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



is the average score for Tournament . is the average skill of Golfer (i.e., how much better they score than the average golfer). Given a set of golf scores for several tournaments, we are interested in obtaining the average skill of the golfers ().

Initially, my results were way off because I was using the same initial value for my golfer means as I was using for my tournament means (which I based blindly on the sample code provided, which used a variable named “est\_avg” for both sets of nodes), whereas the golfer means should actually have been *differences* from the average, and an initial value close to 0 would have been much better. With an initial value for golfer means of 72, it would have taken a lot more samples than I was generating in order to get to a valid steady state, as illustrated by the following mixing plots of the golfer means of TigerWoods and VijaySingh:

*Initial golfermean value = 72:*

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

Setting the initial value of all golfer means to 0 significantly improved the results. With this initial value, the following mixing plots show that a good burn value is probably somewhere around 10,000.

*Initial golfermean value = 0:*

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

Here are histograms of 100,000 samples of the golfer means for TigerWoods and VijaySingh, taken after 10,000 burned samples. The values are much better, and the distributions are beginning to look somewhat Gaussian. Still, there is a significant amount of variance in the samples.

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

Here are the rankings I obtained with 100,000 samples after a burn of 10,000:

1: VijaySingh -3.358989; 90% interval: (-3.787266, -2.602100)

2: TigerWoods -3.244745; 90% interval: (-3.942376, -2.531568)

3: PhilMickelson -2.800336; 90% interval: (-3.434281, -1.996117)

4: ErnieEls -2.773805; 90% interval: (-3.683183, -1.937580)

5: StewartCink -2.404671; 90% interval: (-3.085598, -1.819585)

6: SergioGarcia -2.318692; 90% interval: (-3.089609, -1.649872)

7: ScottVerplank -2.273680; 90% interval: (-2.977398, -1.656846)

8: JayHaas -2.231404; 90% interval: (-2.923095, -1.636209)

9: StephenAmes -2.148486; 90% interval: (-2.749085, -1.519959)

10: PadraigHarrington -2.141603; 90% interval: (-3.235765, -0.972317)

…

594: DavidCarr 3.576623; 90% interval: (1.619768, 5.912245)

595: MattLoving 3.577810; 90% interval: (1.700474, 5.970935)

596: CharlesCoody 3.738503; 90% interval: (1.513889, 5.674814)

597: DaveEichelberger 3.793106; 90% interval: (0.439146, 5.877481)

598: RobertDeruntz 3.839784; 90% interval: (1.803449, 6.202557)

599: LorenPersonett 3.866243; 90% interval: (2.116492, 5.170821)

600: TommyAaron 4.082453; 90% interval: (2.121717, 6.472951)

601: KevinSavage 4.763779; 90% interval: (2.779024, 6.713335)

602: JohnAber 4.804098; 90% interval: (2.951387, 6.441183)

603: DerekSanders 5.091835; 90% interval: (3.123192, 7.004599)

604: ArnoldPalmer 5.994113; 90% interval: (4.442667, 7.457914)

### network\_golfers.py

from node\_normal import \*

from node\_invgamma import \*

from network import \*

from operator import itemgetter

data = []

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

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

line\_data[1] = float(line\_data[1]) # parse the score value as a float

data.append(line\_data)

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

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

# For candidate distributions, we use a Normal with mean 0, variance 1

hypertournmean\_candsd = 1 # variance

hypervar\_candsd = 1 # variance

mean\_candsd = 1 # variance

obsvar\_candsd = 1 # variance

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

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

tournmean = {}

for tourn in tourns:

tournmean[tourn] = NormalNode(72, 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(0, name=golfer, cand\_var=mean\_candsd, mean=0, var=hypergolfervar)

obsvar = InvGammaNode(8.5, 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 = 10000

network = Network(

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

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

samples.plot\_node(golfermean['VijaySingh'])

samples.plot\_node(golfermean['TigerWoods'])

samples.plot\_histogram\_for\_node(golfermean['VijaySingh'])

samples.plot\_histogram\_for\_node(golfermean['TigerWoods'])

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

It took me a couple of tries to get my Bernoulli distribution implemented correctly, as I confused the sampling value used for the candidate with the probability of the candidate.

When G is observed, the most obvious difference is in the Poisson distribution of node F, which is much more likely to have values of 2, 3, and 4, and less likely to have a value of 1.

*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:

IMPOSSIBLE = math.log(0.000000000001)

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

return Node.parent\_node\_value(self.var) > 0

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)

if var == 0:

\_log.debug("Node " + str(self) + ": Cannot compute a normal probability when variance is 0.")

return Node.IMPOSSIBLE

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

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

log\_p = (Node.IMPOSSIBLE 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 = (Node.IMPOSSIBLE 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 = (Node.IMPOSSIBLE 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 = (Node.IMPOSSIBLE 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 = (Node.IMPOSSIBLE 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

import scipy.stats as stats

\_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)

if not isinstance(p, list):

p = [p]

self.p = p

if value < 0 or value > 1:

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

for parent in self.p:

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

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 select\_candidate(self):

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

sample = 1 if random.random() <= 0.5 else 0

return sample

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

"""

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

"""

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

p = stats.bernoulli.pmf(self.current\_value, param\_p)

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

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

return log\_p