Nathan Hadfield

CS 677 (Spring 2014)

# MCMC Lab #2

Code used set up each section of the lab is included below the charts. Code for implementing the nodes and the network is provided at the end.

## Faculty Evaluations

|  |  |
| --- | --- |
|  |  |
|  |  |

### network\_faculty.py

import numpy

from node\_normal import \*

from node\_invgamma import \*

from network import \*

datafilename = 'faculty.dat'

nsamples = 10000

burn = 0

mean\_candsd = 0.2

var\_candsd = 0.15

# Read in Data

data = [float(line) for line in open(datafilename)]

# Use point estimators from the data to come up with starting values.

estimated\_mean = numpy.mean(data)

estimated\_var = numpy.var(data)

def MomentsInvGammaShape(mean, var):

return 1

def MomentsInvGammaScale(mean,var):

return 1

# Create Nodes and Links in Network

meannode = NormalNode(estimated\_mean, name='Mean', cand\_var=mean\_candsd, mean=5, var=(1/3)\*\*2)

varprior\_mean = 1/4

varprior\_stddev = 1/12

varprior\_shape = MomentsInvGammaShape(varprior\_mean, varprior\_stddev\*\*2)

varprior\_scale = MomentsInvGammaScale(varprior\_mean, varprior\_stddev\*\*2)

varnode = InvGammaNode(estimated\_var, name='Variance', cand\_var=var\_candsd, shape=varprior\_shape, scale=varprior\_scale)

for datum in data:

NormalNode(datum, observed=True, mean=meannode, var=varnode)

# Perform simulations and plot results

network = Network([meannode, varnode])

samples = network.collect\_samples(burn, nsamples)

def mean\_prior\_pdf(x):

return stats.norm.pdf(x, 5, 1/3)

def var\_prior\_pdf(x):

return stats.invgamma.pdf(x, a=varprior\_shape, scale=varprior\_scale)

prior\_pdfs = { meannode: mean\_prior\_pdf, varnode: var\_prior\_pdf }

results = {}

for node in [meannode, varnode]:

params = {

'mean': numpy.mean(samples.of\_node(node)),

'var': numpy.var(samples.of\_node(node))

}

results[node] = params

title = "{}: mean = {}, var = {} (burn={}, n={})". \

format(node.pdf\_name, params['mean'], params['var'], burn, nsamples - burn)

samples.plot\_node(node, title=title)

if params['var'] > 0: # histogram fails if all values are the same

samples.plot\_histogram\_for\_node(node, title=title, prior\_pdf=prior\_pdfs[node])

## Professional Golfers

I haven’t yet been able to obtain the same results provided in the lab instructions. Perhaps I just need to let it run longer.

Here are results:

**0 burn, 1000 samples:**

1: MichaelBradley 68.506359; 90% interval: (68.168512, 71.818313)

2: MikeWeir 68.568669; 90% interval: (66.959685, 71.818313)

3: SteveFriesen 68.870070; 90% interval: (67.704821, 71.818313)

4: FrankBensel 68.932266; 90% interval: (68.868960, 71.065689)

5: UlyGrisette 68.988229; 90% interval: (67.596112, 71.818313)

6: DaisukeMaruyama 69.113726; 90% interval: (68.885119, 70.123087)

7: PaulAzinger 69.162171; 90% interval: (68.837082, 71.818313)

8: DavidLundstrom 69.181070; 90% interval: (66.685543, 71.818313)

9: HidemichiTanaka 69.278051; 90% interval: (69.116481, 71.818313)

10: ShingoKatayama 69.298395; 90% interval: (69.126480, 71.818313)

…

594: JamieElliott 74.107154; 90% interval: (71.818313, 74.116564)

595: RodPampling 74.201335; 90% interval: (71.818313, 75.632426)

596: AndreStolz 74.300154; 90% interval: (71.818313, 74.451293)

597: RodCurl 74.355775; 90% interval: (71.818313, 74.475204)

598: ScottDunlap 74.383805; 90% interval: (71.818313, 74.396361)

599: PatrickSheehan 74.435492; 90% interval: (71.818313, 76.893970)

600: KeithFergus 74.530361; 90% interval: (71.818313, 75.065474)

601: JasonDufner 74.566214; 90% interval: (71.818313, 74.786557)

602: JimmyGreen 74.684533; 90% interval: (71.818313, 77.636256)

603: AndyCrain 74.819675; 90% interval: (73.548603, 74.839618)

604: ChrisNallen 75.285112; 90% interval: (71.818313, 75.565003)

**0 burn, 10,000 samples:**

1: CharletonDechert 63.059197; 90% interval: (61.932321, 68.114029)

2: BrettQuigley 63.823470; 90% interval: (61.957505, 70.118672)

3: WilliamLinkIV 64.433450; 90% interval: (61.821228, 71.814321)

4: JeffBrehaut 64.435480; 90% interval: (61.659941, 70.291198)

5: BrendanJones 64.467350; 90% interval: (61.866006, 71.142384)

6: KevinDurkin 64.719385; 90% interval: (61.392781, 70.201949)

7: ChrisStroud 64.721479; 90% interval: (63.019291, 71.818313)

8: TommyTolles 64.784650; 90% interval: (62.335474, 71.682574)

9: PatPerez 64.878027; 90% interval: (63.038874, 69.531869)

10: PaulMcGinley 64.933665; 90% interval: (63.304477, 70.212636)

…

594: PierreFulke 78.028745; 90% interval: (71.996713, 82.943628)

595: TimClark 78.041545; 90% interval: (72.129058, 85.259594)

596: BrianHarman 78.588884; 90% interval: (71.818313, 83.232394)

597: BobAckerman 78.641958; 90% interval: (71.818313, 82.666595)

598: DennisColligan 78.665644; 90% interval: (71.818313, 82.519481)

599: RickFehr 78.864267; 90% interval: (73.065569, 82.801983)

600: ToshiIzawa 79.049189; 90% interval: (70.566723, 85.028566)

601: MichelleWie 79.086877; 90% interval: (72.267396, 82.618260)

602: JasonGore 79.319358; 90% interval: (71.818313, 87.248540)

603: TrippIsenhour 79.325457; 90% interval: (72.559681, 81.779186)

604: ThongchaiJaidee 79.962592; 90% interval: (71.818313, 86.828954)

**10,000 burn, 100,000 samples**

1: DavidFaught 35.567460; 90% interval: (28.085181, 64.012255)

2: JoeOgilvie 37.690226; 90% interval: (29.178827, 60.071037)

3: VanceVeazey 37.803001; 90% interval: (21.383958, 71.586654)

4: BrianDixon 40.833441; 90% interval: (26.970465, 65.075742)

5: KevinStadler 42.298471; 90% interval: (24.054469, 75.066460)

6: RobertDeruntz 42.535868; 90% interval: (31.368566, 63.595836)

7: BobTway 42.794322; 90% interval: (36.734639, 65.448897)

8: CaseyWittenberg 43.693175; 90% interval: (38.414358, 64.497341)

9: ToddBarranger 44.153859; 90% interval: (29.675191, 64.137654)

10: LorenPersonett 44.681982; 90% interval: (36.007485, 66.985873)

…

594: StephenAmes 98.209983; 90% interval: (68.929898, 114.151196)

595: EdFiori 98.758722; 90% interval: (74.667969, 105.211274)

596: BobbyKalinowski 99.448144; 90% interval: (75.693244, 115.008141)

597: MattHendrix 99.689896; 90% interval: (79.921170, 106.713186)

598: MiguelRivera 100.312942; 90% interval: (80.392490, 104.480465)

599: BoydSummerhays 101.314547; 90% interval: (80.844905, 111.848930)

600: FredCouples 101.725960; 90% interval: (65.362897, 113.869461)

601: TjaartvanderWalt 102.357989; 90% interval: (71.908438, 113.004750)

602: AndyMorse 103.448081; 90% interval: (79.315435, 118.139906)

603: HeathSlocum 108.544687; 90% interval: (80.742846, 122.449682)

604: EricAxley 111.032587; 90% interval: (84.755746, 121.812276)

### network\_golfers.py

from node\_normal import \*

from node\_invgamma import \*

from network import \*

from operator import itemgetter

import numpy

logging.basicConfig(level=logging.WARNING, format='[%(levelname)s] %(module)s %(funcName)s(): %(message)s')

hypertournmean\_candsd = 1 # variance

hypervar\_candsd = 1 # variance

mean\_candsd = 1 # variance

obsvar\_candsd = 1 # variance

# tourns # list of tournament #s

# golfers # list of golfer names

# data # tuples of (name, score, tourn) from golfdataR.dat

# est\_avg # estimated average score

data = []

for line in open('golfdataR.dat'):

line\_data = line.strip().split(' ')

line\_data[1] = float(line\_data[1])

data.append(line\_data)

# data = [line.strip().split(' ') for line in open('golfdataR.dat')]

golfers = sorted(set([line[0] for line in data]))

scores = [float(line[1]) for line in data]

tourns = sorted(set([line[2] for line in data]), key=int)

est\_avg = numpy.mean(scores)

#data = [item.strip() for item in (line.split(' ') for line in open('golfdataR.dat'))]

#data = (item.strip() for item in (line.split(' ')) for line in open('golfdataR.dat'))

hypertournmean = NormalNode(72.8, name='Tournament Hyper Mean', cand\_var=hypertournmean\_candsd, mean=72, var=2)

hypertournvar = InvGammaNode(3, name='Tournament Hyper Var', cand\_var=hypervar\_candsd, shape=18, scale=1 / .015)

tournmean = {}

for tourn in tourns:

tournmean[tourn] = NormalNode(est\_avg, name="Tournament {}".format(tourn), cand\_var=mean\_candsd,

mean=hypertournmean, var=hypertournvar)

hypergolfervar = InvGammaNode(3.5, name='Golfer Hyper Var', cand\_var=hypervar\_candsd, shape=18, scale=1 / .015)

golfermean = {}

for golfer in golfers:

golfermean[golfer] = NormalNode(est\_avg, name=golfer, cand\_var=mean\_candsd, mean=0, var=hypergolfervar)

obsvar = InvGammaNode(3.1, name='Observation Var', cand\_var=obsvar\_candsd, shape=83, scale=1 / .0014)

for (name, score, tourn) in data:

NormalNode(score, observed=True, mean=[tournmean[tourn], golfermean[name]], var=obsvar)

# sample from nodes

burn = 10000

nsamples = 100000

network = Network(

[hypertournmean, hypertournvar, hypergolfervar, obsvar] + list(tournmean.values()) + list(golfermean.values()))

samples = network.collect\_samples(burn, nsamples)

ability = []

for golfer in golfermean:

golfermean\_samples = samples.of\_node(golfermean[golfer])[:]

golfermean\_samples.sort()

median = golfermean\_samples[nsamples // 2]

low = golfermean\_samples[int(.05 \* nsamples)]

high = golfermean\_samples[int(.95 \* nsamples)]

ability.append((golfer, low, median, high))

ability = sorted(ability, key=itemgetter(2)) # sort by median score

i = 1

for golfer, low, median, high in ability:

print("{}: {} {:.6f}; 90% interval: ({:.6f}, {:.6f})".format(i, golfer, median, low, high))

i += 1

## Wacky Network

With no observations:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

With G observed to be 5:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | *(Range is empty; can’t plot a histogram.)* |

### network\_wacky.py

from node\_normal import \*

from node\_beta import \*

from node\_gamma import \*

from node\_poisson import \*

from node\_bernoulli import \*

from network import \*

import numpy

logging.basicConfig(level=logging.WARNING,

format='[%(levelname)s] %(module)s %(funcName)s(): %(message)s')

burn = 0

num\_samples = burn + 100000

for g\_observed in [False, True]:

a = NormalNode(20, 'A', mean=20, var=1)

e = BetaNode(0.5, 'E', alpha=1, beta=1)

b = GammaNode(0.2, 'B', shape=a, shape\_modifier=lambda x: x \*\* math.pi, scale=1/7)

d = BetaNode(0.5, 'D', alpha=a, beta=e)

c = BernoulliNode(0, 'C', p=d)

f = PoissonNode(4, 'F', rate=d)

g = NormalNode(5, 'G', mean=e, var=f, observed=g\_observed)

network = Network([a, e, b, d, c, f, g])

samples = network.collect\_samples(burn=burn, n=num\_samples)

for node in network.nodes:

mean = numpy.mean(samples.of\_node(node))

var = numpy.var(samples.of\_node(node))

title = "{} [G observed={}]: mean = {:.4f}, var = {:.4f} (burn={}, n={})"

.format(node.pdf\_name, g\_observed, mean, var, burn, num\_samples-burn)

samples.plot\_node(node, title=title)

samples.plot\_histogram\_for\_node(node, title=title)

## My Network

What happens to the network when D is observed at 5?

a = BetaNode(0.4, 'A', alpha=2, beta=2)

b = GammaNode(4, 'B', shape=3, scale=1/2)

c = NormalNode(0, 'C', mean=b, var=a)

d = PoissonNode(5, 'D', rate=b, observed=d\_observed)

D not observed:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

D observed at 5:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  | *(Range is empty; can’t plot a histogram.)* |

## Metropolis Implementation

### network.py

The Network class stores the nodes and initiates the sampling process. When it is finished, it returns the results in a SampleProcessor object, which can be used to compute statistics and generate plots.

import logging

import evilplot

log = logging.getLogger("network")

class Network(object):

def \_\_init\_\_(self, nodes=None):

self.nodes = [] if nodes is None else nodes

def \_\_str\_\_(self):

pass

def metropolis\_sample\_generator(self):

"""Create samples from the given nodes using the Metrpolis algorithm."""

while True:

for test\_node in self.nodes:

test\_node.sample\_with\_metropolis()

network\_state = []

for node in self.nodes:

network\_state.append(node.current\_value)

yield network\_state

def collect\_samples(self, burn, n, generator=None):

"""Run burn iterations, then collect n samples"""

mcmc = generator

if mcmc is None:

mcmc = self.metropolis\_sample\_generator()

progress\_step = (burn + n) / 10

cur\_sample = 0

log.info("Burning...")

for i in range(burn):

next(mcmc)

cur\_sample += 1

if cur\_sample % progress\_step == 0:

log.warning("{:.0%}... ".format(cur\_sample/(burn+n)))

log.info("Sampling...")

samples = []

for i in range(n):

sample = next(mcmc)

log.debug("Sample: " + str(sample))

samples.append(next(mcmc))

cur\_sample += 1

if cur\_sample % progress\_step == 0:

log.warning("{:.0%}... ".format(cur\_sample/(burn+n)))

return SamplesProcessor(self.nodes, samples)

class SamplesProcessor(object):

def \_\_init\_\_(self, nodes, samples):

if not type(nodes) is list:

raise AssertionError("'nodes' argument is not a list (type = " + type(nodes).\_\_name\_\_ + ")")

self.nodes = nodes

self.samples = samples

def \_\_str\_\_(self):

samples\_str = ", ".join([node.name for node in self.nodes]) + "\n"

samples\_str += "\n".join([", ".join(map(str, sample)) for sample in self.samples])

return samples\_str

def of\_node(self, node):

"""Returns samples for the given node"""

samples = [sample[self.nodes.index(node)] for sample in self.samples]

return samples

def plot\_node(self, node, title=None):

if title is None:

title = u"Samples of {0:s}".format(node.display\_name)

p = evilplot.Plot(title=title)

points = evilplot.Points(list(enumerate(self.of\_node(node))))

points.style = 'lines'

points.linewidth = 1

p.append(points)

p.show()

def plot\_histogram\_for\_node(self, node, title=None, prior\_pdf=None):

if title is None:

title = u"Histogram of samples of {0:s}".format(node.display\_name)

p = evilplot.Plot(title=title)

if not prior\_pdf is None:

priord = evilplot.Function(prior\_pdf)

priord.title = "Prior Dist"

p.append(priord)

hist = evilplot.Histogram(self.of\_node(node), 50, normalize=True)

hist.title = node.display\_name

p.append(hist)

p.show()

### node.py

All of the nodes inherit the Node class, which provides common functionality for Metropolis sampling. The heart of the class is sample\_with\_metropolis(), which implements the core of the the Metropolis algorithm. Subclasses implement log\_current\_conditional\_probability(), which returns a probability for the given node type conditional upon the values of its parents.

import random

import logging

import math

\_log = logging.getLogger("nodes")

class Node:

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_init\_\_(self, value=None, name=None, cand\_var=1, observed=False):

self.name = name

self.current\_value = value

self.cand\_std\_dev = math.sqrt(cand\_var) # std\_dev of Gaussian distribution used to generate candidates

self.is\_observed = observed

self.\_children = [] # subclass init methods should add self to parents' children

self.\_log\_p\_current\_value = None # log of the last sample

def \_\_str\_\_(self):

return self.display\_name()

@property

def pdf\_name(self):

return self.display\_name()

@property

def node\_type(self):

return self.\_\_class\_\_.\_\_name\_\_

@property

def display\_name(self):

return self.name if not self.name is None else self.node\_type

@staticmethod

def parent\_node\_str(node):

return "{:.4f}".format(node) if not isinstance(node, Node) else node.display\_name

@staticmethod

def parent\_node\_value(node):

"""

If node is a list of parent nodes, returns the sum of their values.

"""

if isinstance(node, Node):

return node.current\_value

elif isinstance(node, list):

return sum([Node.parent\_node\_value(a\_node) for a\_node in node])

else:

return node

def connect\_to\_parent\_node(self, parent):

"""

If parent is a list nodes, connects to each of them.

"""

if isinstance(parent, Node):

parent.\_children.append(self)

elif isinstance(parent, list):

for parentnode in parent:

self.connect\_to\_parent\_node(parentnode)

def current\_conditional\_probability(self):

"""Provided for testing; use log\_current\_conditional\_probability instead."""

return math.exp(self.log\_current\_conditional\_probability())

def log\_current\_conditional\_probability(self):

"""Compute the conditional probability of this node given its parents"""

raise NotImplementedError

def current\_unnormalized\_mb\_probability(self):

"""Provided for testing; use log\_current\_unnormalized\_mb\_probability instead."""

return math.exp(self.log\_current\_unnormalized\_mb\_probability())

def log\_current\_unnormalized\_mb\_probability(self):

p = 0.0

for node in self.\_children + [self]:

p += node.log\_current\_conditional\_probability()

return p

def probability\_of\_current\_value\_given\_other\_nodes(self):

return math.exp(self.log\_probability\_of\_current\_value\_given\_other\_nodes())

def log\_probability\_of\_current\_value\_given\_other\_nodes(self):

"""

Needed only for Gibbs sampling. Metropolis sampling only requires

a probability that is proportional to the actual probability, which

saves us from having to determine the integral for the marginal

probability.

"""

raise NotImplementedError

def is\_candidate\_in\_domain(self, cand):

"""Overridden by subclasses to reject samples that are outside the domain of the probability function."""

return True

def select\_candidate(self):

"""Can be overridden by subclasses in order to provide custom distributions. Default is Gaussian."""

return random.gauss(self.current\_value, self.cand\_std\_dev)

def sample\_with\_gibbs(self):

"""

Samples boolean values.

"""

if not self.is\_observed:

p = self.probability\_of\_current\_value\_given\_other\_nodes()

r = random.random()

self.current\_value = (r < p)

\_log.debug("P(" + self.name + ") = " + str(p))

def sample\_with\_metropolis(self):

"""Sample this node using Metropolis."""

\_log.debug("Sampling {}...".format(self))

if not self.is\_observed:

# Metropolis:

# 1 - Use the candidate distribution to select a candidate.

# 2 - Compare the (proportionate) probability of the candidate with the

# (proportionate) probability of the current value.

# 3 - If the probability of the candidate is greater, use it.

# Otherwise, determine whether to use it as a random selection with

# probability proportionate to the probability of the current value.

# 1 - Select a candidate. (Since we're not using Metropolis-Hastings,

# we use a Gaussian normal with variance provided by parameter 'cand\_var'.)

cand = self.select\_candidate()

\_log.debug("last: {}, cand: {}".format(self.current\_value, cand))

# If the candidate falls outside the domain of the probability function,

# we can skip it immediately.

if self.is\_candidate\_in\_domain(cand):

# 2 - Compare the probability of the candidate with that of the current value

# log\_p\_cand = candidate probability

saved\_value = self.current\_value

self.current\_value = cand

log\_p\_cand = self.log\_current\_unnormalized\_mb\_probability()

self.current\_value = saved\_value

# log\_p\_current\_value = current probability

if self.\_log\_p\_current\_value is None:

self.\_log\_p\_current\_value = self.log\_current\_unnormalized\_mb\_probability()

log\_r = log\_p\_cand - self.\_log\_p\_current\_value

log\_u = math.log(random.random())

\_log.debug("log\_r = {}, log\_u = {}".format(log\_r, log\_u))

# 3 - Use candidate with probability proportionate to the ratio of

# its likelihood over the likelihood of the current value.

if log\_u < log\_r:

self.current\_value = cand

self.\_log\_p\_current\_value = log\_p\_cand

### node\_normal.py

from node import Node

import logging

import scipy.stats as stats

import math

\_log = logging.getLogger("node\_normal")

class NormalNode(Node):

def \_\_init\_\_(self, value=0, name=None, mean=0, var=1, cand\_var=1, observed=False):

super().\_\_init\_\_(value=value, name=name, cand\_var=cand\_var, observed=observed)

self.mean = mean

self.var = var

self.connect\_to\_parent\_node(mean)

self.connect\_to\_parent\_node(var)

def \_\_str\_\_(self):

return "{} = {}".format(self.pdf\_name, self.current\_value)

@property

def pdf\_name(self):

return "{}({}, {})".format(self.display\_name, Node.parent\_node\_str(self.mean), Node.parent\_node\_str(self.var))

def log\_current\_conditional\_probability(self):

"""

Return probability given current values of 'mean' and 'var'.

(If 'mean' and 'var' are parent nodes, get their current\_value.)

"""

mean = Node.parent\_node\_value(self.mean)

var = Node.parent\_node\_value(self.var)

p = stats.norm.pdf(self.current\_value, mean, math.sqrt(var))

\_log.debug(" p = {}".format(p))

log\_p = (0 if p == 0 else math.log(p))

\_log.debug("p({}={}) = {}".format(self.display\_name, self.current\_value, p))

return log\_p

### node\_invgamma.py

from node import Node

import logging

import scipy.stats as stats

import math

\_log = logging.getLogger("node\_invgamma")

class InvGammaNode(Node):

def \_\_init\_\_(self, value=1, name=None, shape=1, scale=1, cand\_var=1, observed=False):

super().\_\_init\_\_(value=value, name=name, cand\_var=cand\_var, observed=observed)

self.shape = shape

self.scale = scale

if shape is None:

raise ValueError("Parameter 'shape' is required")

if value <= 0:

raise ValueError("Parameter 'value' must be greater than 0.")

self.connect\_to\_parent\_node(shape)

self.connect\_to\_parent\_node(scale)

def \_\_str\_\_(self):

return "{} = {}".format(self.pdf\_name, self.current\_value)

@property

def pdf\_name(self):

return "{}({}, {})".format(self.display\_name, Node.parent\_node\_str(self.shape), Node.parent\_node\_str(self.scale))

def is\_candidate\_in\_domain(self, cand):

return cand > 0

def log\_current\_conditional\_probability(self):

assert(self.current\_value > 0)

shape = Node.parent\_node\_value(self.shape)

scale = Node.parent\_node\_value(self.scale)

p = stats.invgamma.pdf(self.current\_value, a=shape, scale=scale)

log\_p = (0 if p == 0 else math.log(p))

\_log.debug("p({}={}) = {}".format(self.display\_name, self.current\_value, p))

return log\_p

### node\_gamma.py

from node\_invgamma import \*

\_log = logging.getLogger("node\_gamma")

class GammaNode(InvGammaNode):

def \_\_init\_\_(self, value=1, name=None, shape=1, scale=1, shape\_modifier=None, cand\_var=1, observed=False):

super().\_\_init\_\_(value=value, name=name, shape=shape, scale=scale,

cand\_var=cand\_var, observed=observed)

self.shape\_modifier = shape\_modifier

def log\_current\_conditional\_probability(self):

assert(self.current\_value > 0)

shape = Node.parent\_node\_value(self.shape)

scale = Node.parent\_node\_value(self.scale)

if not self.shape\_modifier is None:

shape = self.shape\_modifier(shape)

p = stats.gamma.pdf(self.current\_value, a=shape, scale=1/scale)

log\_p = (0 if p == 0 else math.log(p))

\_log.debug("p({}={}) = {}".format(self.display\_name, self.current\_value, p))

return log\_p

### node\_poisson.py

from node import Node

import logging

import scipy.stats as stats

import math

import random

\_log = logging.getLogger("node\_poisson")

class PoissonNode(Node):

def \_\_init\_\_(self, value=1, name=None, rate=1, cand\_var=1, observed=False):

super().\_\_init\_\_(value=value, name=name, cand\_var=cand\_var, observed=observed)

self.rate = rate

if value <= 0:

raise ValueError("Parameter 'value' must be greater than 0.")

self.connect\_to\_parent\_node(rate)

def \_\_str\_\_(self):

return "{} = {}".format(self.pdf\_name, self.current\_value)

@property

def pdf\_name(self):

return "{}({})".format(self.display\_name, Node.parent\_node\_str(self.rate))

def is\_candidate\_in\_domain(self, cand):

return cand > 0

def select\_candidate(self):

"""For Poisson, use Metropolis with a candidate distribution that rounds samples from a normal."""

return round(random.gauss(self.current\_value, self.cand\_std\_dev), 0)

def log\_current\_conditional\_probability(self):

assert(self.current\_value > 0)

rate = Node.parent\_node\_value(self.rate)

p = stats.poisson.pmf(self.current\_value, mu=rate)

log\_p = (0 if p == 0 else math.log(p))

\_log.debug("p({}={}) = {}".format(self.display\_name, self.current\_value, p))

return log\_p

### node\_beta.py

from node import Node

import logging

import scipy.stats as stats

import math

\_log = logging.getLogger("node\_beta")

class BetaNode(Node):

def \_\_init\_\_(self, value=1, name=None, alpha=1, beta=1, cand\_var=1, observed=False):

super().\_\_init\_\_(value=value, name=name, cand\_var=cand\_var, observed=observed)

self.alpha = alpha

self.beta = beta

if value < 0 or value > 1:

raise ValueError("Parameter 'value' must be greater than 0 and less than 1.")

self.connect\_to\_parent\_node(alpha)

self.connect\_to\_parent\_node(beta)

def \_\_str\_\_(self):

return "{} = {}".format(self.pdf\_name, self.current\_value)

@property

def pdf\_name(self):

return "{}({}, {})".format(self.display\_name, Node.parent\_node\_str(self.alpha), Node.parent\_node\_str(self.beta))

def is\_candidate\_in\_domain(self, cand):

return 0 <= cand <= 1

def log\_current\_conditional\_probability(self):

assert(self.current\_value > 0)

alpha = Node.parent\_node\_value(self.alpha)

beta = Node.parent\_node\_value(self.beta)

p = stats.beta.pdf(self.current\_value, a=alpha, b=beta)

log\_p = (0 if p == 0 else math.log(p))

\_log.debug("p({}={}) = {}".format(self.display\_name, self.current\_value, p))

return log\_p

### node\_bernoulli.py

from node import Node

import logging

import math

import random

\_log = logging.getLogger("node\_bernoulli")

class BernoulliNode(Node):

def \_\_init\_\_(self, value=1, name=None, p=0.5, observed=False):

super().\_\_init\_\_(value=value, name=name, cand\_var=1, observed=observed)

self.p = p

if value < 0 or value > 1:

raise ValueError("Parameter 'value' must be between 0 and 1.")

self.connect\_to\_parent\_node(p)

def \_\_str\_\_(self):

return "{} = {}".format(self.pdf\_name, self.current\_value)

@property

def pdf\_name(self):

return "{}({})".format(self.display\_name, Node.parent\_node\_str(self.p))

def log\_current\_conditional\_probability(self):

"""

For Bernoulli/Binomial, sample directly instead of trying to use Metropolis.

"""

p = Node.parent\_node\_value(self.p)

sample = 1 if random.random() <= p else 0

log\_sample = (0 if sample == 0 else math.log(sample))

\_log.debug("p({}={}) = {}".format(self.display\_name, self.current\_value, sample))

return log\_sample