp_()
            eps = Variable(std.data.new(std.size()).normal_())
            return eps.mul(std).add_(mu)
        else:
            return mu

    def decode(self, z):
        h3 = self.relu(self.fc3(z))
        return self.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar
```



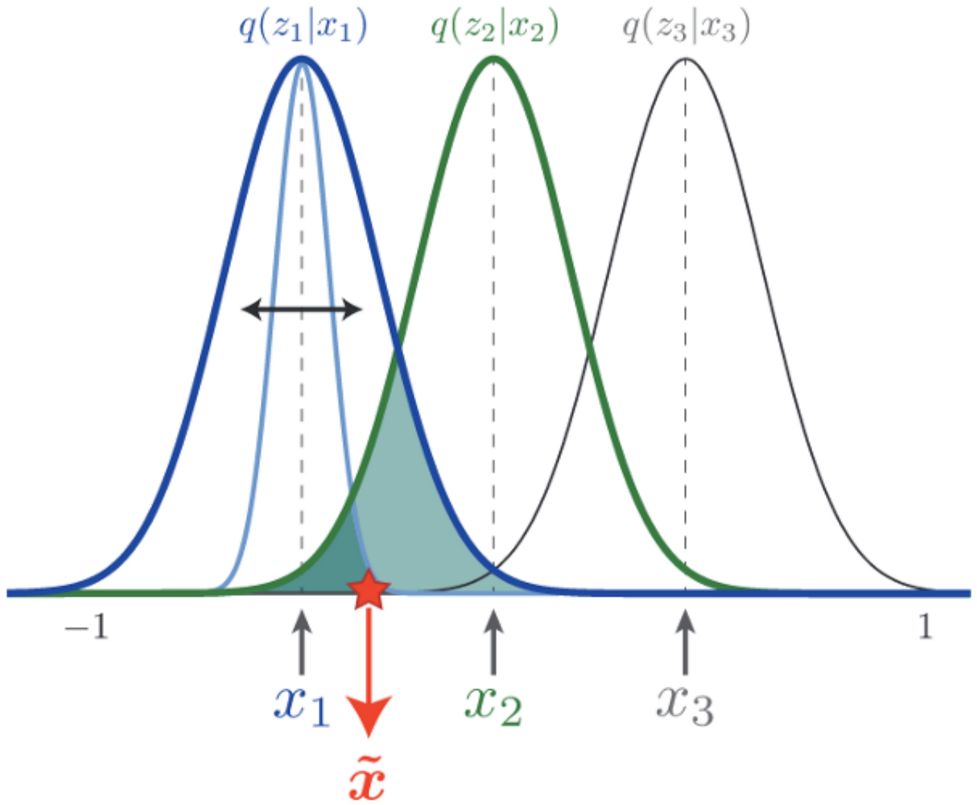



```
model = VAE()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x,
        x.view(-1, 784), size_average=False)
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

    return BCE + KLD

def train(epoch):
    model.train()
    train_loss = 0
    for batch_idx, (data, _) in enumerate(train_loader):
        data = Variable(data)
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(data)
        loss = loss_function(recon_batch, data, mu, logvar)
        loss.backward()
        train_loss += loss.data[0]
        optimizer.step()
    return train_loss / len(train_loader.dataset)

def test(epoch):
    model.eval()
    test_loss = 0
    for i, (data, _) in enumerate(test_loader):
        data = Variable(data, volatile=True)
        recon_batch, mu, logvar = model(data)
        test_loss += loss_function(recon_batch, data, mu, logvar).data[0]
    test_loss /= len(test_loader.dataset)
    return test_loss
```

Images from here

Disentanglement Issues

- can be understood from a gaussian mixtures perspective
- we would prefer data locality
- thus crank up the prior (regularization) term
- this is called the βVAE



How to implement?

- possible in pytorch, also in pymc3
- see convolutional VAE for MNIST in pymc3
- notice that MNIST, which we did earlier as supervised is now being done unsupervised.

Why?

See pymc3 for e.g. for auto-encoding LDA

- variational auto-encoders algorithm which allows us to perform inference efficiently for large datasets
- use tunable and flexible encoders such as multilayer perceptrons (MLPs) as our variational distribution to approximate complex variational posterior
- then its just ADVI with mini-batch on PyMC3 or pytorch. Can use for any posterior, example LDA, or custom for MNIST

How good is variational Bayes?

- its used heavily for models like LDA (latent-dirichlet allocation)
- but surprisingly the "goodness-of-fit" of the posterior approximation has been handled on a case by case basis
- until now: see Yao et. al

The Bayesian Workflow

(from Betancourt, and Savage)

Prior to Observation

1. Define Data and interesting statistics
2. Build Model
3. Analyze the joint, and its data marginal (prior predictive) and its summary statistics
4. fit posteriors to simulated data to calibrate
    - check sampler diagnostics, and correlate with simulated data
    - use rank statistics to evaluate prior-posterior consistency
    - check posterior behaviors and behaviors of decisions

Posterior to Observation

1. Fit the Observed Data and Evaluate the fit
    - check sampler diagnostics, poor performance means generative model not consistent with actual data
2. Analyze the Posterior Predictive Distribution
    - do posterior predictive checks, now comparing actual data with posterior-predictive simulations
    - consider expanding the model
3. Do model comparison (if needed)
    - usually within a nested model, but you might want to apply a different modeling scheme, in which case use loo
    - you might want to ensemble instead

Two ideas from Yao et. al.

- pareto shape parameter k from PSIS tells you goodness of fit (see here for @junpenglao pymc3 implementation, WIP). The idea comes from the process of smoothing in LOOCV estimation
- VSBC (variational simulation based callibration) : Extends calibration from Bayesian Workflow to variational case. pymc3 experimentation by @junpenglao here, WIP

Model Comparison: How to handle non-nested models?

- cross-validation
- less data to fit so biased models
- we are not talking here about cross-validation to do hyperparameter optimization
- specifically we will use Leave-One-Out-Cross-Validation (LOOCV) with importance sampling

LOOCV

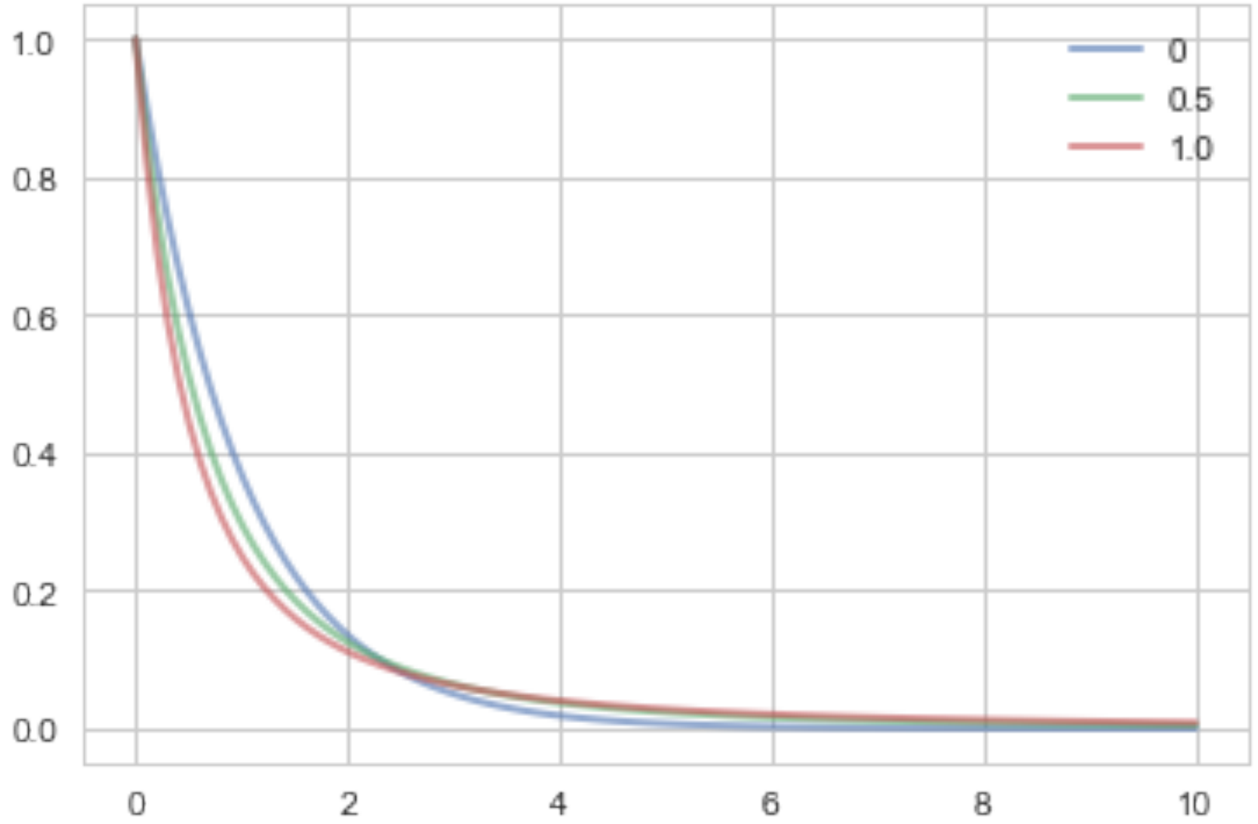
- The idea here is that you fit a model on N-1 data points, and use the Nth point as a validation point. Clearly this can be done in N ways.
- the N-point and N-1 point posteriors are likely to be quite similar, and one can sample one from the other by using importance sampling.

$$E_f[h] = \frac{\sum_s w_s h_s}{\sum_s w_s} \text{ where } w_s = f_s/g_s.$$

Fit the full posterior once. Then we have

$$w_s = \frac{p(\theta_s|y_{-i})}{p(\theta_s|y)} \propto \frac{1}{p(y_i|\theta_s, y_{-i})}$$

- the importance sampling weights can be unstable out in the tails.
- importance weights have a long right tail, pymc (`pm.loo`) fits a generalized pareto to the tail (largest 20% importance ratios) for each held out data point i (a MLE fit). This smooths out any large variations.

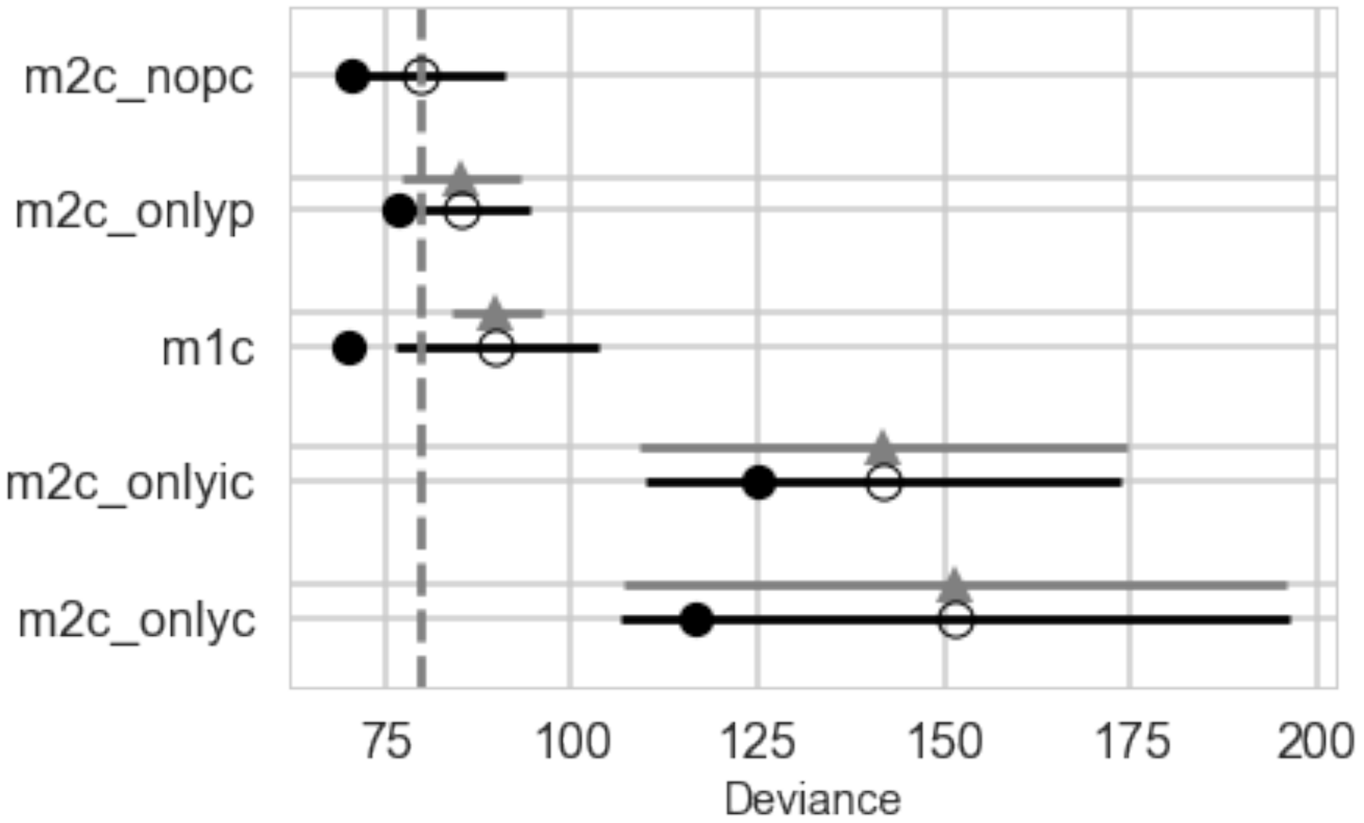


$$elpd_{loo} = \sum_i log(p(y_i|y_{-i}))$$

$$= \sum_i log\left(\frac{\sum_s w_{is} p(y_i|\theta_s)}{\sum_s w_{is}}\right)$$

over the training sample.

Oceanic tools LOOCV



What should you use?

1. LOOCV and WAIC are fine. The former can be used for models not having the same likelihood, the latter can be used with models having the same likelihood.

2. WAIC is fast and computationally less intensive, so for same-likelihood models (especially nested models where you are really performing feature selection), it is the first line of attack

3. One does not always have to do model selection. Sometimes just do posterior predictive checks to see how the predictions are, and you might deem it fine.

4. For hierarchical models, WAIC is best for predictive performance within an existing cluster or group. Cross validation is best for new observations from new groups

PSIS for variational posterior

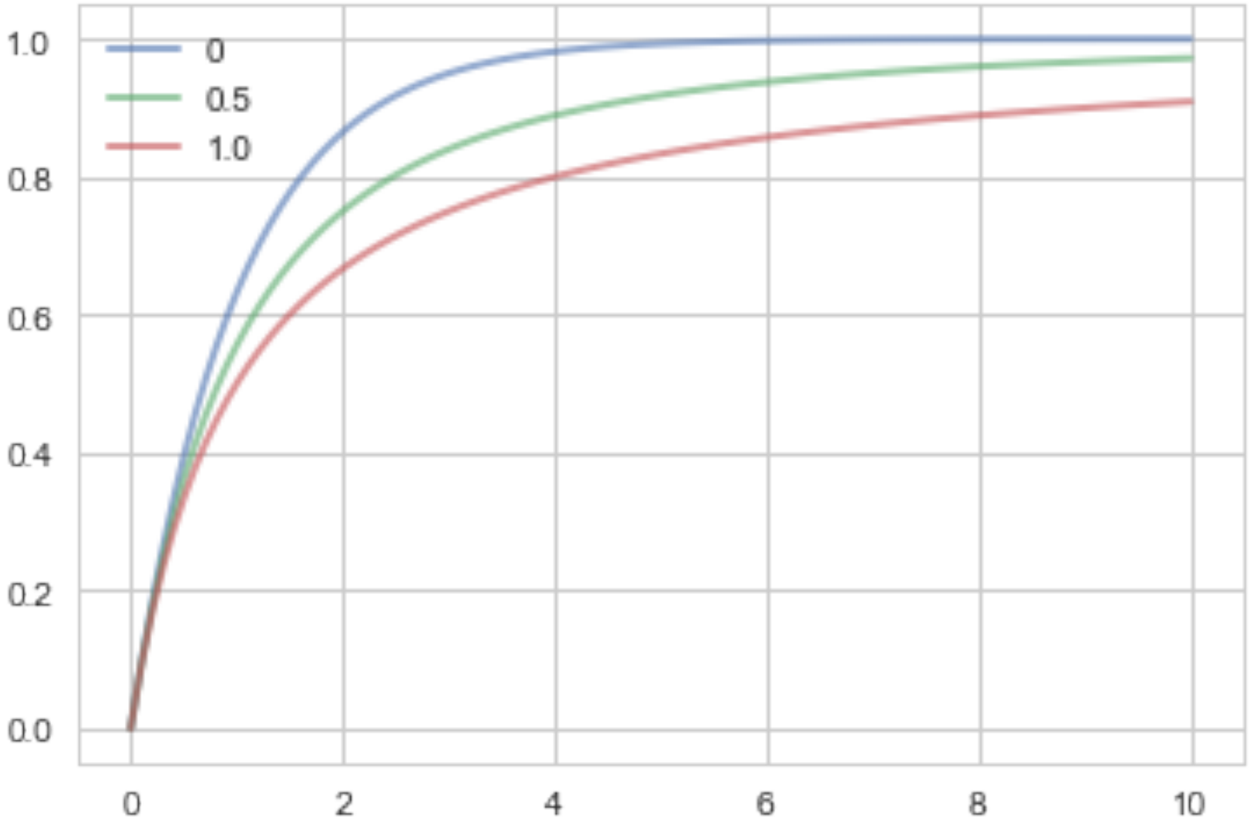
Want $E_p[h(\theta)]$. But we calculate $E_q[h(\theta)] = (1/S)\sum_s h(\theta_s)$ which is biased.

Use importance sampling: $E_p[h(\theta)] = \frac{\sum_s w_s h(\theta_s)}{\sum_s w_s}$ where

$w_s = p(\theta_s, y)/q$.

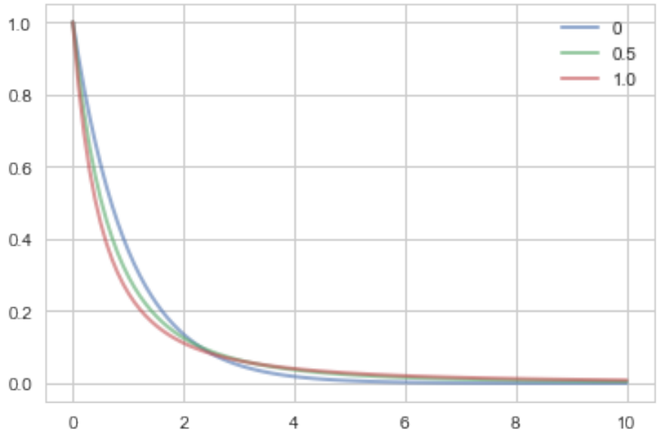
These w_s may have large or infinite variance.

Use PSIS: fit shape k Pareto to M largest w_s and replace them by expected values of corresponding order statistics under the pareto. Also truncate all weights at raw maximum w_s. Use joint as pareto cares not about multiplying factors.



M empirically set as $min(S/5, 3\sqrt{S})$.

Result from extreme value theory (Pickands–Balkema–de Haan theorem): conditional excess distribution function is a generalized pareto



$k < 0.5$ great, ok between 0.5 and 0.7, not so good after 0.7, weights too large.

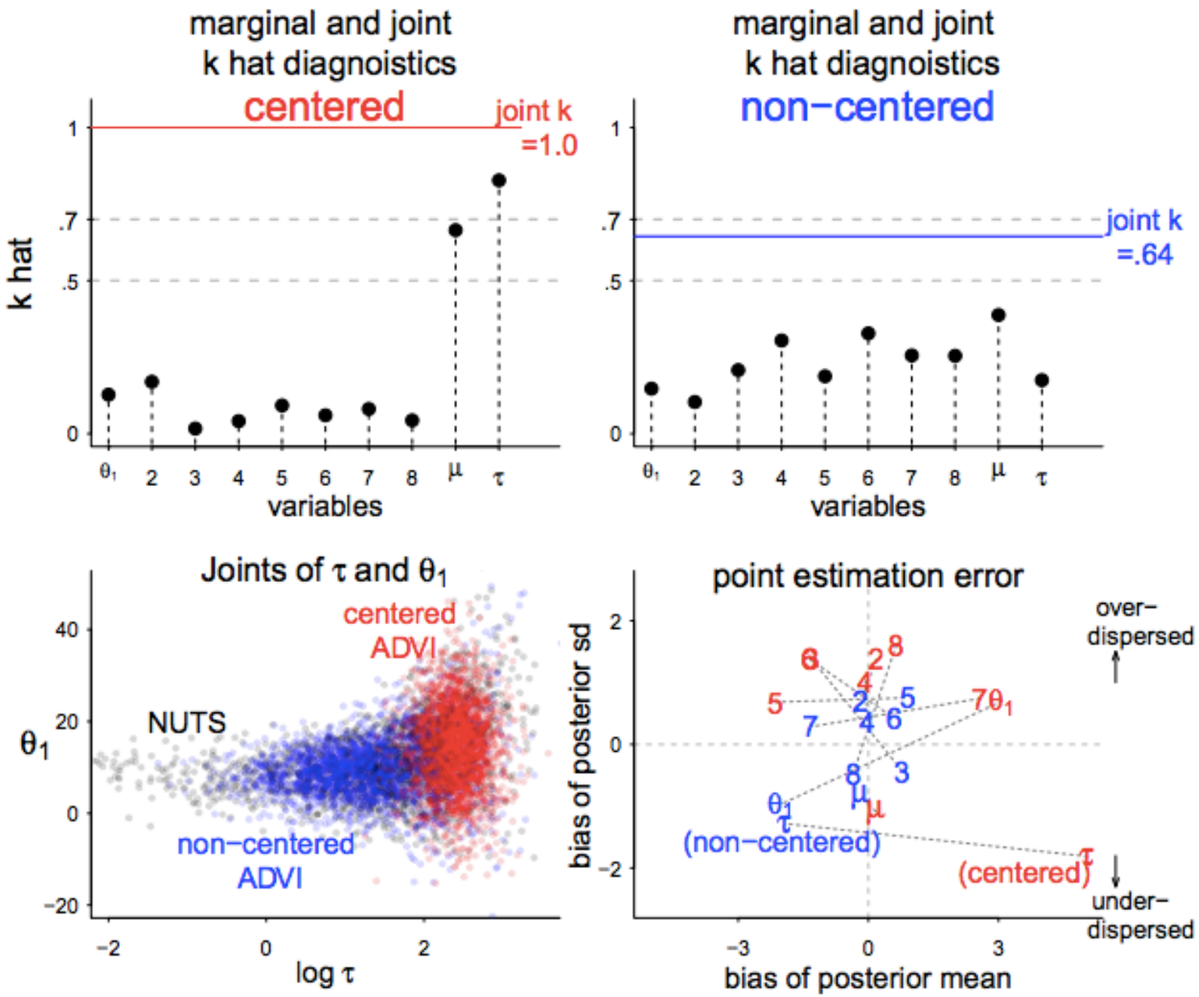
source

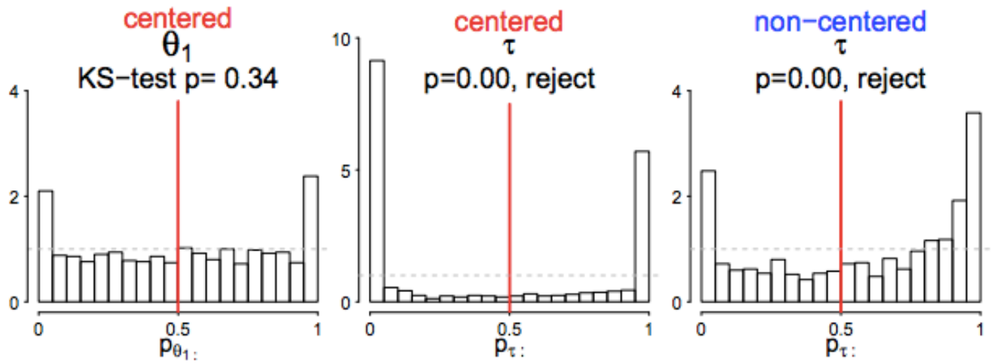
VSBC

- basic idea from bayesian workflow, posterior from data simulated from prior ($\theta_0 \sim p(\theta)$) should look like the prior. That is, ideally order statistics uniform
- in VSBC fit the posterior variationally. Will have some mismatch
- quantify mismatch by asymmetry in histogram of ith marginal callibration probabilities $p_{ij} = P_q(\theta_i < [\theta_j^0]_i)$

Left: ADVI posterior and pareto shape statistics

Below: VSBC histogram





Mixture Models

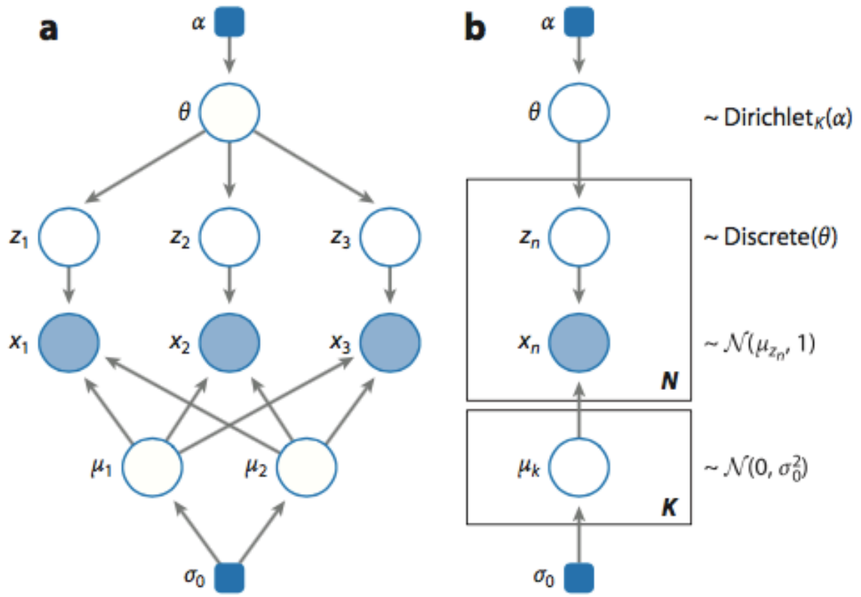


Figure 3

(*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?

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

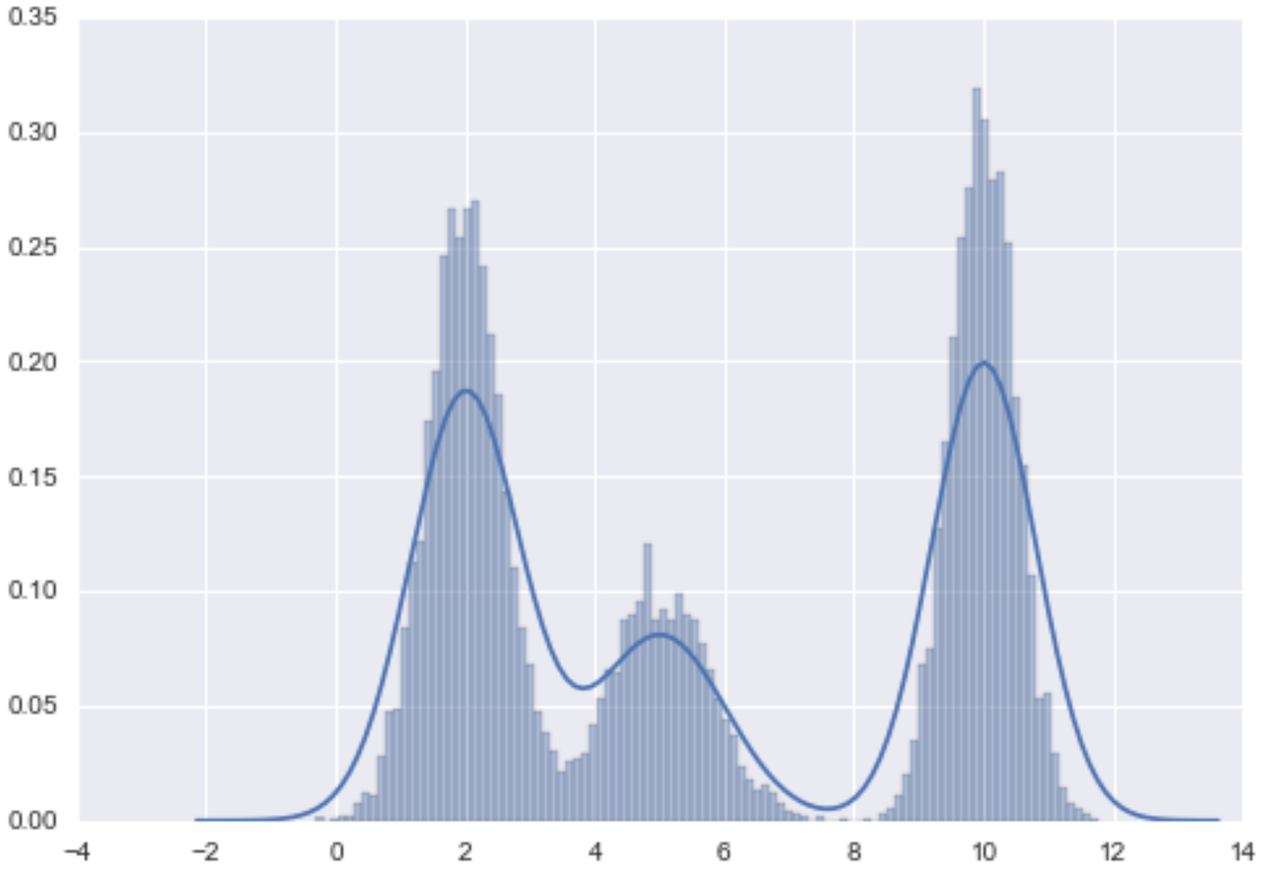
where Z says which component X is drawn from.

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



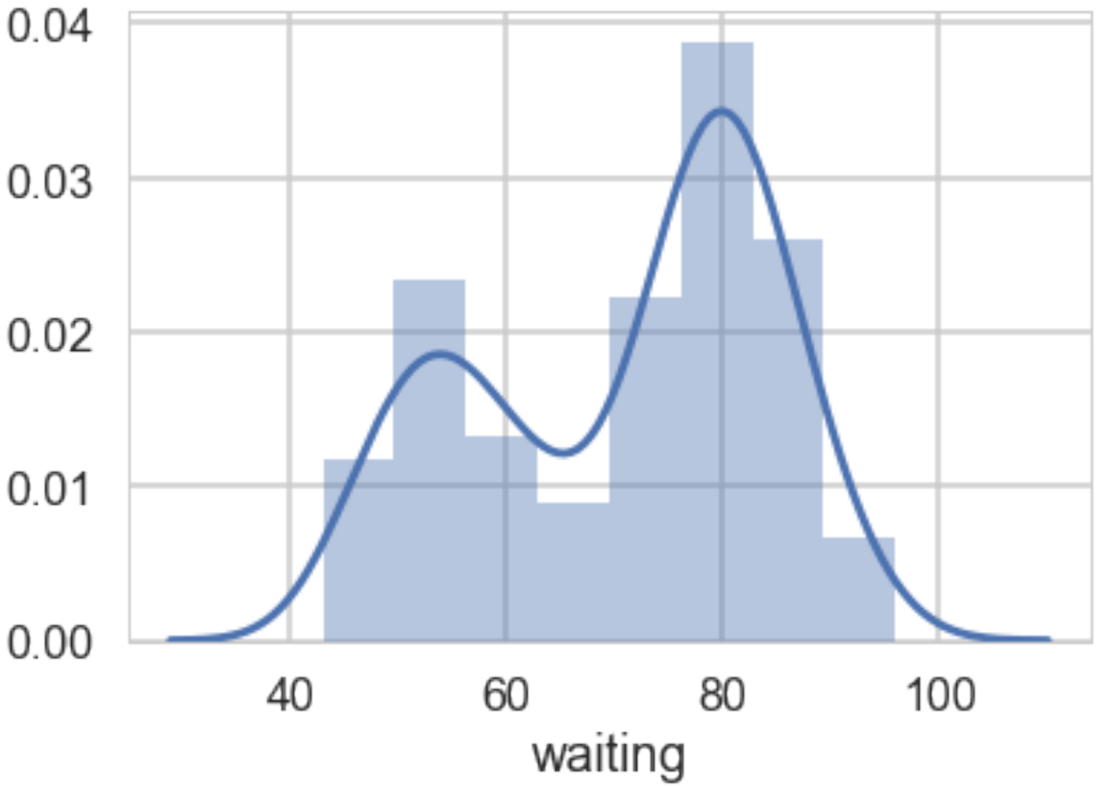
$$p(x|\{\theta_k\}) = \sum_k \lambda_k N(x|\mu_k, \Sigma_k)$$

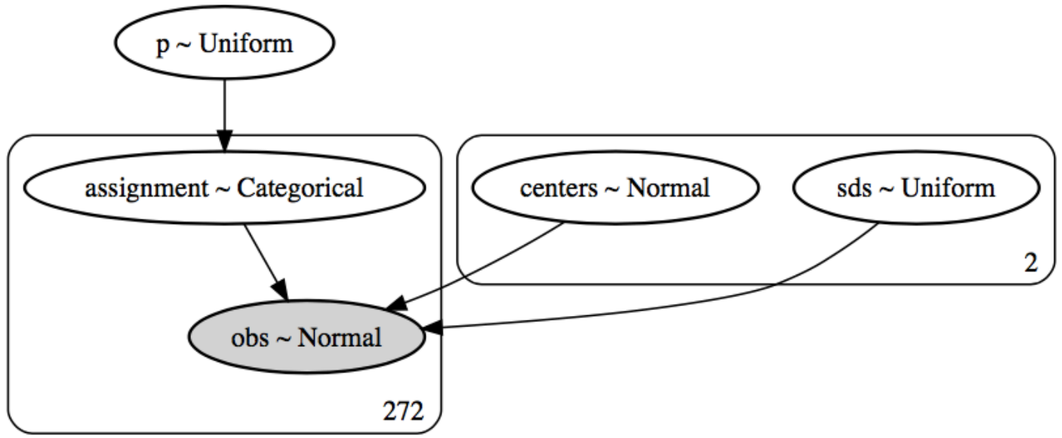
Generative:

```
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]])
```

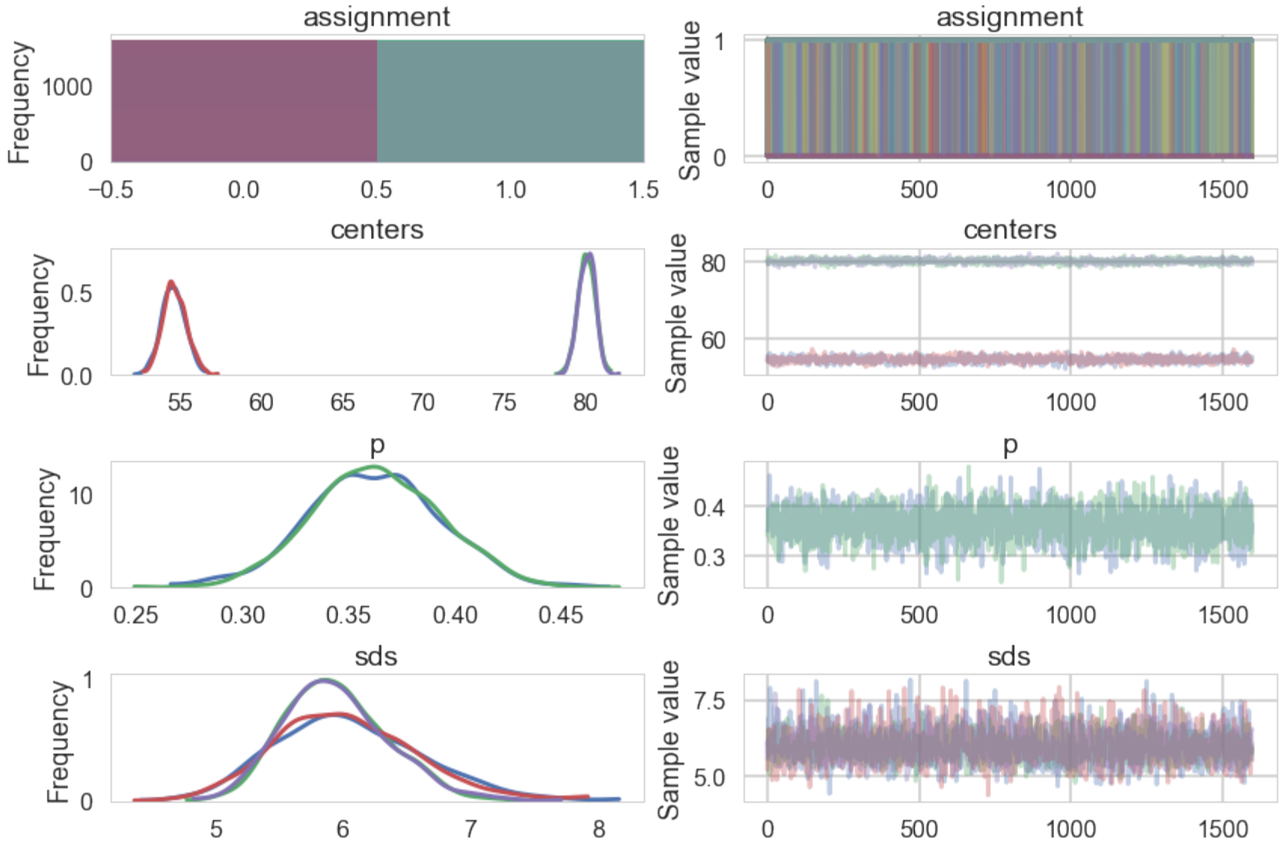
Old faithful Geyser



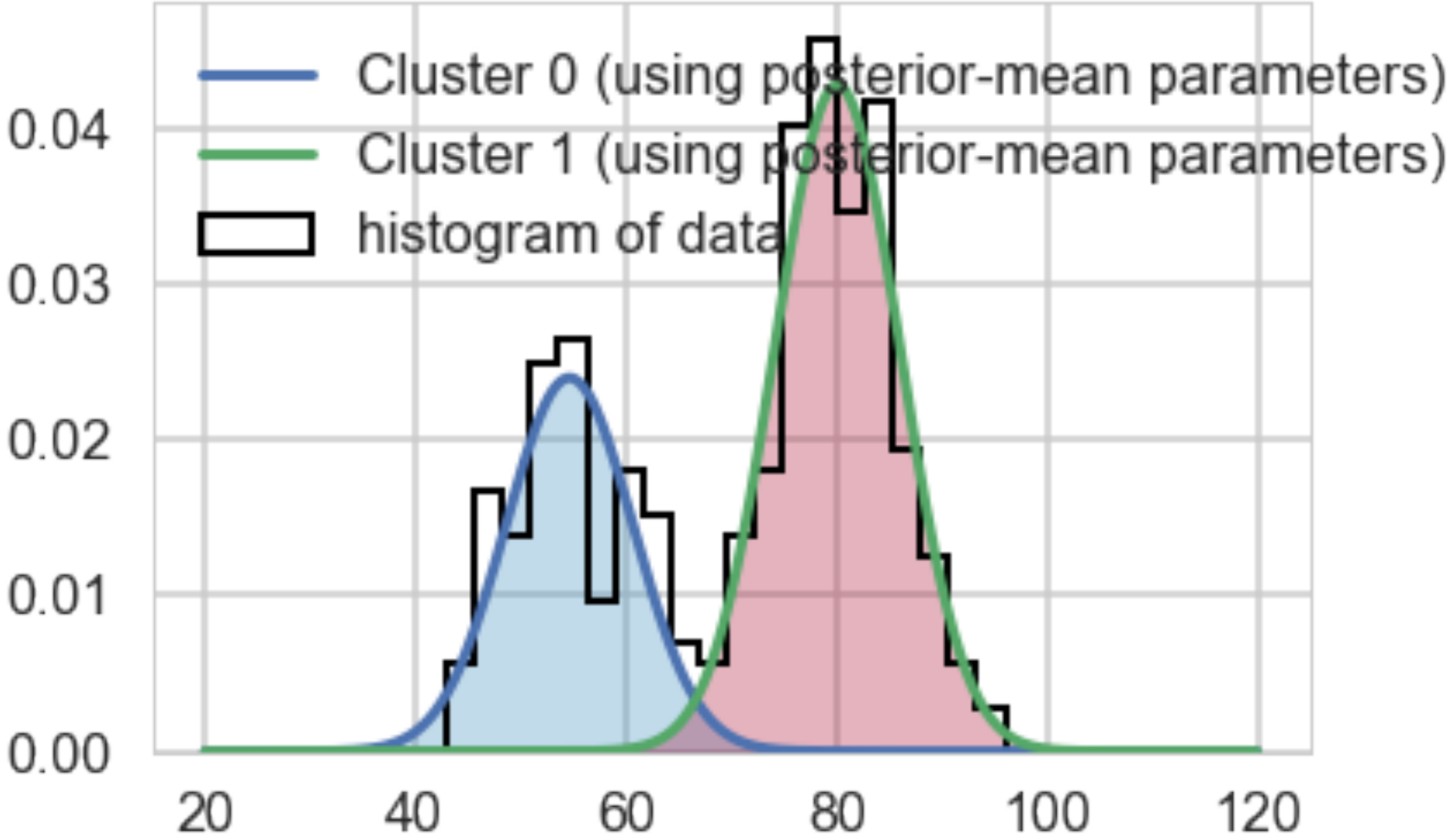


Sampling mixture models: 2 Gaussians

```
with pm.Model() as ofmodel:
    p1 = pm.Uniform('p', 0, 1)
    p2 = 1 - p1
    p = tt.stack([p1, p2])
    assignment = pm.Categorical("assignment", p,
                        shape=ofdata.shape[0])
    sds = pm.Uniform("sds", 0, 40, shape=2)
    centers = pm.Normal("centers",
            mu=np.array([50, 80]),
            sd=np.array([20, 20]),
            shape=2)

    observations = pm.Normal("obs",
        mu=centers[assignment],
        sd=sds[assignment],
        observed=ofdata.waiting)
```



Visualizing Clusters using posterior-mean parameters



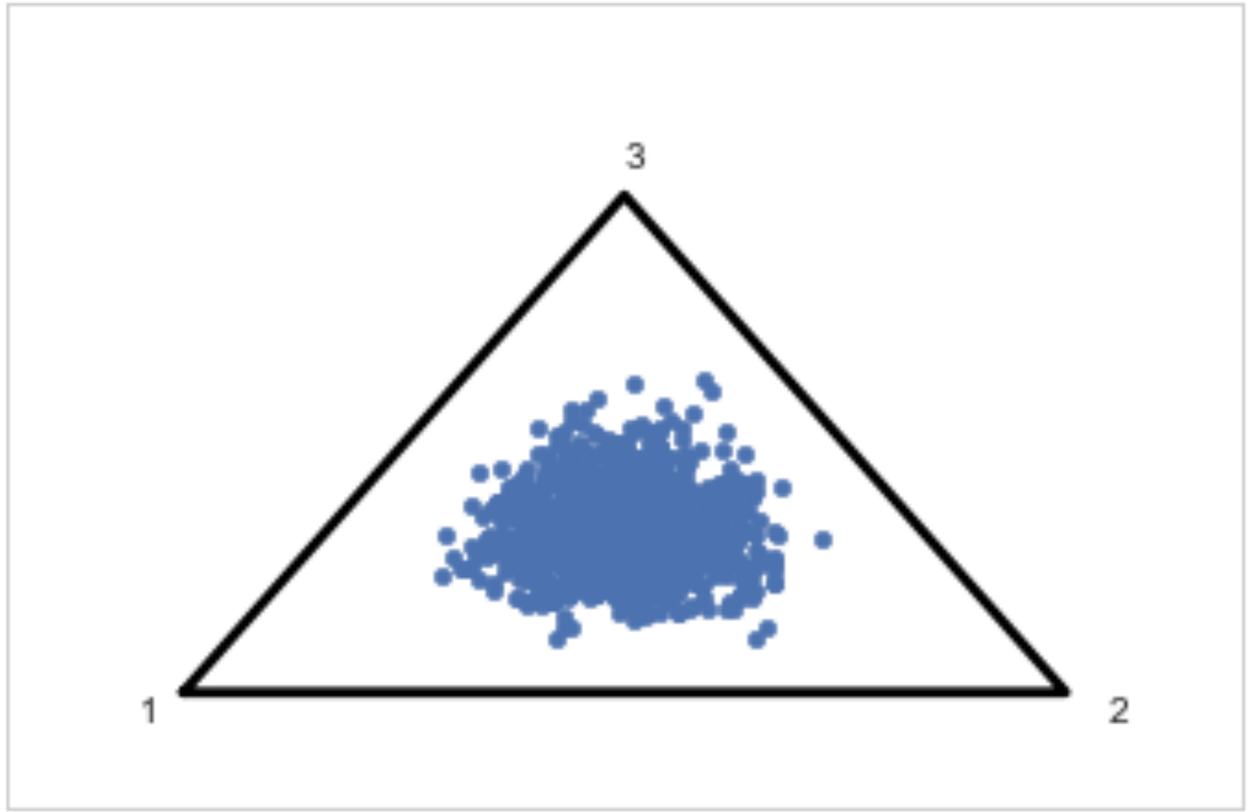
```
with pm.Model() as classmodel1:
    p1 = pm.Uniform('p', 0, 1)
    p2 = 1 - p1
    p = tt.stack([p1, p2])
    #Notice NO "observed" below
    assignment_tr = pm.Categorical("assignment_tr", p)
    sds = pm.Uniform("sds", 0, 100, shape=2)
    centers = pm.Normal("centers",
                        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)
```

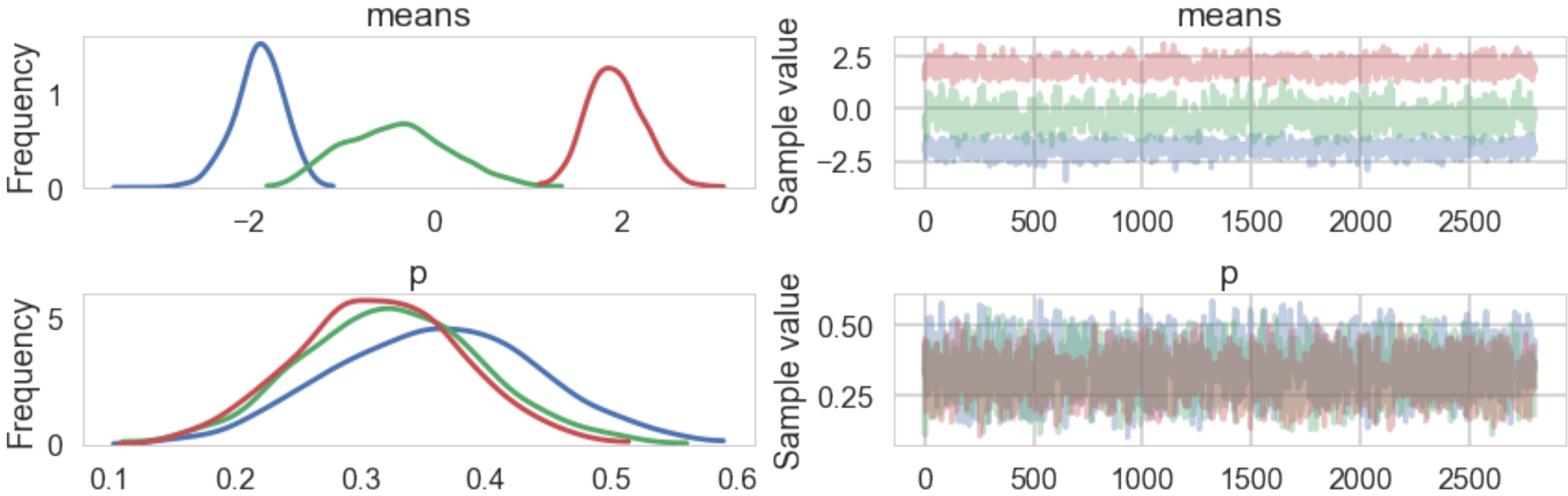
Sampling Mixture Models

- very very hard
- samplers can get stuck in a mode, possibly multimodal posteriors
- non-identifiability due to label switching
- most people use EM or Variational Inference
- can use explicit marginalization to make it easier, see lab

3 close by gaussians

```
with pm.Model() as mofb:
    p = pm.Dirichlet('p',
        a=np.array([10., 10., 10.]), shape=3)
    # ensure all clusters have some points
    p_min_potential = pm.Potential('p_min_potential',
        tt.switch(tt.min(p) < .1, -np.inf, 0))
    # cluster centers
    means = pm.Normal('means', mu=0, sd=10, shape=3,
        transform=tr.ordered,
        testval=np.array([-1, 0, 1]))

    category = pm.Categorical('category',
        p=p,
        shape=data.shape[0])

    # likelihood for each observed value
    points = pm.Normal('obs',
        mu=means[category],
        sd=1., #sds[category],
        observed=data)
```






```
Multiprocess sampling (2 chains in 2 jobs)
CompoundStep
>NUTS: [means, p]
>CategoricalGibbsMetropolis: [category]
Sampling 2 chains: 100%|██████████| 21000/21000 [06:13<00:00, 56.23draws/s]
There were 10 divergences after tuning. Increase `target_accept` or reparameterize.
There were 7 divergences after tuning. Increase `target_accept` or reparameterize.
The number of effective samples is smaller than 10% for some parameters.
```

The log-sum-exp trick and mixtures

Suppose you want to calculate $log\_sum\_exp(a, b) = log(exp(a) + exp(b))$.

For numerical stability, wecan write this as:
$log\left(exp(a) + exp(b)\right) = c + log\left[exp(a - c) + exp(b - c)\right],$

where $c = max(a, b)$. Then one of $a - c$ or $b - c$ is zero and the other is negative.

In pymc3, from https://github.com/pymc-devs/pymc3/blob/master/pymc3/math.py

```
def logsumexp(x, axis=None):
    # Adapted from https://github.com/Theano/Theano/issues/1563
    x_max = tt.max(x, axis=axis, keepdims=True)
    return tt.log(tt.sum(tt.exp(x - x_max), axis=axis, keepdims=True)) + x_max
```

Why? Marginalizing over discretes.

For example (as taken from the Stan Manual), the mixture of $N(-1, 2)$ and $N(3, 1)$ with mixing proportion $\lambda = (0.3, 0.7)$:

$$log p(y|\lambda, \mu, \sigma) = log\left[0.3 \times N(y|-1, 2) + 0.7 \times N(y|3, 1)\right]$$

$$= log\left[exp(log(0.3 \times N(y|-1, 2))) + exp(log(0.7 \times N(y|3, 1)))\right]$$

$$= \texttt{log\_sum\_exp}\left(log(0.3) + log\, N(y|-1, 2),\ \ log(0.7) + log\, N(y|3, 1)\right).$$

If we do this, we can go directly from the Dirichlet priors for p and forget the `category` variable

pymc3 does this for us

```
import pymc3.distributions.transforms as tr
with pm.Model() as mof3:
    p = pm.Dirichlet('p', a=np.array([10., 10., 10.]), shape=3)
    means = pm.Normal('means', mu=0, sd=10, shape=3, transform=tr.ordered,
                      testval=np.array([-1, 0, 1]))


    points = pm.NormalMixture('obs', p, mu=means, sd=1, observed=data)
```

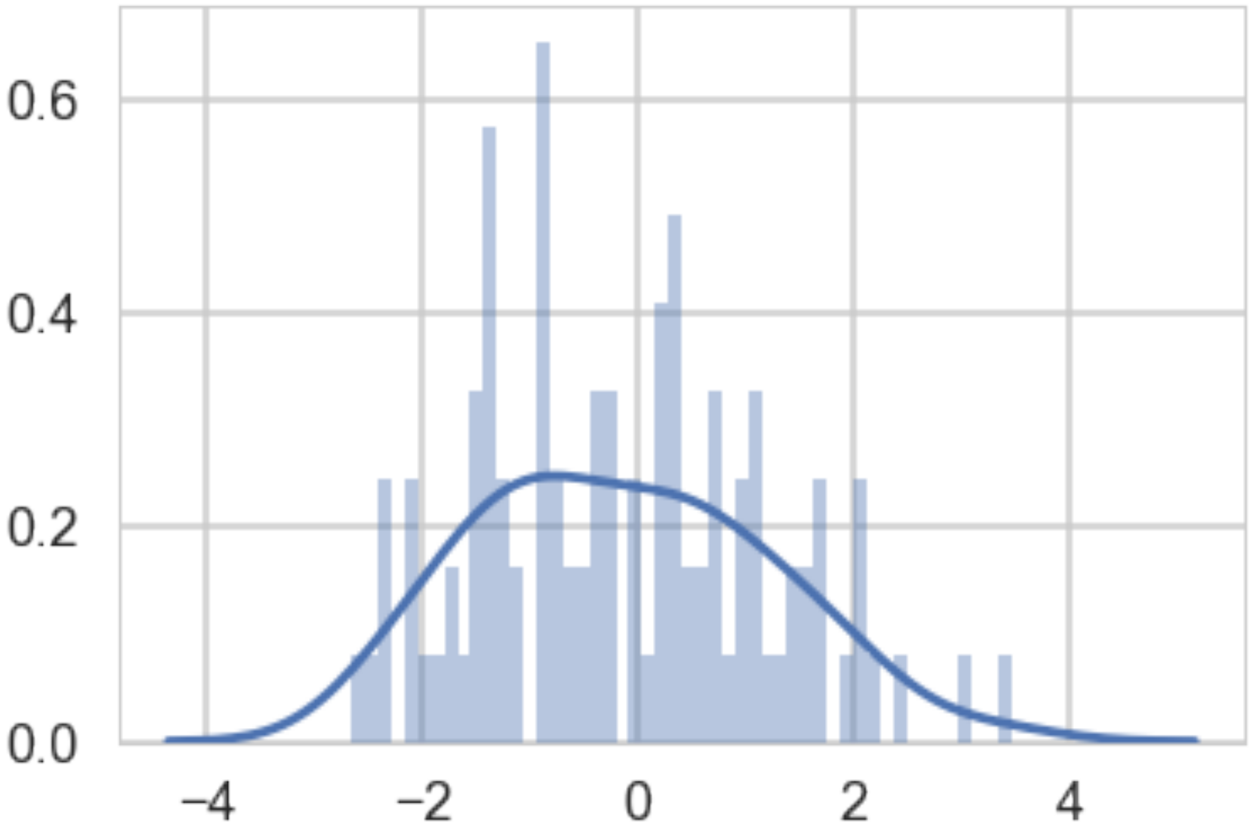
By Hand

```
def logp_normal(mu, sigma, value):
    # log probability of individual samples
    delta = lambda mu: value - mu
    return (-1 / 2.) * (tt.log(2 * np.pi) + tt.log(sigma*sigma) +
                        (delta(mu)* delta(mu))/(sigma*sigma))

# Log likelihood of Gaussian mixture distribution
def logp_gmix(mus, pis, sigmas, n_samples, n_components):

    def logp_(value):
        logps = [tt.log(pis[i]) + logp_normal(means[i], sigmas[i], value)
                 for i in range(n_components)]

        return tt.sum(logsumexp(tt.stacklists(logps)[:, :n_samples], axis=0))

    return logp_
```

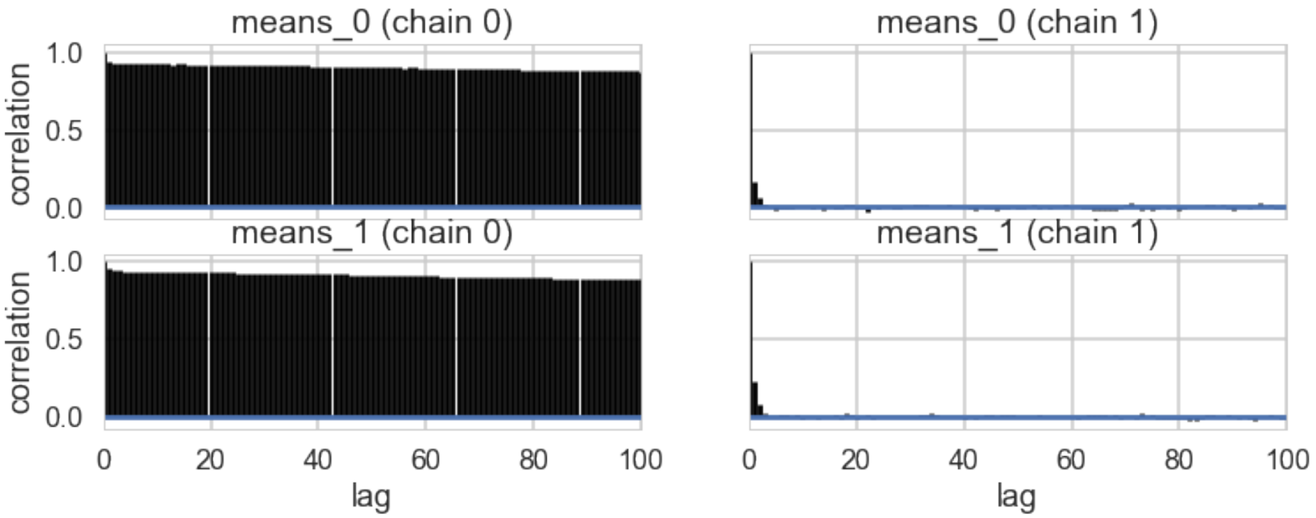
```
with pm.Model() as mof2:
    p = pm.Dirichlet('p',
        a=np.array([10., 10., 10.]), shape=3)

    # cluster centers
    means = pm.Normal('means', mu=0, sd=10,
        shape=3, transform=tr.ordered,
        testval=np.array([-1, 0, 1]))

    sds = [1., 1., 1.]
    # likelihood for each observed value
    points = pm.DensityDist('obs', logp_gmix(means, p, sds, data.shape[0], 3),
                        observed=data)
```

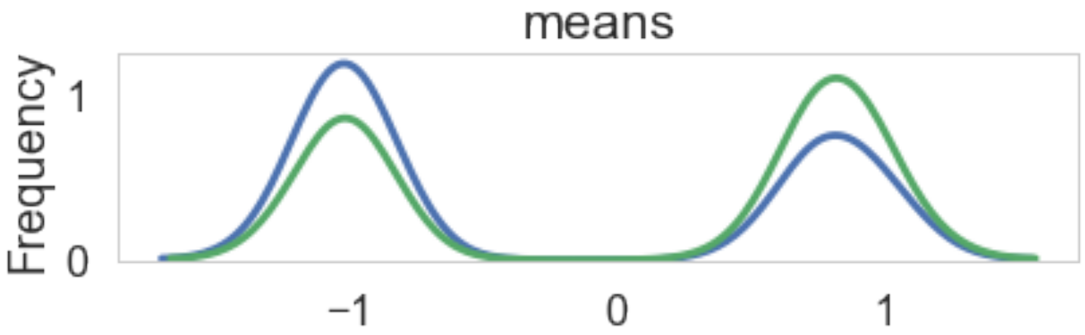
Now we can use NUTS or ADVI as no discrete parameters are left in the problem

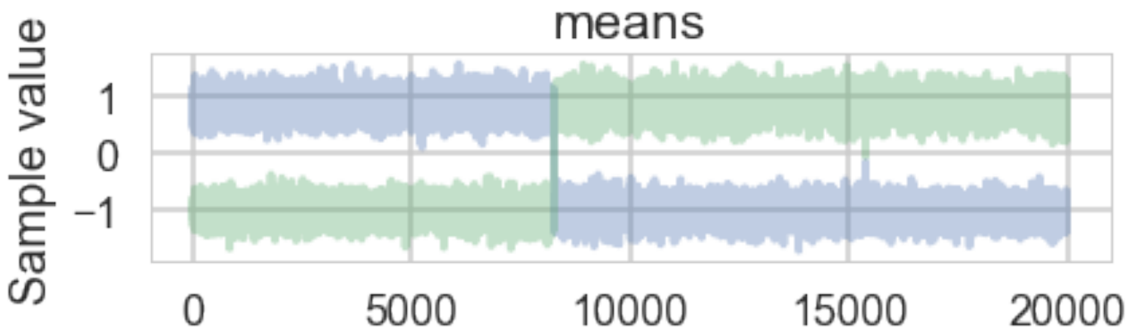
Sampling mixture models: 2 close Gaussians



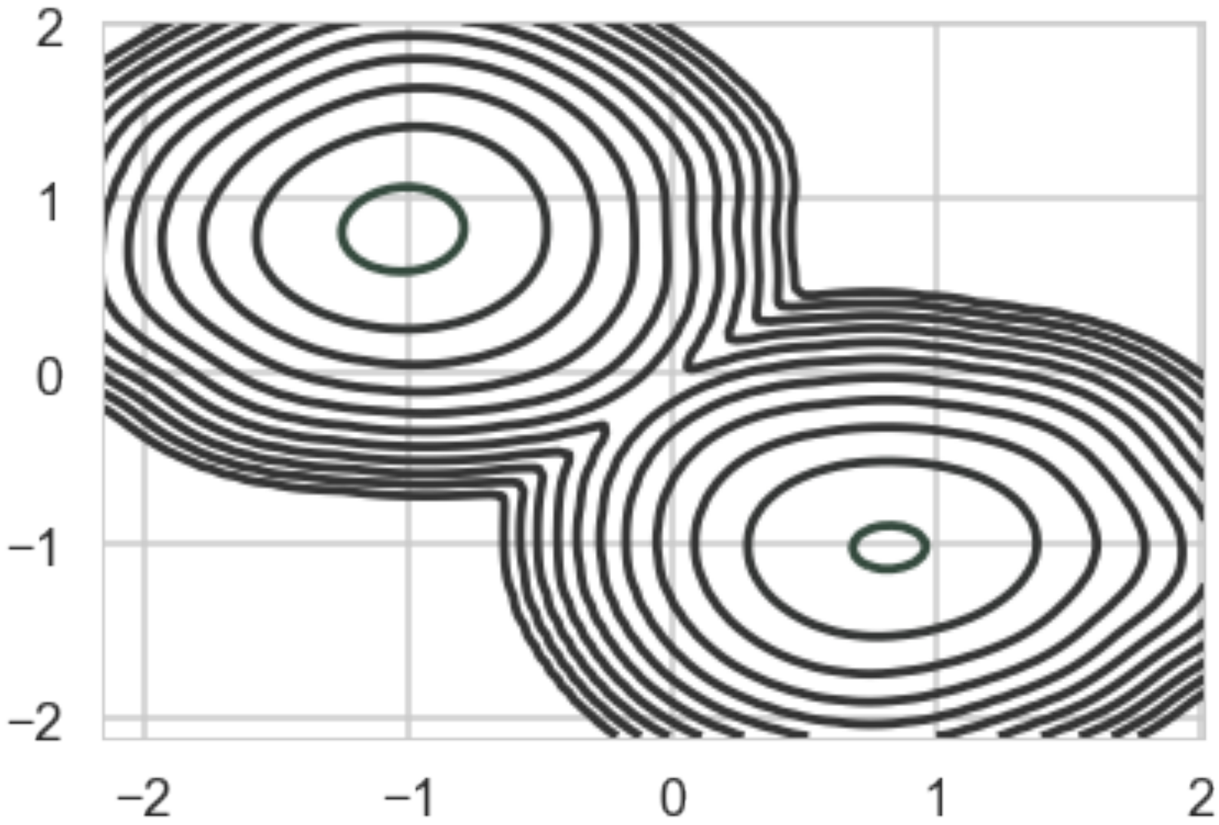

```
with pm.Model() as model1:
    p = [1/2, 1/2]
    means = pm.Normal('means', mu=0, sd=10, shape=2)
    points = pm.NormalMixture('obs', p, mu=means,
        sd=1, observed=data)
```


means_0 (chain 0)
means_0 (chain 1)
means_1 (chain 0)
means_1 (chain 1)
correlation
1.0
0.5
0.0
0
20
40
60
80
100
lag


means
Frequency
1
0
−1
0
1

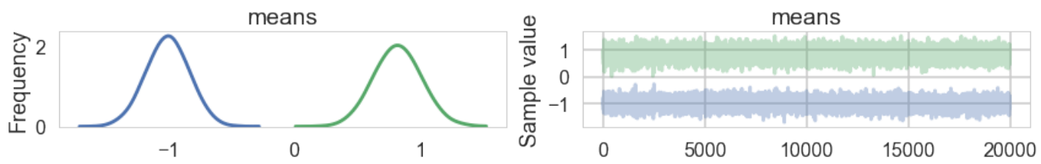

means
Sample value
1
0
−1
0
5000
10000
15000
20000

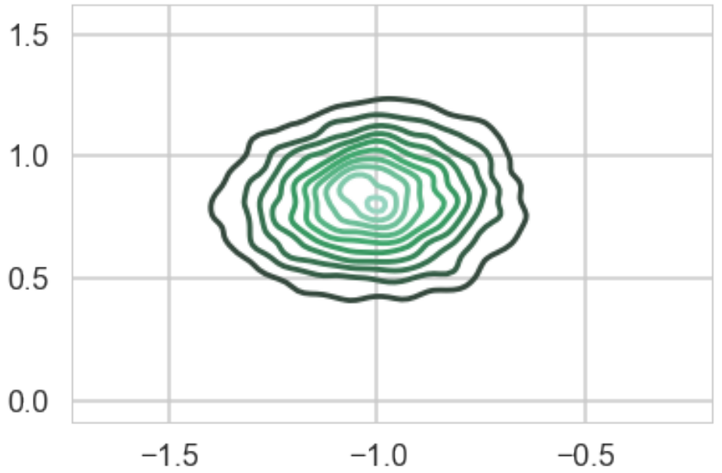
Multi-modal posterior



Fix by ordering

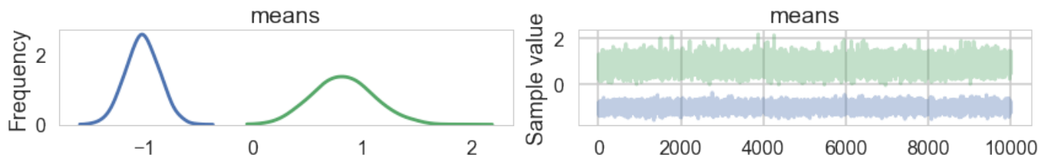
```
import pymc3.distributions.transforms as tr
with pm.Model() as model2:
    p = [1/2, 1/2]

    means = pm.Normal('means', mu=0, sd=10,
        shape=2, transform=tr.ordered,
                  testval=np.array([-1, 1]))
    points = pm.NormalMixture('obs', p, mu=means,
        sd=1, observed=data)
```

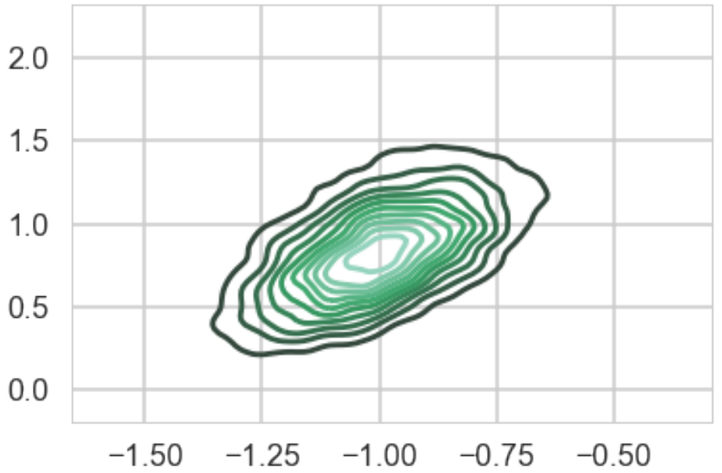




ADVI

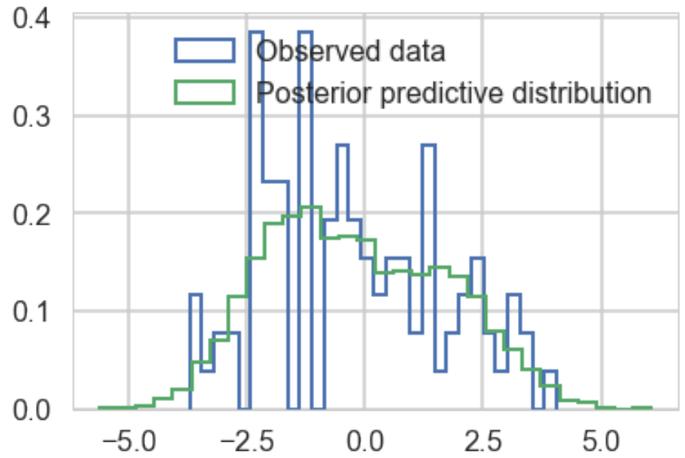
```
advi2 = pm.ADVI(model=model2)
advi2.fit(n=15000)
samps2=advi2.approx.sample(10000)
```

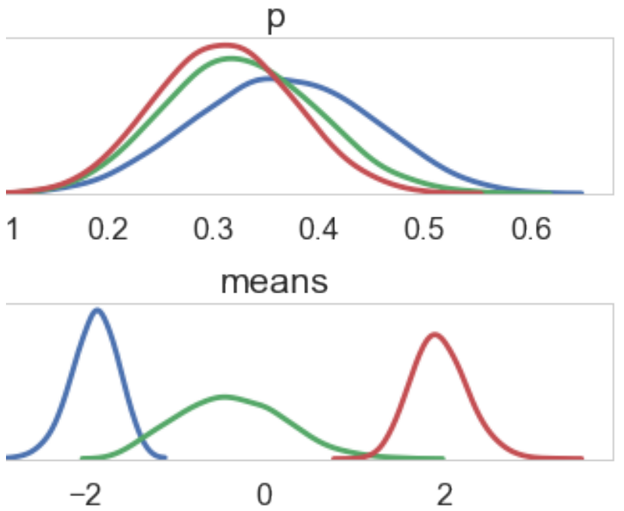


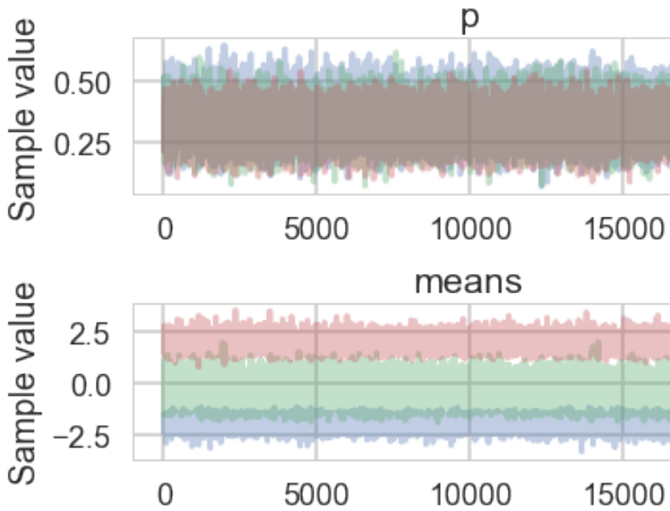


Back to the 3 gaussians

```
import pymc3.distributions.transforms as tr
with pm.Model() as mof3:
    p = pm.Dirichlet('p',
        a=np.array([10., 10., 10.]), shape=3)
    means = pm.Normal('means', mu=0, s
        d=10, shape=3, transform=tr.ordered,
        testval=np.array([-1, 0, 1]))
    points = pm.NormalMixture('obs', p,
        mu=means, sd=1, observed=data)
```







And with ADVI