Nathan Hadfield
CS 677 (Spring 2014)

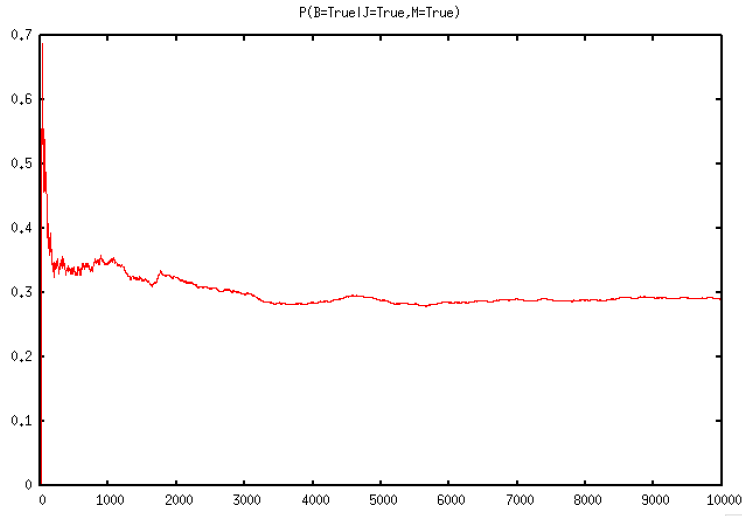
MCMC Part 1: Boolean

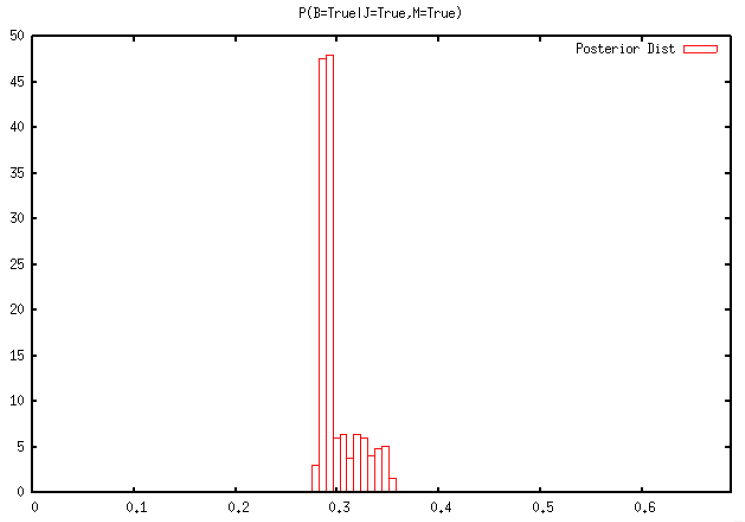
Burglar Alarm Network

(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
```

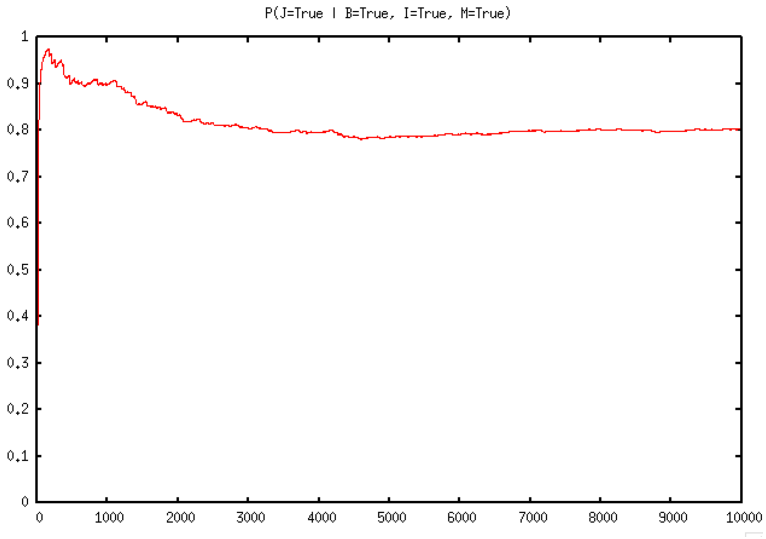
Plots show results of 10,000 samples without burn.





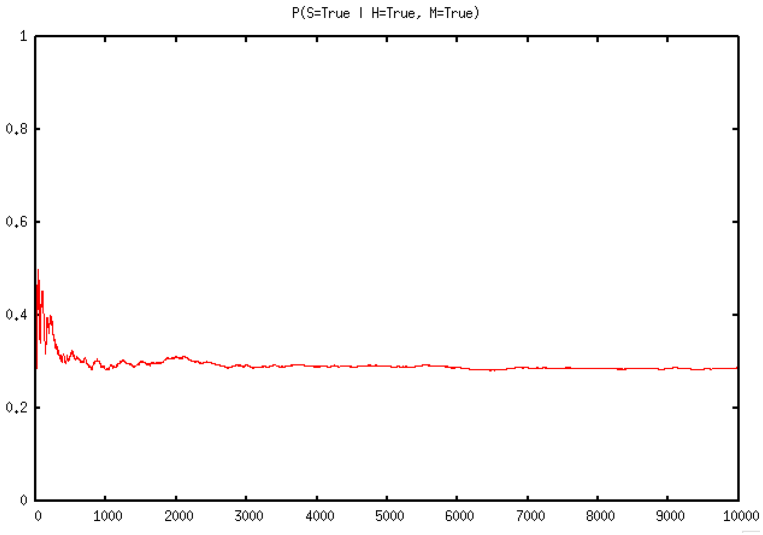
Conviction Network (From Daniel)

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



Sleep Network (My Own)

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



Why I Don't Get Enough Sleep

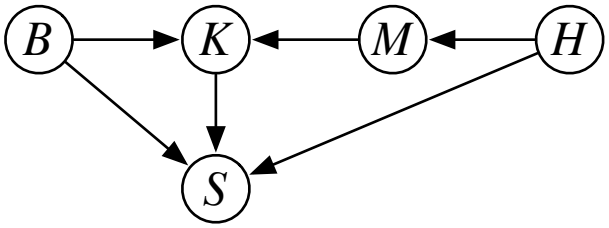
M	*B*	*P(K)*
t	*t*	*0.5*
t	*f*	*0.3*
f	*t*	*0.3*
f	*f*	*0.2*

H	*P(M)*
t	*0.14*
f	*0.3*

P(H)
0.6

P(B)
0.3

B	*K*	*H*	*P(S)*
t	*t*	*t*	*0.05*
t	*t*	*f*	*0.1*
t	*f*	*t*	*0.1*
t	*f*	*f*	*0.5*
f	*t*	*t*	*0.1*
f	*t*	*f*	*0.4*
f	*f*	*t*	*0.5*
f	*f*	*f*	*0.8*



B = DogsBark, K = KidsWake, M = KidsWatchMovie, H = HomeworkDue, S = GoodSleep

If the dogs bark, they might wake up the kids, and they might keep me from sleeping. If the kids wake up, I'm probably not going to get good sleep. If the kids watch a movie they're more likely to have a bad dream and wake up. If I don't have homework due, there's a better chance the kids will watch a movie (with me). If I do have homework due, I'm unlikely to get good sleep.

I'd like to find the probability of getting a good night's sleep if I have homework and the kids watch a movie: P(S=t|H=t,M=t)

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}, {}))
```