Nathan Hadfield

CS 677 (Spring 2014)

# MCMC Part 1: Boolean

## Burglar Alarm Network

(100,000 samples after 10,000 burns)

P(B=False | J=True, M=True) = 0.71116

P(B=True | J=True, M=True) = 0.28884

P(A=True | J=True, M=True) = 0.75775

P(E=True | J=True, M=True) = 0.17268

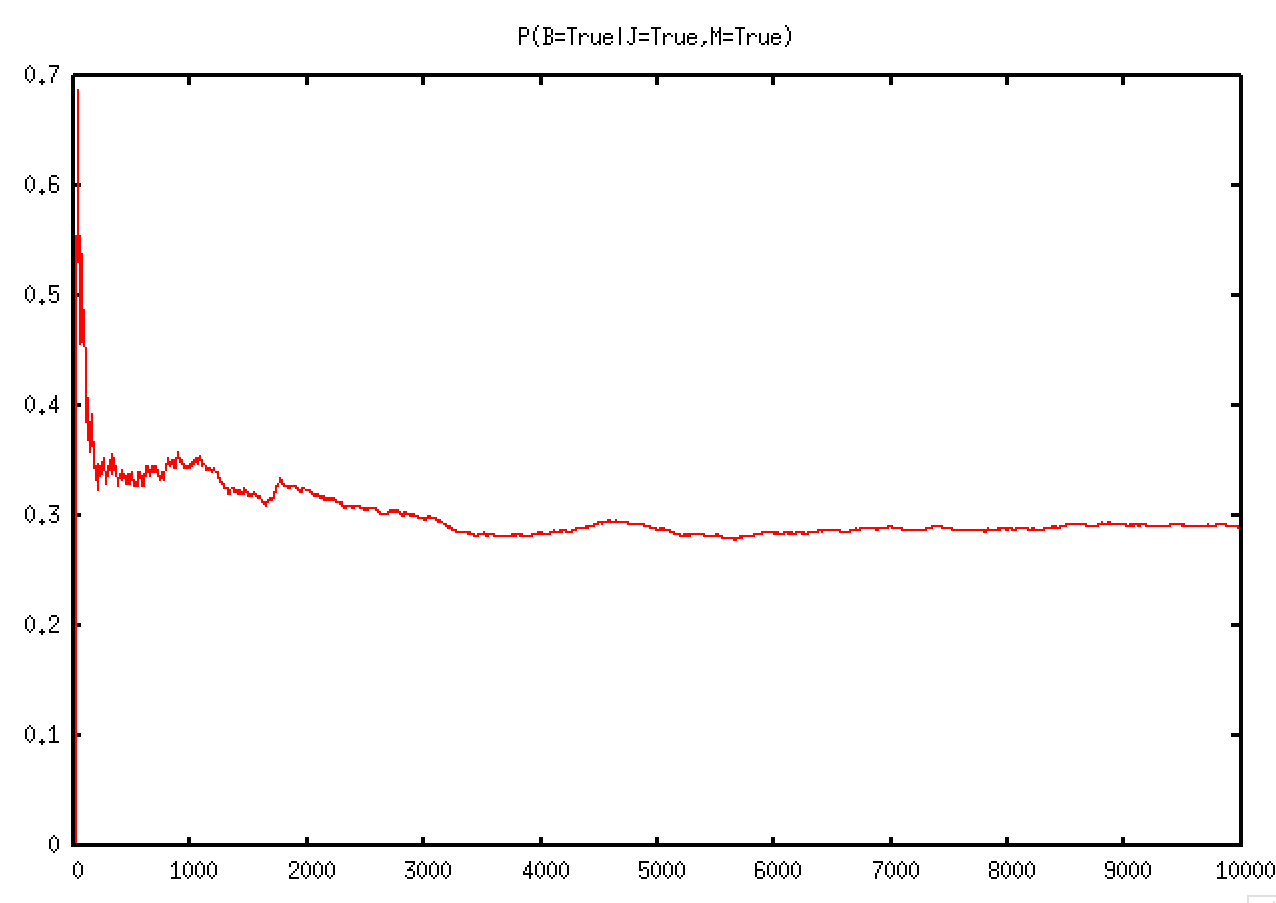
P(B=True | J=False, M=False) = 0.0001

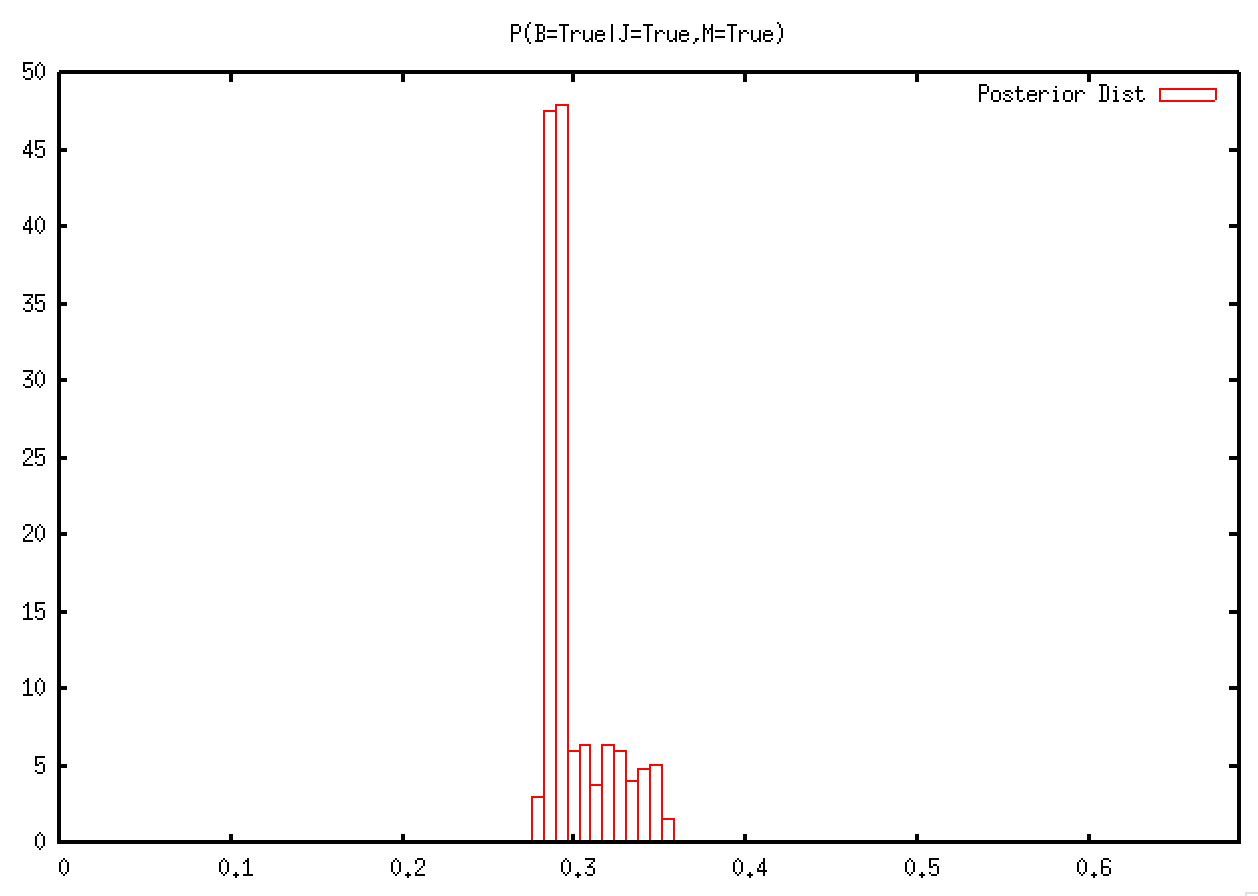
P(B=True | J=True, M=False) = 0.00619

P(B=True | J=True) = 0.01463

P(B=True | M=True) = 0.05864

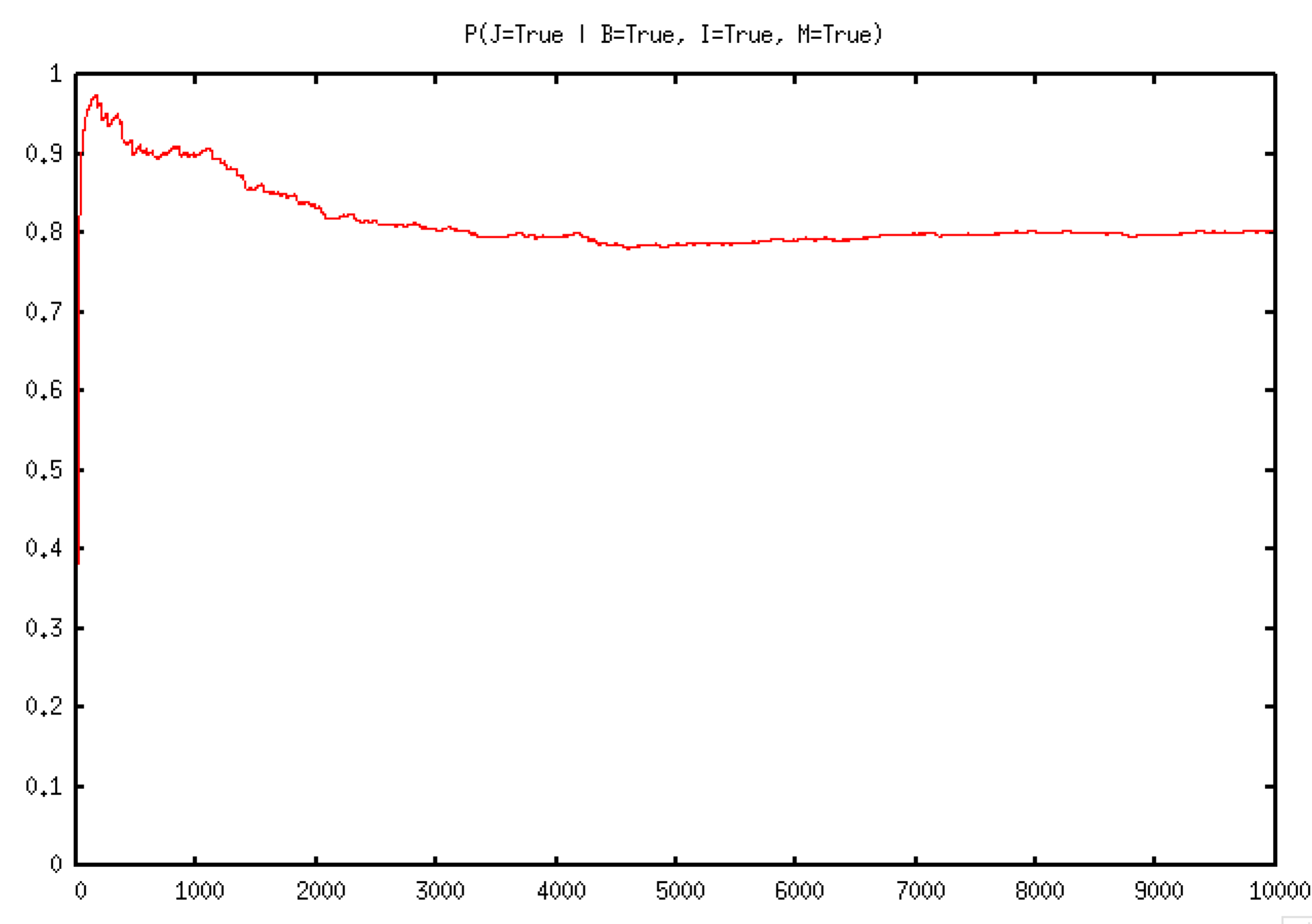
Plots show results of 10,000 samples without burn.





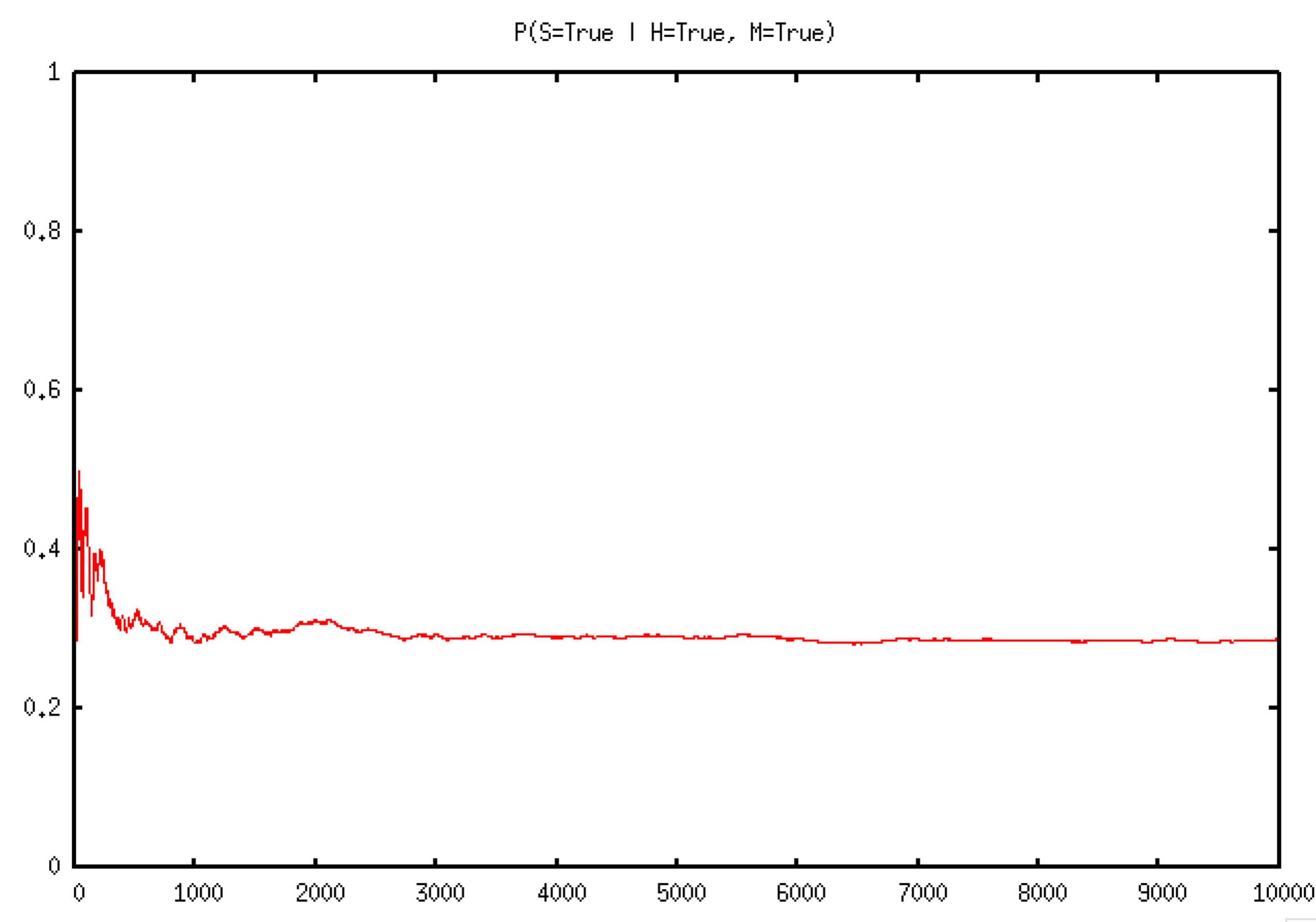
## Conviction Network (From Daniel)

P(J=True | B=True, I=True, M=True) = 0.80717



## Sleep Network (My Own)

P(S=True | H=True, M=True) = 0.28724



## Implementation

My implementation consists of three Python modules:

* **mcmc.py**: Provides the model setup.
* **nodes.py**: Defines an abstract class *Node* for representing a Bayesian network node, and a subclass *BernoulliNode* for nodes that can have a True or False value.
* **network.py**: Defines a *Network* class that contains the network nodes and generates samples, and a *SampleProcessor* class that stores the sampling results and uses them to evaluate probabilities and generate plots.

### mcmc.py

from nodes import \*

from network import \*

import logging

log = logging.getLogger("mcmc")

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

logging.getLogger().setLevel(logging.ERROR)

b = BernoulliNode(name='B', prob=[0.001], value=False)

a = BernoulliNode(name='A', prob=[0.95, 0.94, 0.29, 0.001], value=False)

e = BernoulliNode(name='E', prob=[0.002], value=False)

j = BernoulliNode(name='J', prob=[0.90, 0.05], value=True)

m = BernoulliNode(name='M', prob=[0.70, 0.01], value=True)

b.children = [a]

e.children = [a]

a.parents = [b, e]

a.children = [j, m]

j.parents = [a]

m.parents = [a]

j.is\_observed = True

m.is\_observed = True

network = Network(nodes=[b, e, a, j, m])

samples = network.collect\_samples(burn=0, n=10000)

log.info("Totals: " + str(samples.totals()))

print("P(B=False | J=True, M=True) = " + str(samples.p({b: False}, {j: True, m: True})))

print("P(B=True | J=True, M=True) = " + str(samples.p({b: True}, {j: True, m: True})))

print("P(A=True | J=True, M=True) = " + str(samples.p({a: True}, {j: True, m: True})))

print("P(E=True | J=True, M=True) = " + str(samples.p({e: True}, {j: True, m: True})))

samples.plot\_mixing("P(B=True|J=True,M=True)", {b: True}, {j: True, m: True})

samples.plot\_histogram("P(B=True|J=True,M=True)", {b: True}, {j: True, m: True})

# Different observed nodes; have to resample

network = Network(nodes=[b, e, a, j, m])

samples = network.collect\_samples(burn=10000, n=100000)

j.current\_value = False

m.current\_value = False

samples = network.collect\_samples(burn=10000, n=100000)

print("P(B=True | J=False, M=False) = " + str(samples.p({b: True}, {j: False, m: False})))

j.current\_value = True

m.current\_value = False

samples = network.collect\_samples(burn=10000, n=100000)

print("P(B=True | J=True, M=False) = " + str(samples.p({b: True}, {j: True, m: False})))

j.is\_observed = True

j.current\_value = True

m.is\_observed = False

samples = network.collect\_samples(burn=10000, n=100000)

print("P(B=True | J=True) = " + str(samples.p({b: True}, {j: True})))

j.is\_observed = False

m.is\_observed = True

m.current\_value = True

samples = network.collect\_samples(burn=10000, n=100000)

print("P(B=True | M=True) = " + str(samples.p({b: True}, {m: True})))

### nodes.py

import random

import logging

log = logging.getLogger("nodes")

class Node:

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

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

def \_\_init\_\_(self, name=None, value=None, parents=[], children=[], is\_observed=False):

self.name = name

self.parents = parents

self.children = children

self.current\_value = value

self.is\_observed = is\_observed

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

return self.display\_name()

@property

def node\_type(self):

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

@property

def display\_name(self):

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

def sample(self):

"""

Set current\_value according to probability given values of all other nodes

Subclasses must implement this method.

"""

raise NotImplementedError

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

"""

Subclasses must implement this method.

"""

raise NotImplementedError

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

p = 1.0

for node in self.children + [self]:

p \*= node.current\_conditional\_probability()

return p

def current\_conditional\_probability(self):

"""

Compute the probability of the current value of this node conditional on the current values of its parents

"""

parent\_values = dict((node, node.current\_value) for node in self.parents)

p = self.probability\_of\_event(parent\_values)

# p is the probability of the current value being true. If the current

# value is actually false, the probability is 1-p.

if not self.current\_value:

p = 1 - p

return p

class BernoulliNode(Node):

def \_\_init\_\_(self, name, value=True, parents=[], children=[], prob=None):

super().\_\_init\_\_(name, value, parents, children)

self.prob = prob

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

val = self.display\_name + "(" + str(self.prob) + ") = " + str(self.current\_value)

return val

def probability\_of\_event(self, event):

"""

Calculate probability of node/values given in event dict.

Nodes must be contained within 'parents' dictionary.

Probability table has 2^n rows. E.g., if parents are A, B:

A=true, B=true = prob[0]

A=true, B=false = prob[1]

A=false, B=true = prob[2]

A=false, B=false = prob[3]

"""

assert len(self.prob) == 2\*\*len(self.parents), \

"Prob table for Bernoulli node '" + self.display\_name + "' does not have enough entries for its " \

+ str(len(self.parents)) + " parents."

table\_idx = 0

for parent\_node in self.parents:

table\_idx \*= 2

parent\_event = event[parent\_node]

if parent\_event:

assert isinstance(parent\_event, bool), "Current value '" + str(parent\_event) \

+ "' of parent '" + parent\_node.display\_name \

+ "' of Bernoulli node '" + self.display\_name \

+ "' is not a boolean."

table\_idx += 1

assert table\_idx < len(self.prob)

table\_idx = len(self.prob)-1 - table\_idx # reverse the index to make the first item map to the first node

p = self.prob[table\_idx]

return p

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

"""

Compute the probability of this node given the probability of all the other nodes.

Only have to calculate probabilities for nodes in the Markov Blanket (parents, children,

parents of children),

by dividing the conditional probability of its current value by its marginal probability.

"""

saved\_value = self.current\_value

num = self.current\_unnormalized\_mb\_probability()

# calculate marginal probability by adding the current value (True/False) with its opposite (False/True)

self.current\_value = not self.current\_value

denom = num + self.current\_unnormalized\_mb\_probability()

self.current\_value = saved\_value

return num/denom

def sample(self):

if not self.is\_observed:

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

# If current value is false, then the probability we calculated is the probability

# of the node being false. We want the probability of the node being true.

if self.current\_value is False:

p = 1-p

r = random.random()

self.current\_value = (r < p)

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

### network.py

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 sample\_generator(self):

"""Create samples from the given nodes"""

while True:

for test\_node in self.nodes:

test\_node.sample()

network\_state = []

for node in self.nodes:

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

yield network\_state

def collect\_samples(self, burn, n):

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

progress\_step = (burn + n) / 10

cur\_sample = 0

mcmc = self.sample\_generator()

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 is\_sample\_match(self, sample, event):

is\_match = False

for idx, node in enumerate(self.nodes):

if node in event:

if sample[idx] != event[node]:

break

else:

is\_match = True

return is\_match

def totals(self, start=None, end=None):

if start is None:

start = 0

if end is None:

end = len(self.samples)

num\_nodes = len(self.nodes)

totals = [0] \* num\_nodes

for i in range(start, end):

sample = self.samples[i]

for idx in range(num\_nodes):

if sample[idx]:

totals[idx] += 1

return totals

def p(self, outcomes, givens, start=None, end=None):

"""

:param outcome: dictionary of nodes and values

:param given: dictionary of nodes and values

:return: probability (float in range[0..1])

"""

if start is None:

start = 0

if end is None:

end = len(self.samples)

matching\_givens\_count = 0

matching\_outcomes\_count = 0

outcomes\_and\_givens = {}

for d in [outcomes, givens]:

outcomes\_and\_givens.update(d)

for i in range(start, end):

sample = self.samples[i]

if self.is\_sample\_match(sample, givens):

matching\_givens\_count += 1

if self.is\_sample\_match(sample, outcomes\_and\_givens):

matching\_outcomes\_count += 1

p = 0 if matching\_givens\_count == 0 else matching\_outcomes\_count / matching\_givens\_count

return p

def plot\_mixing(self, name, outcomes, givens):

prob\_samples = [self.p(outcomes, givens, 0, i) for i in range(len(self.samples))]

p = evilplot.Plot(title=u"{0:s}".format(name))

points = evilplot.Points(list(enumerate(prob\_samples)))

points.style = 'lines'

points.linewidth = 1

p.append(points)

#p.write\_gpi('plots/mix-%s.gpi' % name)

p.show()

def plot\_histogram(self, name, outcomes, givens):

prob\_samples = [self.p(outcomes, givens, 0, i) for i in range(len(self.samples))]

p = evilplot.Plot(title=u"{0:s}".format(name))

postd = evilplot.Histogram(prob\_samples, 100, normalize=True)

postd.title = 'Posterior Dist'

p.append(postd)

p.show()

## Unit Tests

### test\_nodes.py

from unittest import TestCase

from nodes import \*

class TestBernoulliNode(TestCase):

def setUp(self):

self.b = BernoulliNode(name='B', prob=[0.001])

self.a = BernoulliNode(name='A', prob=[0.95, 0.94, 0.29, 0.001])

self.e = BernoulliNode(name='E', prob=[0.002])

self.j = BernoulliNode(name='J', prob=[0.90, 0.05])

self.m = BernoulliNode(name='M', prob=[0.70, 0.01])

self.b.children = [self.a]

self.e.children = [self.a]

self.a.parents = [self.b, self.e]

self.a.children = [self.j, self.m]

self.j.parents = [self.a]

self.m.parents = [self.a]

def test\_probability\_of\_event(self):

self.assertEquals(0.95, self.a.probability\_of\_event({self.b: True, self.e: True}),

"Incorrect probability lookup.")

self.assertEquals(0.94, self.a.probability\_of\_event({self.b: True, self.e: False}),

"Incorrect probability lookup.")

self.assertEquals(0.29, self.a.probability\_of\_event({self.b: False, self.e: True}),

"Incorrect probability lookup.")

self.assertEquals(0.001, self.a.probability\_of\_event({self.b: False, self.e: False}),

"Incorrect probability lookup.")

self.assertEquals(0.001, self.b.probability\_of\_event({}),

"Incorrect probability lookup.")

self.assertEquals(0.001, self.b.probability\_of\_event({self.b: False, self.e: False}),

"Incorrect probability lookup.")

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

self.b.current\_value = True

self.e.current\_value = True

self.a.current\_value = True

self.assertEquals(0.95, self.a.current\_conditional\_probability(),

"Incorrect conditional probability given current values of node and its parents.")

self.e.current\_value = False

self.assertEquals(0.94, self.a.current\_conditional\_probability(),

"Incorrect conditional probability given current values of node and its parents.")

self.a.current\_value = False

self.assertEquals(1-0.94, self.a.current\_conditional\_probability(),

"Incorrect conditional probability given current values of node and its parents.")

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

# initially all nodes are True

# 0.95\*0.90\*0.70 = 0.5985

self.assertAlmostEqual(0.95\*0.90\*0.70, self.a.current\_unnormalized\_mb\_probability(), places=20)

self.b.current\_value = False

self.e.current\_value = False

self.a.current\_value = True

self.j.current\_value = False

self.m.current\_value = False

# 0.001\*(1-0.90)\*(1-0.70) = 0.00003

self.assertAlmostEqual(0.001\*(1-0.90)\*(1-0.70), self.a.current\_unnormalized\_mb\_probability(), places=20)

self.a.current\_value = False

# (1-0.001)\*(1-0.05)\*(1-0.01) = 0.9395595

self.assertAlmostEqual((1-0.001)\*(1-0.05)\*(1-0.01), self.a.current\_unnormalized\_mb\_probability(), places=20)

self.b.current\_value = True

self.e.current\_value = False

self.a.current\_value = True

self.j.current\_value = False

self.m.current\_value = False

# 0.001\*0.94 = 0.00094

self.assertAlmostEqual(0.001\*0.94, self.b.current\_unnormalized\_mb\_probability(), places=20)

self.b.current\_value = False

# (1-0.001)\*0.001 = 0.000999

self.assertAlmostEqual((1-0.001)\*0.001, self.b.current\_unnormalized\_mb\_probability(), places=20)

self.b.current\_value = False

self.e.current\_value = False

self.a.current\_value = False

self.j.current\_value = False

self.m.current\_value = False

# (1-0.001)\*(1-0.001) = 0.998001

self.assertAlmostEqual((1-0.001)\*(1-0.001), self.b.current\_unnormalized\_mb\_probability(), places=20)

self.b.current\_value = True

self.e.current\_value = False

self.a.current\_value = False

self.j.current\_value = False

self.m.current\_value = False

# 0.001\*(1-0.94) = 0.0006

self.assertAlmostEqual(0.001\*(1-0.94), self.b.current\_unnormalized\_mb\_probability(), places=20)

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

# initially all nodes are True

p\_b\_true = 0.95 \* 0.001

p\_b\_false = 0.29 \* (1-0.001)

p = p\_b\_true / (p\_b\_true + p\_b\_false)

self.assertAlmostEqual(p, self.b.probability\_of\_current\_value\_given\_other\_nodes(), places=20)

p\_a\_true = 0.95 \* 0.90 \* 0.70

p\_a\_false = (1-0.95) \* 0.05 \* 0.01

p = p\_a\_true / (p\_a\_true + p\_a\_false)

self.assertAlmostEqual(p, self.a.probability\_of\_current\_value\_given\_other\_nodes(), places=20)

### test\_samples\_processor.py

from unittest import TestCase

from network import \*

from nodes import \*

import textwrap

class TestSamplesProcessor(TestCase):

def setUp(self):

self.a = BernoulliNode(name='A', prob=[0.5])

self.b = BernoulliNode(name='B', prob=[0.2])

self.c = BernoulliNode(name='C', prob=[0.7])

self.samples = [(True, False, False),

(True, False, True),

(False, False, True),

(False, True, True),

(True, False, True)]

self.processor = SamplesProcessor([self.a, self.b, self.c], self.samples)

def test\_str(self):

samples\_str = textwrap.dedent("""\

A, B, C

True, False, False

True, False, True

False, False, True

False, True, True

True, False, True""")

self.assertEqual(samples\_str, str(self.processor), "Incorrect string representation")

def test\_is\_sample\_match(self):

self.assertTrue(self.processor.is\_sample\_match([True, False, False], {self.a: True, self.b: False, self.c: False}))

self.assertFalse(self.processor.is\_sample\_match([True, False, False], {self.a: True, self.b: False, self.c: True}))

self.assertTrue(self.processor.is\_sample\_match([True, False, False], {self.a: True, self.c: False}))

self.assertFalse(self.processor.is\_sample\_match([True, False, False], {self.a: True, self.c: True}))

self.assertFalse(self.processor.is\_sample\_match([True, False, False], {self.a: True, self.c: True}))

self.assertTrue(self.processor.is\_sample\_match([True, False, False], {self.a: True}))

self.assertTrue(self.processor.is\_sample\_match([True, False, False], {}))

def test\_p(self):

self.assertEquals(1/4, self.processor.p({self.a: False, self.b: True}, {self.c: True}))

self.assertEquals(2/3, self.processor.p({self.a: True}, {self.b: False, self.c: True}))

self.assertEquals(3/5, self.processor.p({self.a: True}, {}))