Probabilistic Programming

Marius Popescu

popescunmarius@gmail.com

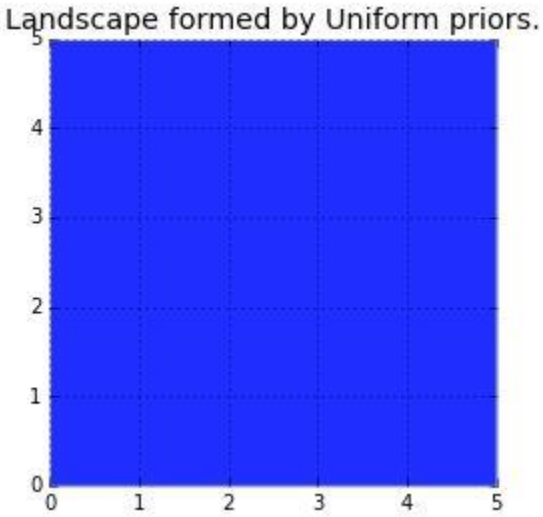
2020 - 2021

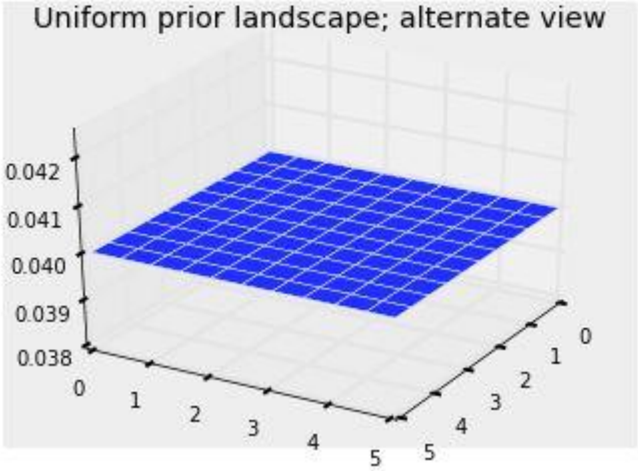
Fitting Models

Markov Chain Monte Carlo (MCMC)

The Bayesian Landscape

When we setup a Bayesian inference problem with N unknowns, we are implicitly creating an N dimensional space for the prior distributions to exist in. Associated with the space is an additional dimension, which we can describe as the surface, or curve, that sits on top of the space, that reflects the prior probability of a particular point. The surface on the space is defined by our prior distributions. For example, if we have two unknowns p_1 and p_2, and priors for both are $\text{Uniform}(0,5)$, the space created is a square of length 5 and the surface is a flat plane that sits on top of the square (representing that every point is equally likely):

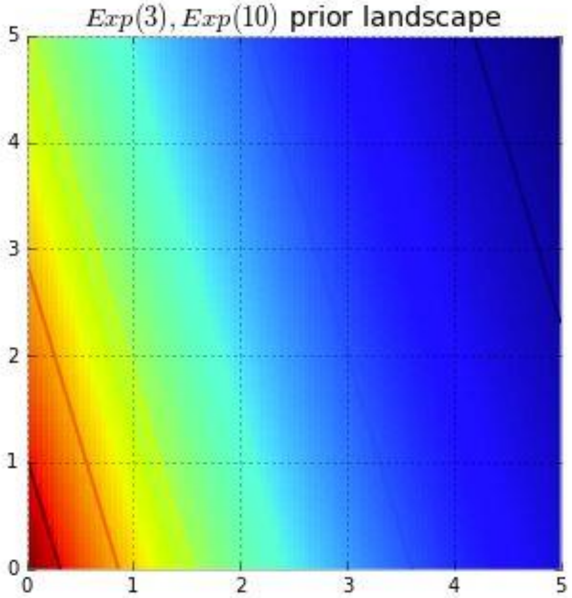


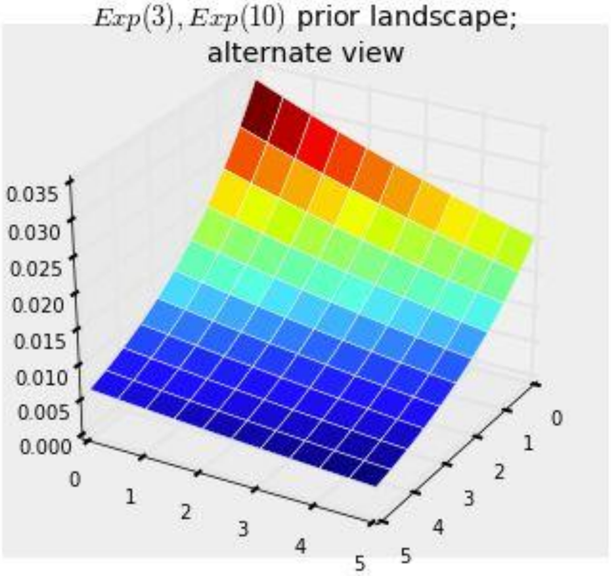


The Bayesian Landscape

Alternatively, if the two priors are $\text{Exp}(3)$ and $\text{Exp}(10)$, then the space is all positive numbers on the 2-D plane, and the surface induced by the priors looks like a water fall that starts at the point (0,0) and flows over the positive numbers.

The plots below visualize this. The more dark red the color, the more prior probability is assigned to that location. Conversely, areas with darker blue represent that our priors assign very low probability to that location:





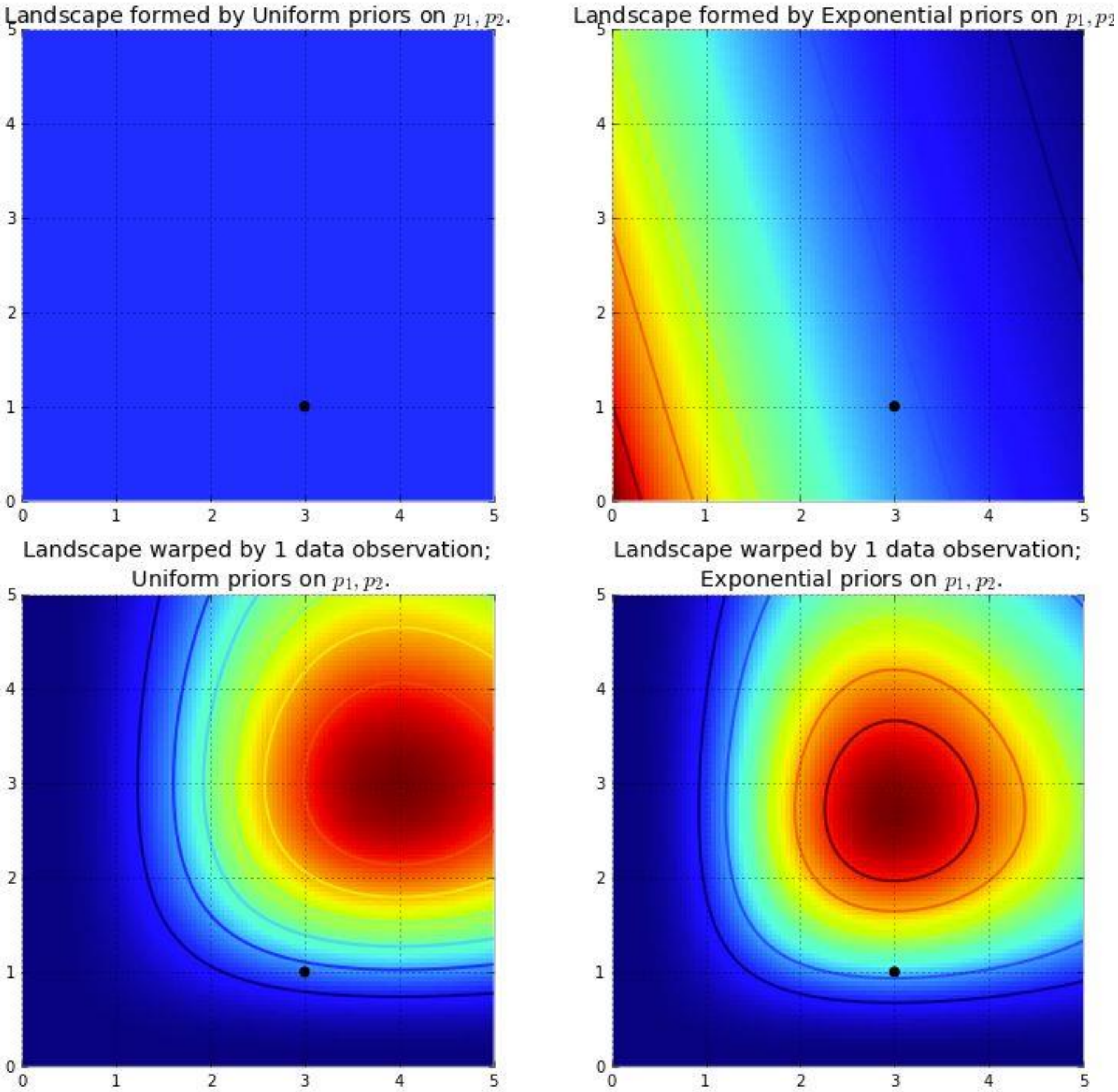
The Bayesian Landscape

If these surfaces describe our *prior distributions* on the unknowns, what happens to our space after we incorporate our observed data X? The data X does not change the space, but it changes the surface of the space by *pulling and stretching the fabric of the prior surface* to reflect where the true parameters likely live. More data means more pulling and stretching, and our original shape becomes mangled or insignificant compared to the newly formed shape. Less data, and our original shape is more present. Regardless, the resulting surface describes the *posterior distribution*.

For two dimensions, the data essentially *pushes up* the original surface to make *tall mountains*. The tendency of the observed data to *push up* the posterior probability in certain areas is checked by the prior probability distribution, so that lower prior probability means more resistance. Thus in the double-exponential prior case above, a mountain (or multiple mountains) that might erupt near the (0,0) corner would be much higher than mountains that erupt closer to (5,5), since there is more resistance (low prior probability) near (5,5). The peak reflects the posterior probability of where the true parameters are likely to be found. Importantly, if the prior has assigned a probability of 0, then no posterior probability will be assigned there.

The Bayesian Landscape

Suppose the priors mentioned above represent different parameters λ of two Poisson distributions. We observe a few data points and visualize the new landscape:



Landscape formed by Uniform priors on p_1, p_2.

Landscape formed by Exponential priors on p_1, p_2

Landscape warped by 1 data observation; Uniform priors on p_1, p_2.

Landscape warped by 1 data observation; Exponential priors on p_1, p_2.

Exploring the Landscape using the MCMC

We should explore the deformed posterior space generated by our prior surface and observed data to find the posterior mountain.

The idea behind MCMC is to perform an intelligent search of the space. To say "search" implies we are looking for a particular point, which is perhaps not an accurate as we are really looking for a broad mountain.

MCMC returns samples from the posterior distribution, not the distribution itself. Stretching our mountainous analogy to its limit, MCMC performs a task similar to repeatedly asking "How likely is this pebble I found to be from the mountain I am searching for?", and completes its task by returning thousands of accepted pebbles in hopes of reconstructing the original mountain

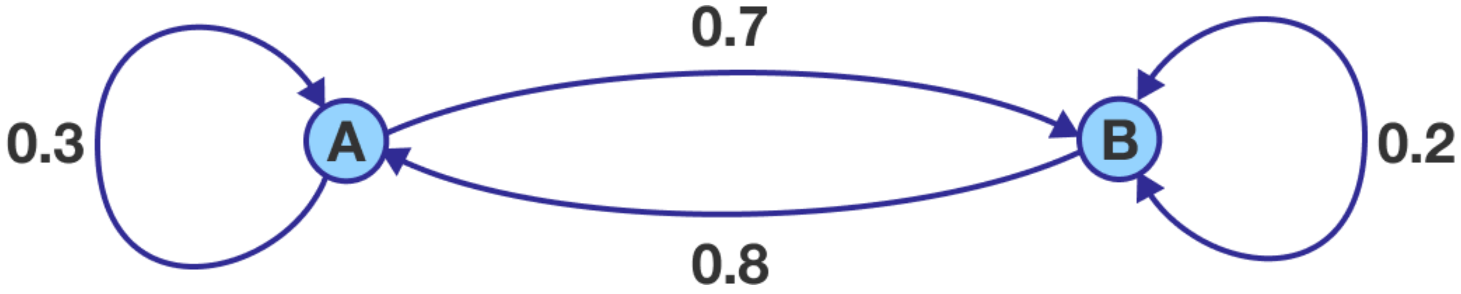
Exploring the Landscape using the MCMC

1. Start at current position.
2. Propose moving to a new position (investigate a pebble near you).
3. Accept/Reject the new position based on the position's adherence to the data and prior distributions (ask if the pebble likely came from the mountain).
4. If you accept: Move to the new position. Return to Step 1.

   Else: Do not move to new position. Return to Step 1.

1. After a large number of iterations, return all accepted positions.

Markov Chains



$X_t \in \{\mathrm{A, B}\}$, $\{\mathrm{A, B}\}$ state space (A, B states)

π_0 the distribution of X_0 (initial distribution of states) $\pi_0 = \begin{pmatrix} \mathrm{A} & \mathrm{B} \\ 0.5 & 0.5 \end{pmatrix}$

$$P = \begin{array}{c|cc} & \mathrm{A} & \mathrm{B} \\ \hline \mathrm{A} & 0.3 & 0.7 \\ \mathrm{B} & 0.8 & 0.2 \end{array} \quad \text{transition matrix}$$

$$\pi_1 = \pi_0 P = (0.5\ \ 0.5)\begin{pmatrix} 0.3 & 0.7 \\ 0.8 & 0.2 \end{pmatrix} = (0.55\ \ 0.45), \quad \pi_2 = \pi_1 P, \quad \ldots$$

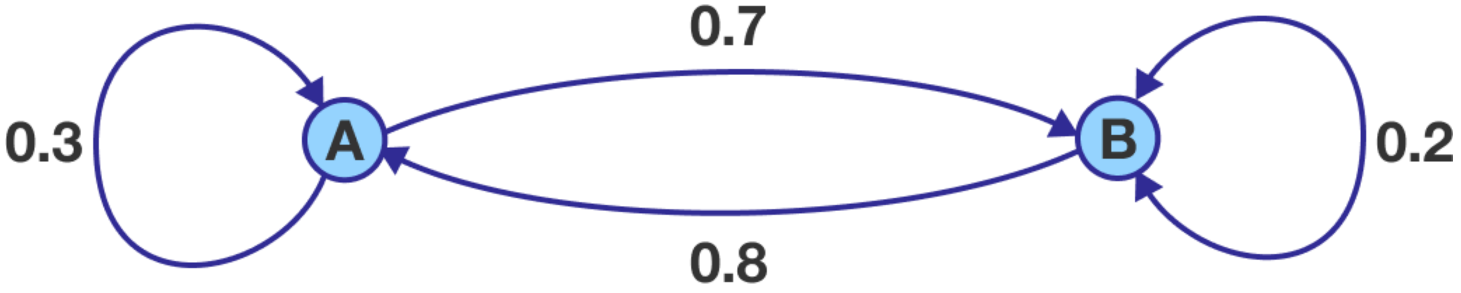
A Markov chain is a special type of *stochastic process*. The standard definition of a stochastic process is an ordered collection of random variables: $\{X_t : t \in T\}$ where t is frequently (but not necessarily) a time index.

If we think of X_t as a state X at time t, and invoke the following dependence condition on each state:

$$P(X_{t+1} = x_{t+1} | X_t = x_t, X_{t-1} = x_{t-1}, \ldots, X_0 = x_0) = P(X_{t+1} = x_{t+1} | X_t = x_t)$$

then the stochastic process is known as a Markov chain. This conditioning specifies that the future depends on the current state, but not past states. Thus, the Markov chain wanders about the state space, remembering only where it has just been in the last time step. The collection of transition probabilities is sometimes called a *transition matrix* when dealing with discrete states, or more generally, a *transition kernel*.

Markov Chains



$$\pi = \begin{pmatrix} \frac{8}{15} & \frac{7}{15} \end{pmatrix}$$

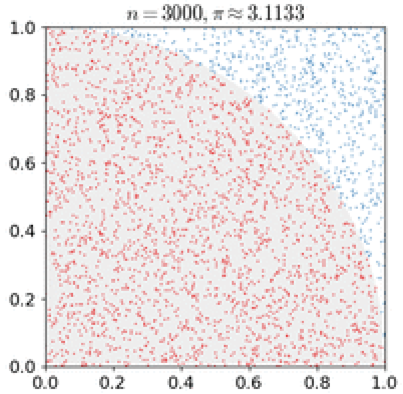
$$\begin{pmatrix} \frac{8}{15} & \frac{7}{15} \end{pmatrix} \begin{pmatrix} \frac{3}{10} & \frac{7}{10} \\ \frac{8}{10} & \frac{2}{10} \end{pmatrix} = \begin{pmatrix} \frac{8}{15} & \frac{7}{15} \end{pmatrix}$$

A *stationary distribution* of a Markov chain is a probability distribution that remains unchanged in the Markov chain as time progresses:

$$\pi = \pi P$$

Ergodic Markov chains have a unique stationary distribution

Monte Carlo Method



Monte Carlo method applied to approximating the value of π. After placing 30,000 random points, the estimate for π is within 0.07% of the actual value.

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Their essential idea is using randomness to solve problems that might be deterministic in principle

$$I = \int h(x) f(x) dx \quad f \text{ a probability density function}$$

If we can produce a reasonable number of random vectors x_i sampled from f we can use these values to approximate the unknown integral by finite sums:

$$\hat{I} = \frac{1}{n} \sum_{i=1}^{n} h(x_i)$$

By the strong law of large numbers: $\hat{I} \rightarrow I$ with probability 1

MCMC: A Class of Algorithms

- Design a Markov chain that has the desired distribution (posterior distribution) as its stationary distribution (step function / method)
- Simulate (Monte Carlo) the Markov chain a sufficient number of steps to reach its stationary distribution (burn-in)
- Simulate the Markov chain to sample from the desired distribution (posterior distribution)

The Metropolis-Hastings Algorithm

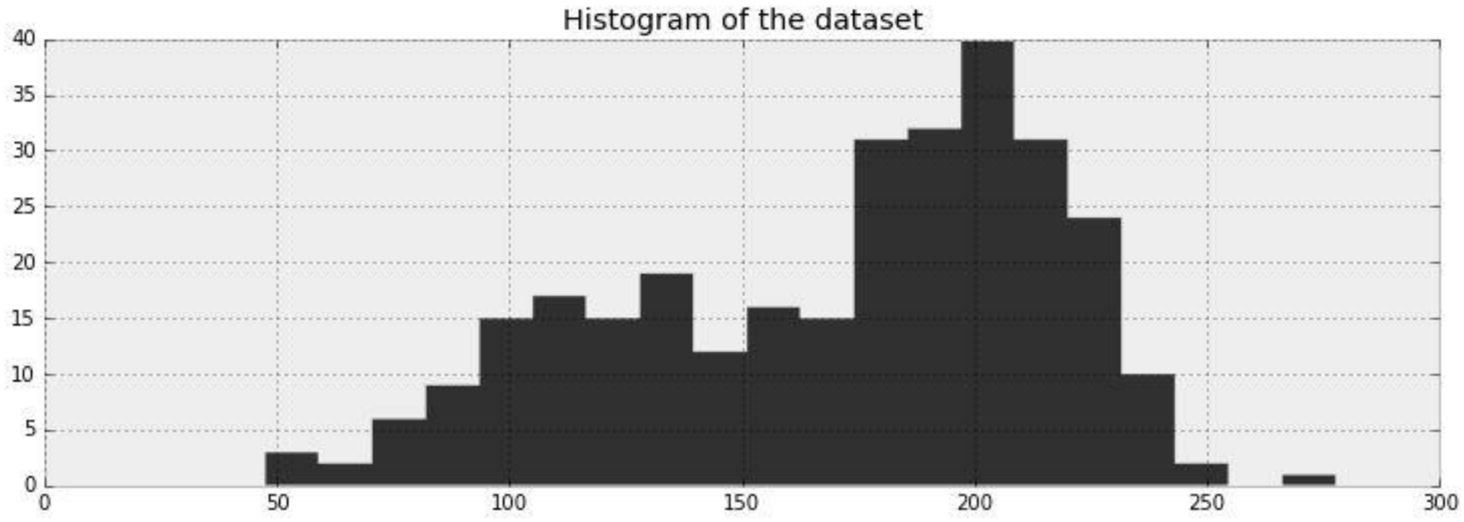
- The Metropolis-Hastings algorithm generates candidate state transitions from an alternate distribution, and accepts or rejects each candidate probabilistically.
- A proposal distribution $Q(\theta'|\theta)$ will be used to propose the next candidate θ' from the current state θ. The proposal distribution must be able to explore the entire space.
- The proposed parameter value θ' is accepted with some probability given by the acceptance probability $\alpha(\theta',\theta) = \min(1, \frac{\pi(\theta')Q(\theta|\theta')}{\pi(\theta)Q(\theta'|\theta)})$. If the proposal distribution is symmetric (as originally proposed by Metropolis) then: $\alpha(\theta',\theta) = \min(1, \frac{\pi(\theta')}{\pi(\theta)})$.
- The chain is generated from the proposed parameter θ' as follow

$$\theta^{t+1} = \begin{cases} \theta' & \text{if accepted; } \alpha(\theta',\theta) > u \;\; u \sim \text{Uniform}(0,1) \\ \theta^t & \text{otherwise} \end{cases}$$

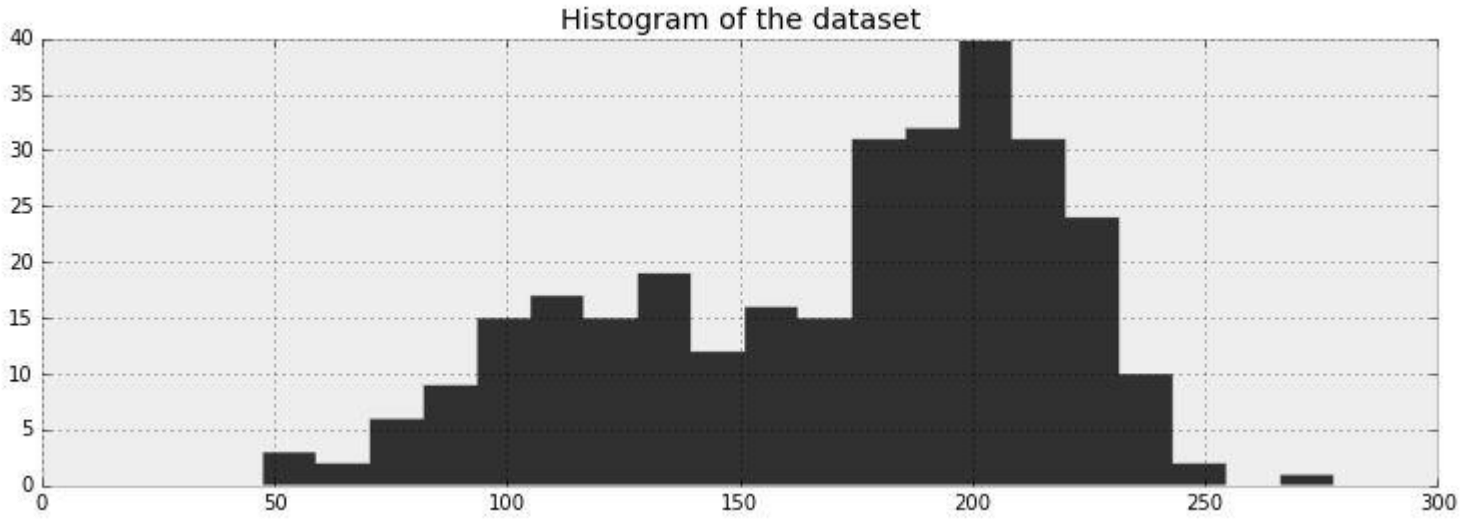
MCMC in PyMC

Example: Unsupervised Clustering using a Mixture Model

Suppose we are given the following dataset. What does the data suggest?



Mixture Model



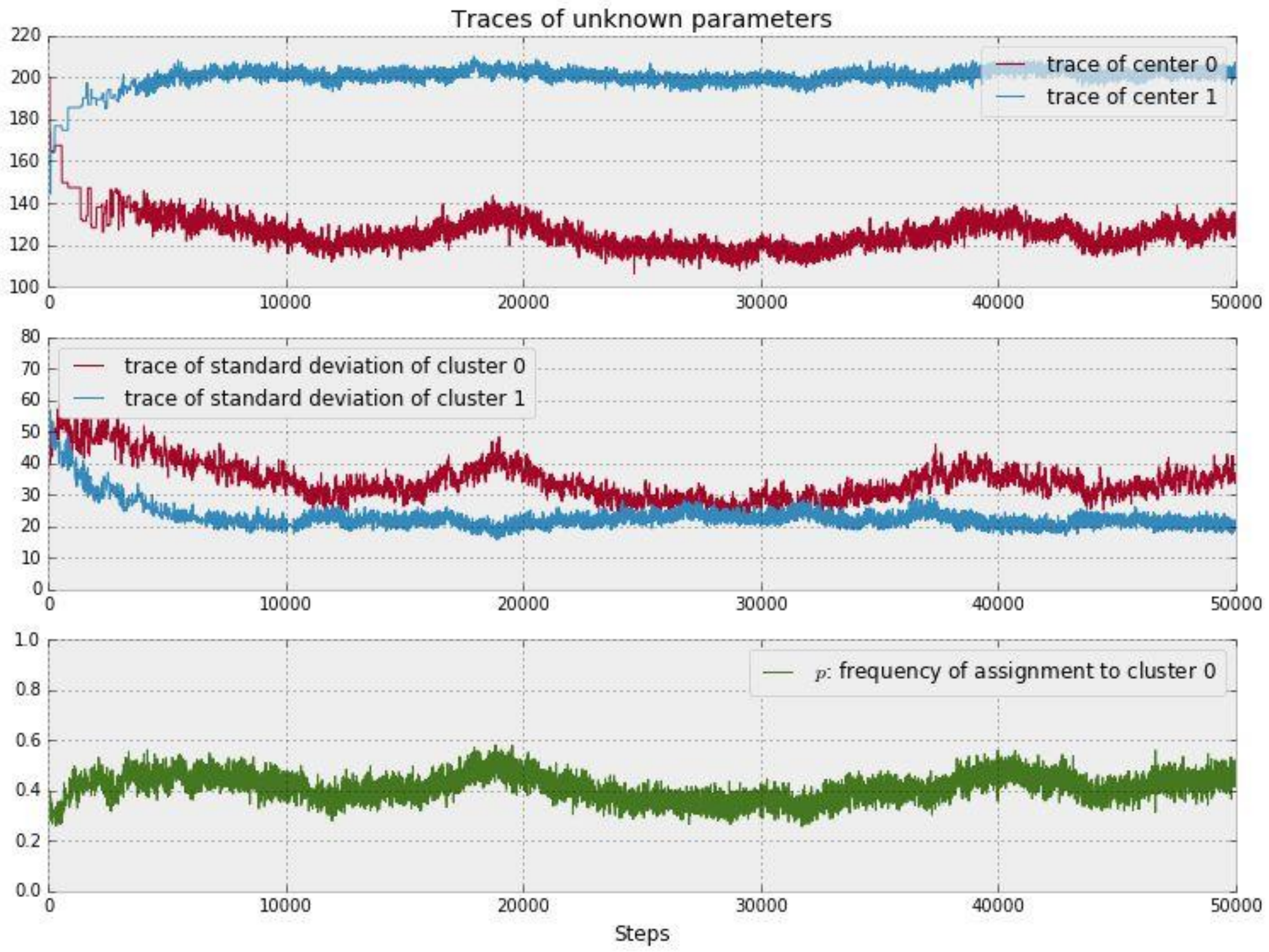
It appears the data has a bimodal form, that is, it appears to have two peaks, one near 120 and the other near 200. Perhaps there are two clusters within this dataset.

We can propose how the data might have been created:

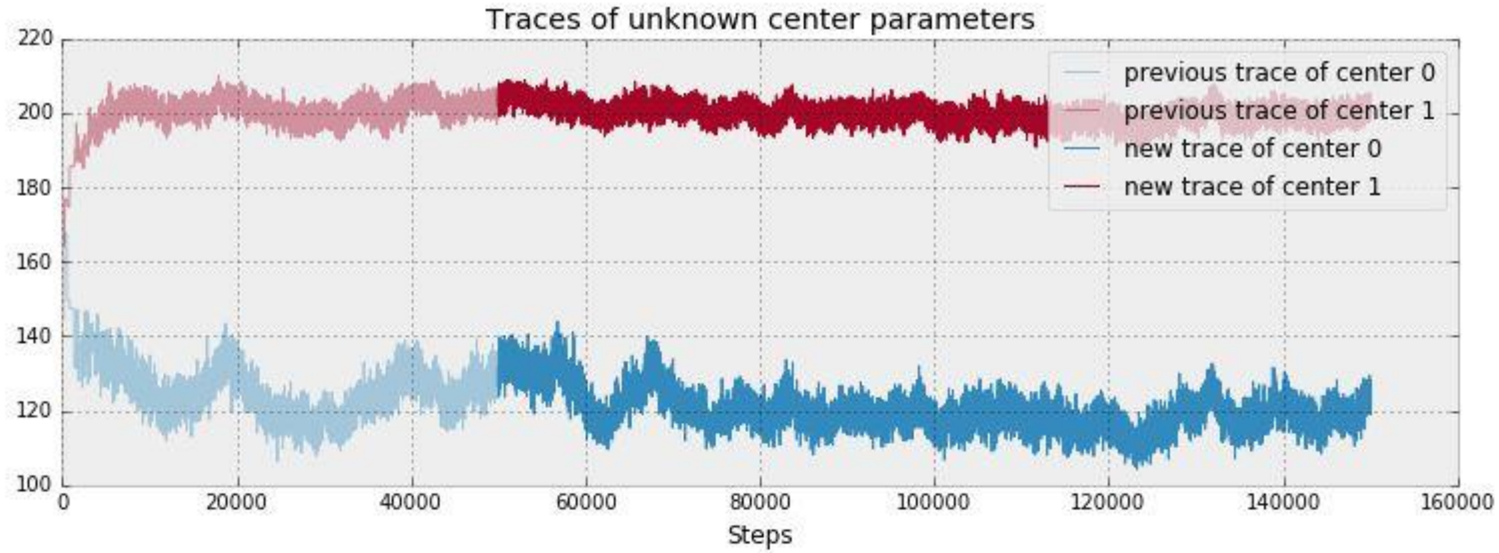
1. For each data point, choose cluster 1 with probability p, else choose cluster 2.
2. Draw a random variate from a Normal distribution with parameters μ_i and σ_i where i was chosen in step 1.
3. Repeat.

PyMC Mixture Model

```
data = np.loadtxt("mixture_data.csv", delimiter=",")

p = pm.Uniform("p", 0, 1)
assignment = pm.Categorical("assignment", [p, 1 - p], size=data.shape[0])
stds = pm.Uniform("stds", 0, 100, size=2)
taus = 1.0 / stds ** 2
centers = pm.Normal("centers", [120, 190], [0.01, 0.01], size=2)

@pm.deterministic
def center_i(assignment=assignment, centers=centers):
    return centers[assignment]

@pm.deterministic
def tau_i(assignment=assignment, taus=taus):
    return taus[assignment]

observations=pm.Normal("obs", center_i, tau_i, value=data, observed=True)

model = pm.Model([p, assignment, observations, taus, centers])
mcmc = pm.MCMC(model)
mcmc.sample(50000)
```

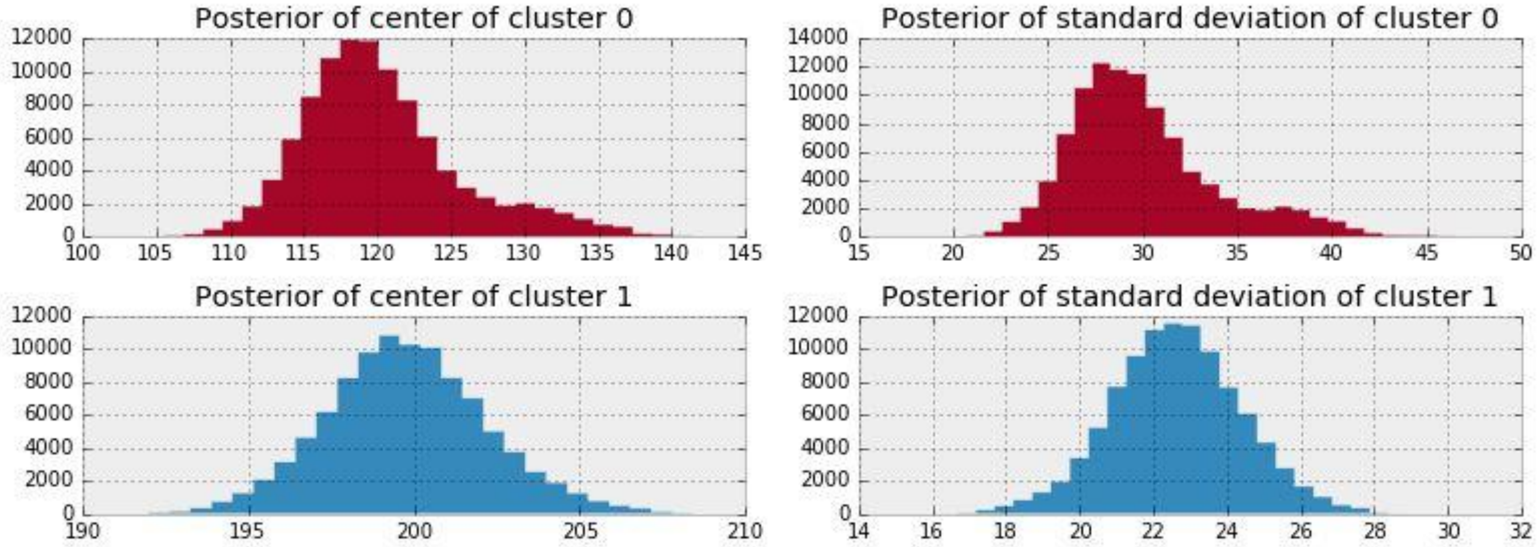

Traces of unknown parameters
220
200
180
160
140
120
100
trace of center 0
trace of center 1
0
10000
20000
30000
40000
50000
80
70
60
50
40
30
20
10
0
trace of standard deviation of cluster 0
trace of standard deviation of cluster 1
1.0
0.8
0.6
0.4
0.2
0.0
p: frequency of assignment to cluster 0
Steps

MCMC after one hundred thousand more times

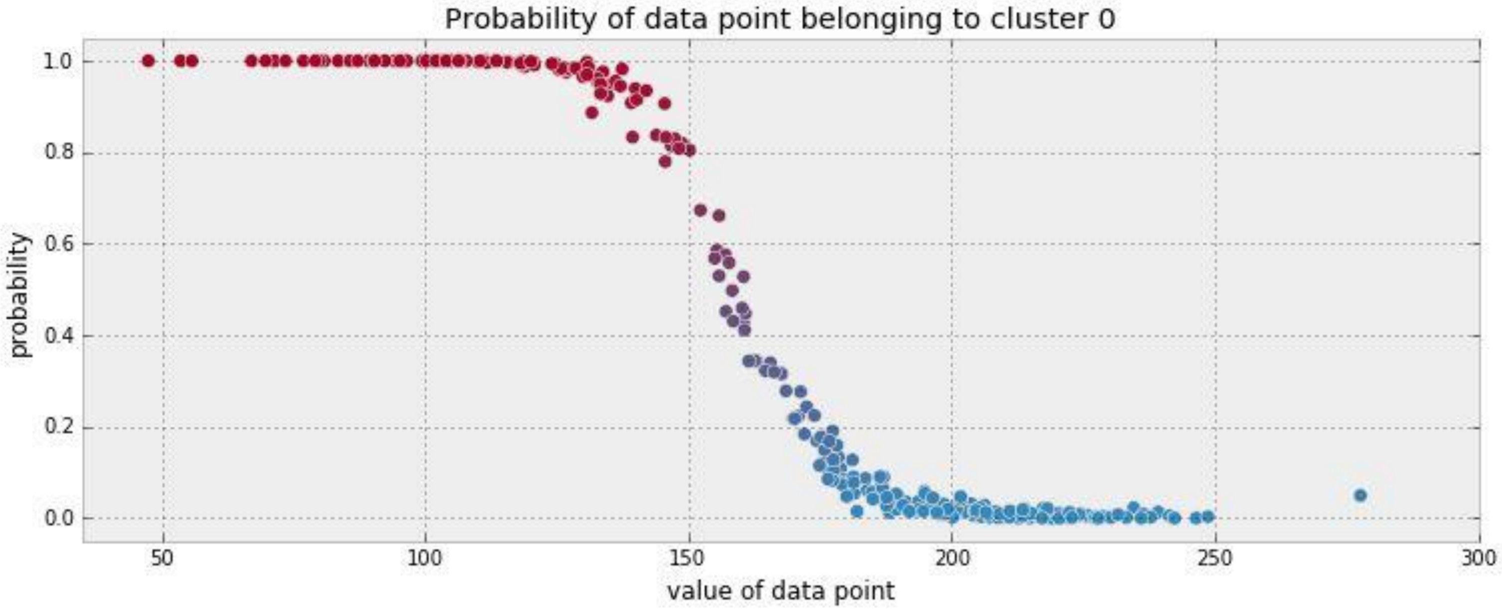


```
mcmc.sample(100000)

center_trace = mcmc.trace("centers", chain=1)[:]

prev_center_trace = mcmc.trace("centers", chain=0)[:]
```

Posterior Distributions of the Center and Standard Deviation Variables

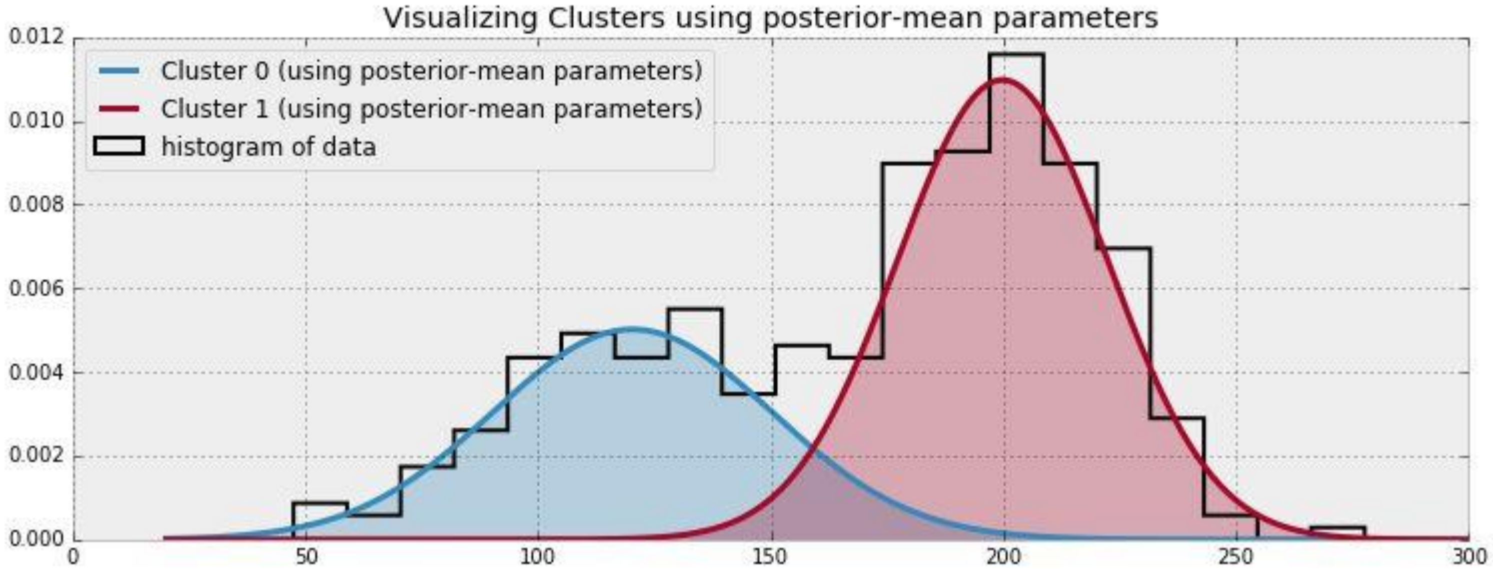


The Posterior Distributions for the Labels of the Data Points



How can we choose just a single pair of values for the mean and variance and determine a sorta-best-fit gaussian?

One quick way (which has nice theoretical properties), is to use the mean of the posterior distributions



Cluster Prediction

What about prediction? Suppose we observe a new data point, say $x = 175$, and we wish to label it to a cluster. It is foolish to simply assign it to the closer cluster center, as this ignores the standard deviation of the clusters, and we have seen from the plots above that this consideration is very important. More formally: we are interested in the probability (as we cannot be certain about labels) of assigning $x = 175$ to cluster 1. Denote the assignment of x as L_x, which is equal to 0 or 1, and we are interested in $\mathrm{P}(L_x = 1 | x = 175)$.

A naive method to compute this is to re-run the above MCMC with the additional data point appended. The disadvantage with this method is that it will be slow to infer for each novel data point. Alternatively, we can try a less precise, but much quicker method.

Cluster Prediction

For a particular sample set of parameters for our posterior distribution, $(\mu_0, \sigma_0, \mu_1, \sigma_1, p)$, we are interested in asking "Is the probability that x is in cluster 1 greater than the probability it is in cluster 0?", where the probability is dependent on the chosen parameters.

$$P(L_x = 1 | x = 175) > P(L_x = 0 | x = 175)$$

$$\frac{P(x = 175 | L_x = 1) P(L_x = 1)}{P(x = 175)} > \frac{P(x = 175 | L_x = 0) P(L_x = 0)}{P(x = 175)}$$

$$P(x = 175 | L_x = 1) P(L_x = 1) > P(x = 175 | L_x = 0) P(L_x = 0)$$

```
norm_pdf = stats.norm.pdf
p_trace = mcmc.trace("p")[:]
x = 175

v = p_trace * norm_pdf(x, loc=center_trace[:, 0], scale=std_trace[:, 0]) > \
    (1 - p_trace) * norm_pdf(x, loc=center_trace[:, 1], scale=std_trace[:, 1])

print("Probability of belonging to cluster 0:", v.mean())   # Probability of belonging to cluster 0: 0.025
```

Diagnosing Convergence

MCMC: Science & Art

- Science:

  The Markov chain will converge to the true posterior distribution… after infinite iterations

- Art:

  To determine the minimum number of samples required to ensure a reasonable approximation to the target posterior density

Assessing Convergence

- One approach to analyzing convergence is analytical, whereby the variance of the sample at different sections of the chain are compared to that of the limiting distribution. These methods use distance metrics to analyze convergence, or place theoretical bounds on the sample variance, and though they are promising, they are generally difficult to use and are not prominent in the MCMC literature.
- More common is a statistical approach to assessing convergence. With this approach, rather than considering the properties of the theoretical target distribution, only the statistical properties of the observed chain are analyzed. Reliance on the sample alone restricts such convergence criteria to heuristics. As a result, convergence cannot be guaranteed. Although evidence for lack of convergence using statistical convergence diagnostics will correctly imply lack of convergence in the chain, the absence of such evidence will not guarantee convergence in the chain. Nevertheless, negative results for one or more criteria may provide some measure of assurance to users that their sample will provide valid inferences.
- **The most straightforward approach for assessing convergence is based on simply plotting and inspecting traces and histograms of the observed MCMC sample.**

Assessing Convergence: Good Signs

- If the trace of values for each of the stochastics exhibits asymptotic behavior over the last m iterations, this may be satisfactory evidence for convergence.
- A similar approach involves plotting a histogram for every set of k iterations (perhaps 50-100) beyond some burn in threshold n; if the histograms are not visibly different among the sample intervals, this is reasonable evidence for convergence.
- The traces of several MCMC chains initialized with different starting values give similar results. Overlaying these traces on the same set of axes should (if convergence has occurred) show each chain tending toward the same equilibrium value, with approximately the same variance.
- Chain "mixes well". (i.e., chain has run much longer than any observed timescale for correlation between samples)

Assessing Convergence: Warning Signs

- Differences within or across Markov chains
- “Poor mixing”
- Autocorrelation between states of Markov chain

Assessing Convergence: More Formal Methods

- Geweke: Compares the mean and variance of segments from the beginning and end of a single chain (`pymc.geweke, pymc.Matplot.geweke_plot`)
- Raftery: Estimates the minimum chain length needed to estimate a percentile to some precision (`pymc.raftery_lewis`)
- The Gelman-Rubin statistic uses an analysis of variance approach to assessing convergence. This diagnostic uses multiple chains to check for lack of convergence, and is based on the notion that if multiple chains have converged, by definition they should appear very similar to one another; if not, one or more of the chains has failed to converge (`pymc.gelman_rubin`)

Autocorrelation

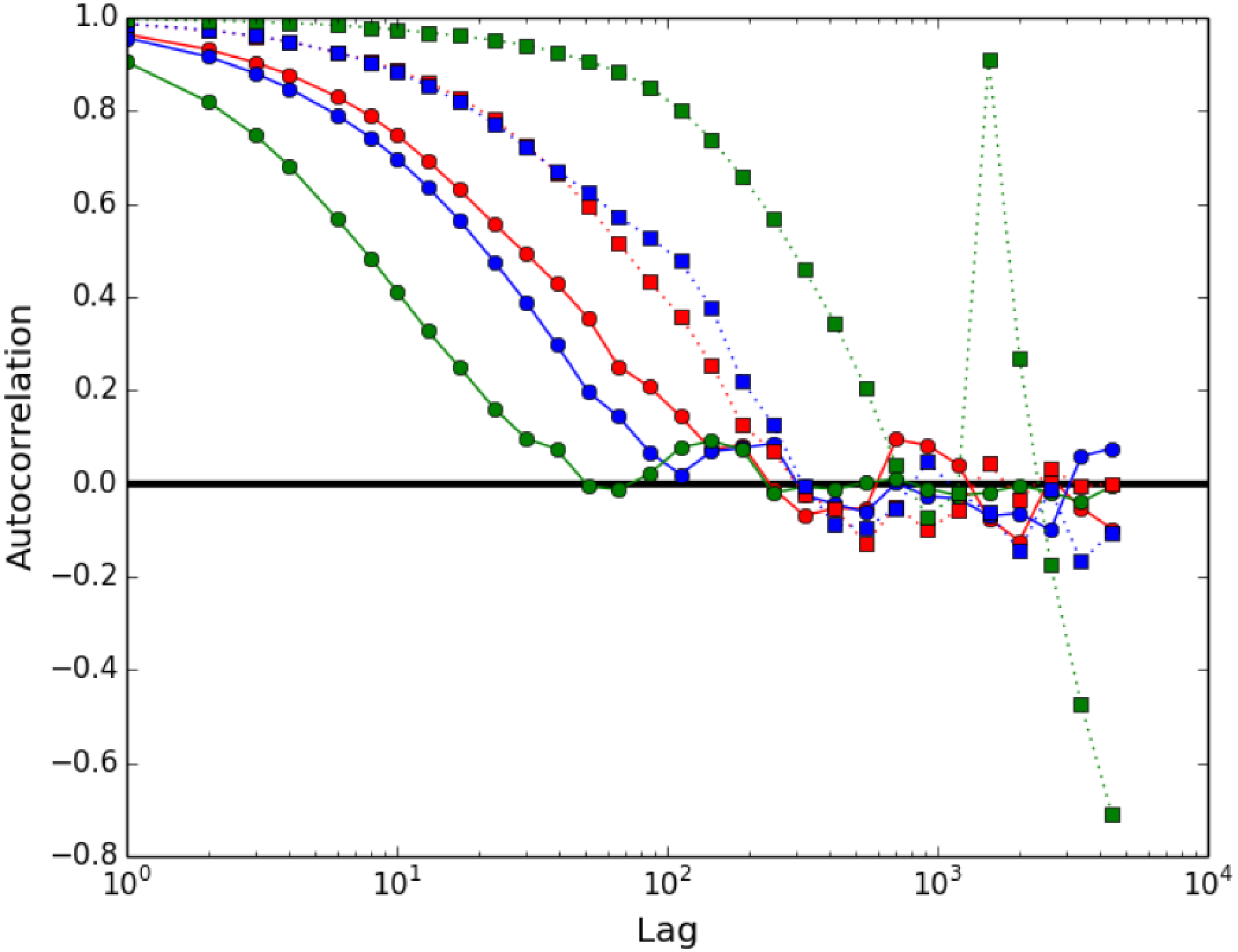
- Autocorrelation is a measure of how related a series of numbers is with itself. A measurement of 1.0 is perfect positive autocorrelation, 0 no autocorrelation, and -1 is perfect negative correlation. One way to think of autocorrelation is "If I know the position of the series at time s, can it help me know where I am at time t?"
- Samples from MCMC algorithms are ususally autocorrelated, due partly to the inherent Markovian dependence structure. The degree of autocorrelation can be quantified using the autocorrelation function:

$$\rho_k = \frac{\mathrm{Cov}(X_t, X_{t+k})}{\sqrt{\mathrm{Var}(X_t)\mathrm{Var}(X_{t+k})}} = \frac{E[(X_t - \theta)(X_{t+k} - \theta)]}{\sqrt{E[(X_t - \theta)^2 E[(X_{t+k} - \theta)^2]]}}$$

- What is smallest k (*lag*) to give an $\rho_k \approx 0$?

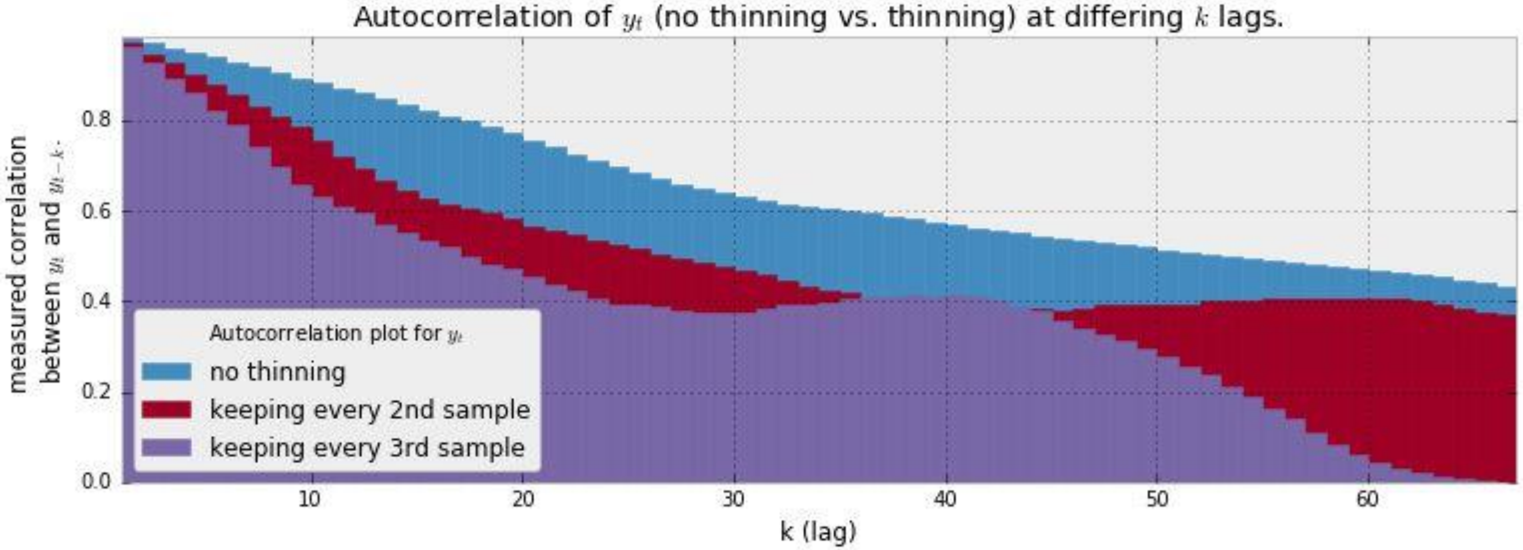
Autocorrelation

- What is smallest k (*lag*) to give an $\rho_k \approx 0$?
- One of several methods for estimating how many iterations of Markov chain are needed



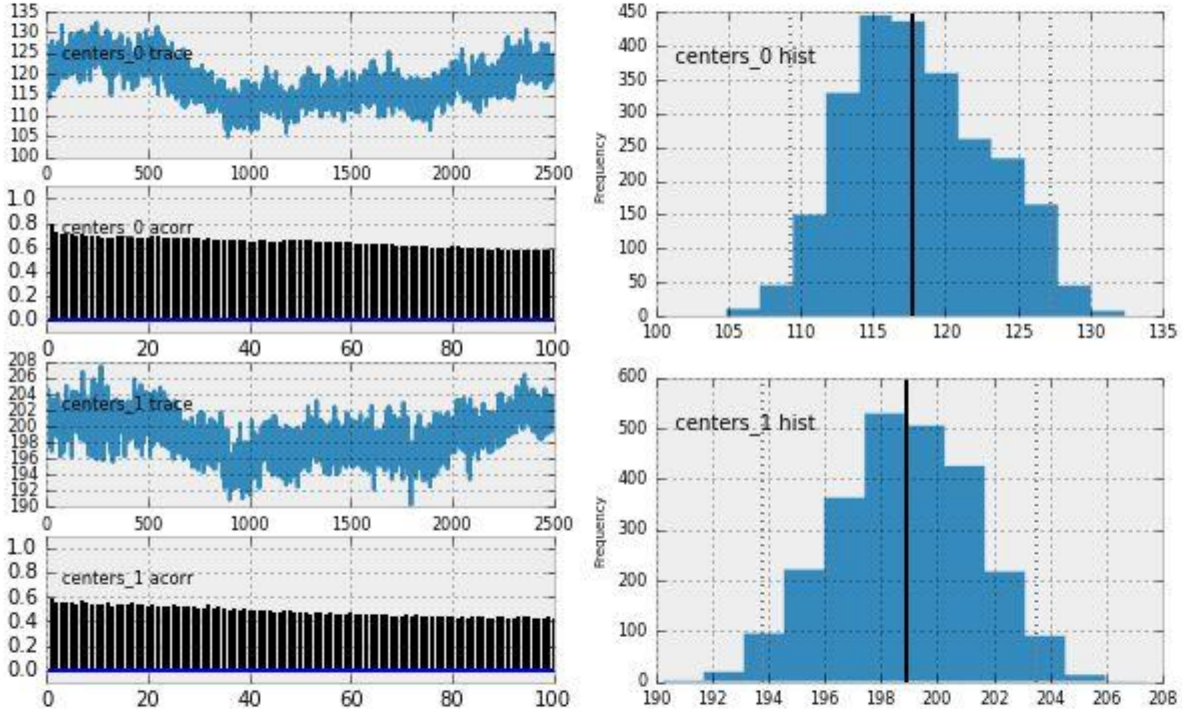
Thinning

Another issue can arise if there is high-autocorrelation between posterior samples. Many post-processing algorithms require samples to be independent of each other. This can be solved, or at least reduced, by only returning to the user every nth sample, thus removing some autocorrelation. With more thinning, the autocorrelation drops quicker. There is a tradeoff though: higher thinning requires more MCMC iterations to achieve the same number of returned samples. For example, 10 000 samples unthinned is 100 000 with a thinning of 10 (though the latter has less autocorrelation).



pymc.Matplot.plot()

```
from pymc.Matplot import plot as mcplot


mcmc.sample(25000, 0, 10)

mcplot(mcmc.trace("centers", 2), common_scale=False)
```



Useful Tips for MCMC

Good heuristics to help convergence and speed up the MCMC

Intelligent Starting Values

- It would be great to start the MCMC algorithm off near the posterior distribution, so that it will take little time to start sampling correctly. We can aid the algorithm by telling where we *think* the posterior distribution will be by specifying the `value` parameter in the `Stochastic` variable creation. In many cases we can produce a reasonable guess for the parameter. For example, if we have data from a Normal distribution, and we wish to estimate the μ parameter, then a good starting value would be the *mean* of the data.

- For most parameters in models, there is a frequentist estimate of it. These estimates are a good starting value for MCMC algorithms.

Using MAP to Improve Convergence

- Poor starting values can prevent any convergence, or significantly slow it down. Ideally, we would like to have the chain start at the *peak* of our landscape, as this is exactly where the posterior distributions exist. Hence, if we started at the "peak", we could avoid a lengthy burn-in period and incorrect inference. Generally, we call this "peak" the *maximum a posterior* or, more simply, the MAP.
- Of course, we do not know where the MAP is. PyMC provides an object that will approximate, if not find, the MAP location.

```
model = pm.Model( [p, assignment, taus, centers ] )


map_ = pm.MAP( model )
map_.fit() #stores the fitted variables' values in foo.value


mcmc = pm.MCMC( model )
mcmc.sample( 100000, 50000 )
```

- The MAP can also be used as a solution to the inference problem, as mathematically it is the most likely value for the unknowns. But, this location ignores the uncertainty and doesn't return a distribution.

Priors

- If the priors are poorly chosen, the MCMC algorithm may not converge, or at least have difficulty converging. Consider what may happen if the prior chosen does not even contain the true parameter: the prior assigns 0 probability to the unknown, hence the posterior will assign 0 probability as well. This can cause pathological results.

- Often, lack of convergence or evidence of samples crowding to boundaries implies something is wrong with the chosen priors.

Custom Step Methods

Random Graphs: Waxman Random Networks

Spatially embedded random networks (SERNs) stem from the notion that longer links are more expensive. The probability of an edge in a SERN is dependent on the distance between the two nodes. Edges in a SERN are independent (conditional on distance), and hence the probability distribution of a spatially embedded random network is given by:

$$P(G) = \prod_{(i,j)\in E} p_{ij} \prod_{(i,j)\notin E} (1 - p_{ij})$$

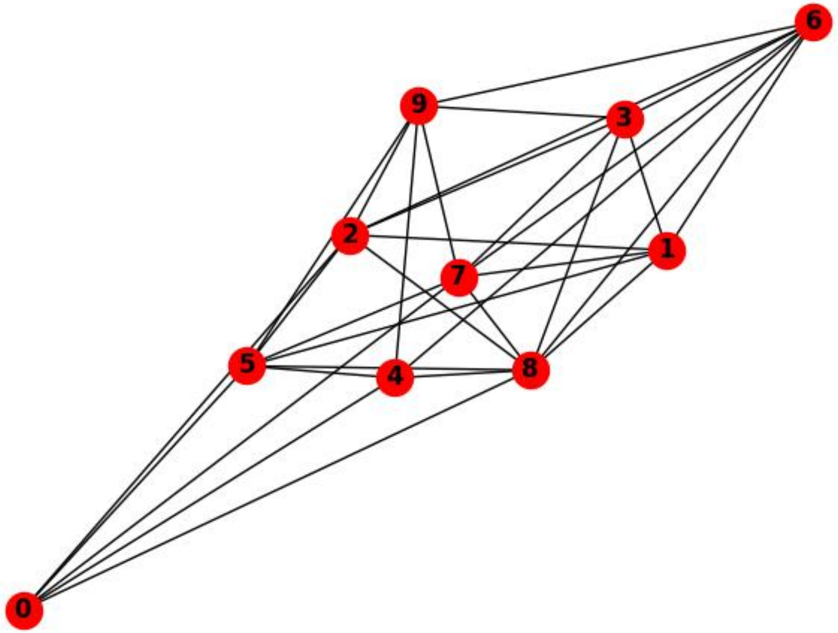
where p_{ij} is the probability of an edge for the specific SERN of interest

In the Waxman case:

$$p_{ij} = \alpha e^{-\frac{d_{ij}}{\beta L}}$$

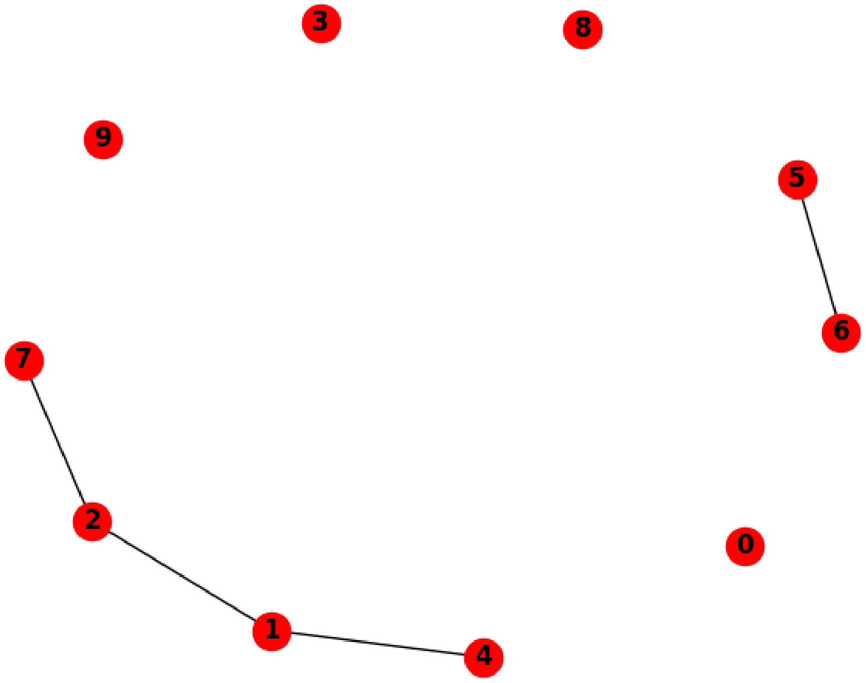
Random Graphs: Waxman Random Networks

```
import networkx as nx
from matplotlib import pyplot as plt

G = nx.waxman_graph(10, alpha=1, beta=1, L=9, domain=(1, 0, 10, 0))

nx.draw(G, with_labels=True, font_weight='bold’)

plt.show()
```



Random Graphs: Waxman Random Networks

```
import networkx as nx
from matplotlib import pyplot as plt

G = nx.waxman_graph(10, alpha=0.5, beta=0.1, L=9, domain=(1, 0, 10, 0))

nx.draw(G, with_labels=True, font_weight='bold’)

plt.show()
```



Generating Connected Waxman Graphs

Idea:

We start from a connected graph. We choose two distinct nodes at random and consider the possible link between them. If the link is already in the graph, we consider it for deletion, and if the link is not in the graph, we consider it for inclusion.

Gray, C., Mitchell, L., & Roughan, M. (2018). Generating Connected Random Graphs. arXiv preprint arXiv:1806.11276.

Start from the Complete Graph 😉

```
alpha = 0.5
beta = 0.1
L= 9.0

G0 = nx.Graph()

for i in range(1, 10):
    for j in range(i + 1, 11):
        G0.add_edge(i, j)

#G0.add_path(range(1, 11))

#G0.add_path(range(1, 11))
#G0.remove_edge(2, 3)#G0.remove_edge(3, 4)#G0.add_edge(2, 4)
#G0.add_edge(3, 7)
#G0.add_edge(8, 10)
```

Waxman Graphs Likelihood

$$P(G) = \prod_{(i,j)\in E} p_{ij} \prod_{(i,j)\notin E} (1 - p_{ij})$$

```
@pm.stochastic(dtype=nx.Graph)
def cwg(value = G0, alpha = alpha, beta = beta, L = L):
    tmp = 0
    for i in range(1, len(value)):
        for j in range(i + 1, len(value)+1):
            if value.has_edge(i, j):
                tmp += np.log(alpha) - ((j - i) / (beta * L))
            else:
                tmp += np.log(1 - alpha * np.exp((i - j) / (beta * L)))
    return tmp
```

Subclassing Metropolis

A Metropolis-Hastings step method only needs to implement the following methods, which are called by `Metropolis.step()`:

- `propose()`: sets the values of all `self.stochastics` to new, proposed values
- `reject()`: usually just restore the state

```
class CWGMetropolis(pm.Metropolis):

def __init__(self, stochastic):
        # Initialize superclass
        pm.Metropolis.__init__(self, stochastic, scale=1., verbose=0, tally=False)
```

Subclassing Metropolis

```
def propose(self):
        """ Add an edge or remove an edge"""
        G = self.stochastic.value

        G.u_new = np.random.choice(G.nodes()); G.v_new = np.random.choice(G.nodes())
        while G.u_new == G.v_new:
            G.v_new = np.random.choice(G.nodes())

        if G.has_edge(G.u_new, G.v_new):
            G.remove_edge(G.u_new, G.v_new)
            if not nx.is_connected(G):
                G.add_edge(G.u_new, G.v_new)
        else:
            G.add_edge(G.u_new, G.v_new)
        self.stochastic.value = G
```

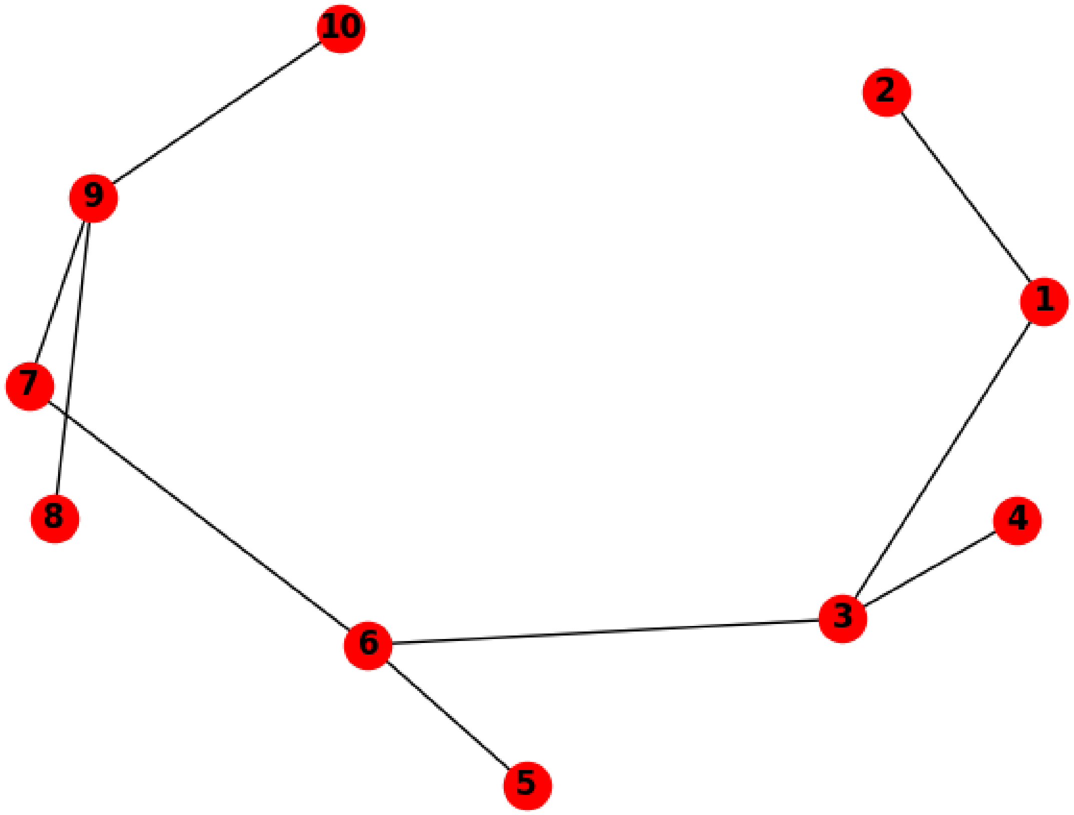
Subclassing Metropolis

```
def reject(self):
      """ Restore the graph"""
      G = self.stochastic.value
      if G.has_edge(G.u_new, G.v_new):
          G.remove_edge(G.u_new, G.v_new)
      else:
          G.add_edge(G.u_new, G.v_new)
      self.rejected += 1
      self.stochastic.value = G
```

MCMC

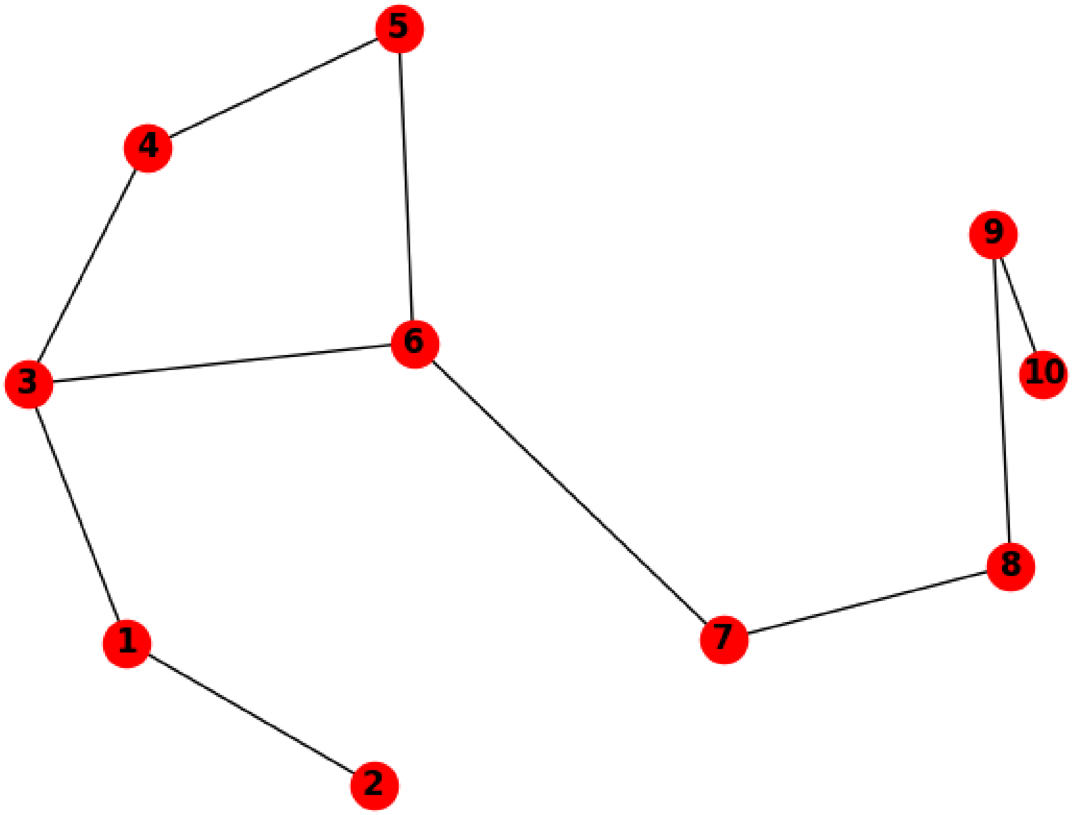
```
mcmc = pm.MCMC([cwg, average_degree])

mcmc.use_step_method(CWGMetropolis, cwg)

mcmc.sample(100000)
```

Results



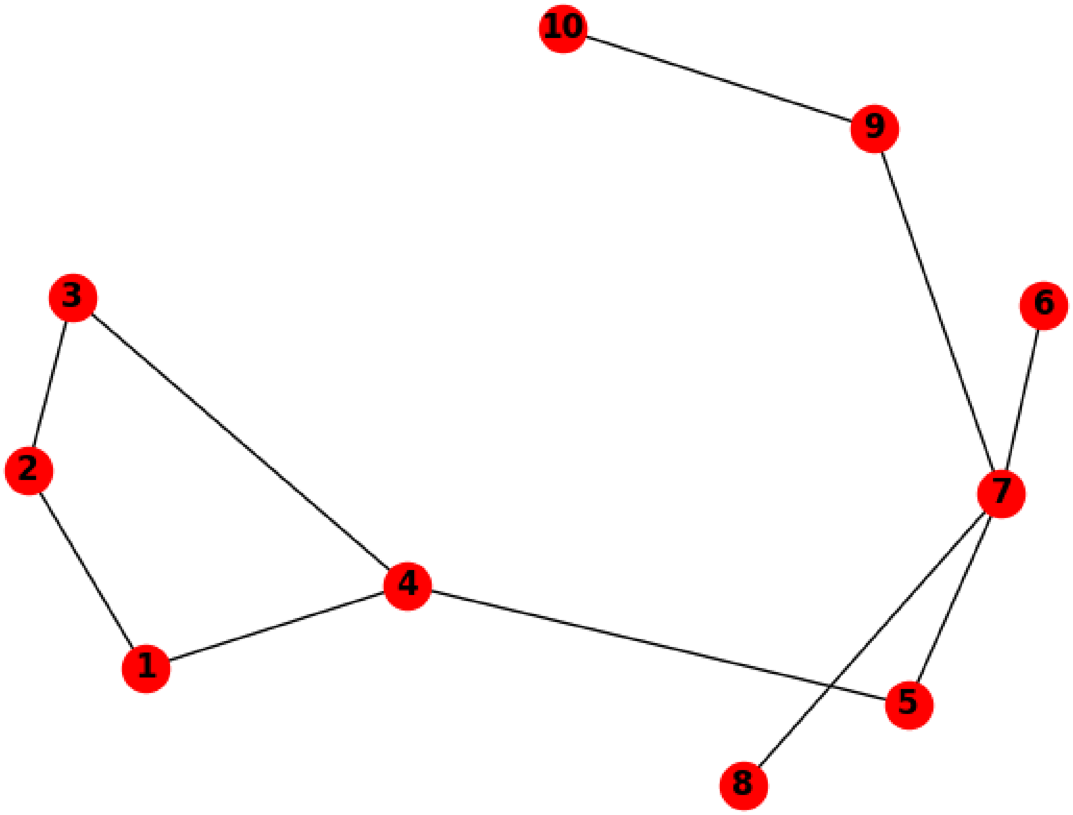
```
mcmc.sample(100)
nx.draw(cwg.value, with_labels=True, font_weight='bold')
plt.show()
```

Results



```
mcmc.sample(100)
nx.draw(cwg.value, with_labels=True, font_weight='bold')
plt.show()
```

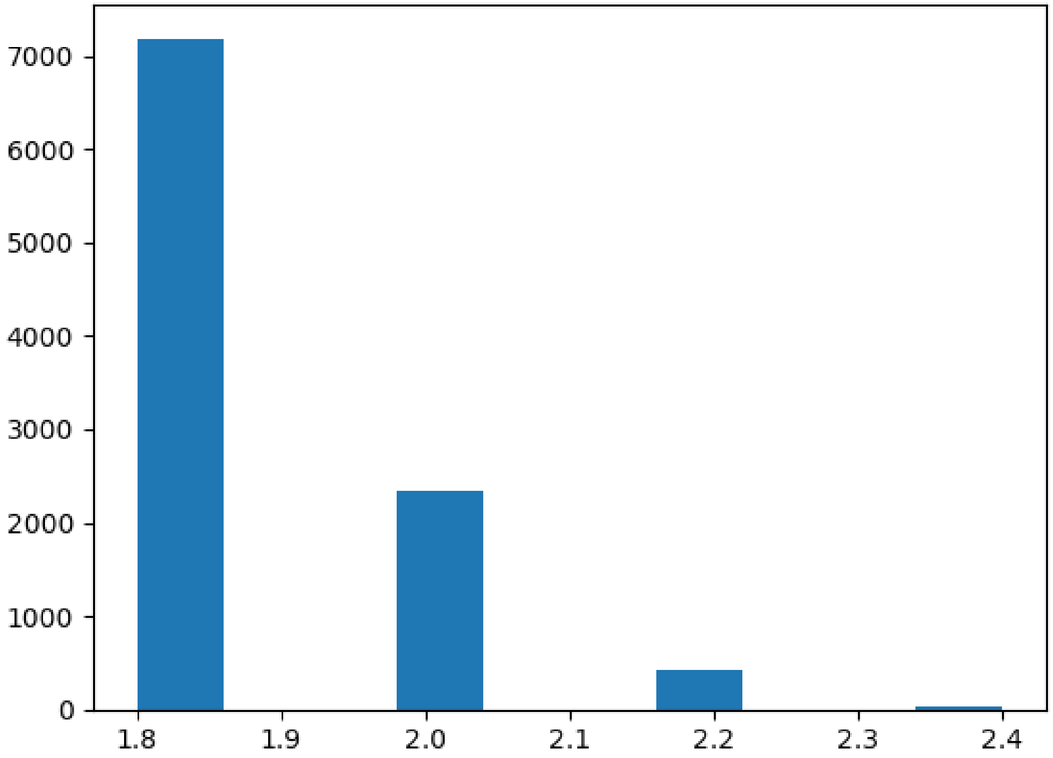
Results



```
mcmc.sample(100)
nx.draw(cwg.value, with_labels=True, font_weight='bold')
plt.show()
```

The Distribution of Average Degrees

```
@pm.deterministic
def average_degree(G = cwg):
#    return np.sum([t[1] for t in list(G.degree())]) /
len(G)
    return np.sum(list(G.degree().values())) / len(G)


avgd_samples = mcmc.trace("average_degree")[:]
```


7000
6000
5000
4000
3000
2000
1000
0
1.8
1.9
2.0
2.1
2.2
2.3
2.4