Project 5

Thomas Moore

1. Building a bayesian network
2. Predict the *probability* that a patient has lung cancer using prior sampling
3. Predict if a patient has lung cancer given he has bronchitis and positive X-ray using the following mehod:

   a) rejection sampling

   b) likelihood weighting

   c) enumeration

Defining Classes and Functions

```
from probability import *



# probability distribution class

class ProbDist:
    """A discrete probability distribution. You name the random variable
    in the constructor, then assign and query probability of values.
    >>> P = ProbDist('Flip'); P['H'], P['T'] = 0.25, 0.75; P['H']
    0.25
    >>> P = ProbDist('X', {'lo': 125, 'med': 375, 'hi': 500})
    >>> P['lo'], P['med'], P['hi']
    (0.125, 0.375, 0.5)
    """

    def __init__(self, varname='?', freqs=None):
        """If freqs is given, it is a dictionary of values - frequency pairs,
        then ProbDist is normalized."""
        self.prob = {}
        self.varname = varname
        self.values = []
        if freqs:
            for (v, p) in freqs.items():
                self[v] = p
            self.normalize()
```

```
    def __getitem__(self, val):
        """Given a value, return P(value)."""
        try:
            return self.prob[val]
        except KeyError:
            return 0

    def __setitem__(self, val, p):
        """Set P(val) = p."""
        if val not in self.values:
            self.values.append(val)
        self.prob[val] = p

    def normalize(self):
        """Make sure the probabilities of all values sum to 1.
        Returns the normalized distribution.
        Raises a ZeroDivisionError if the sum of the values is 0."""
        total = sum(self.prob.values())
        if not isclose(total, 1.0):
            for val in self.prob:
                self.prob[val] /= total
        return self

    def show_approx(self, numfmt='{:.3g}'):
        """Show the probabilities rounded and sorted by key, for the
        sake of portable doctests."""
        return ', '.join([('{}: ' + numfmt).format(v, p)
                          for (v, p) in sorted(self.prob.items())])

    def __repr__(self):
        return "P({})".format(self.varname)


# defining Bayes node class
class BayesNode:
    """A conditional probability distribution for a boolean variable,
    P(X | parents). Part of a BayesNet."""

    def __init__(self, X, parents, cpt):
        """X is a variable name, and parents a sequence of variable
        names or a space-separated string.  cpt, the conditional
        probability table, takes one of these forms:

        * A number, the unconditional probability P(X=true). You can
          use this form when there are no parents.

        * A dict {v: p, ...}, the conditional probability distribution
          P(X=true | parent=v) = p. When there's just one parent.

        * A dict {(v1, v2, ...): p, ...}, the distribution P(X=true |
          parent1=v1, parent2=v2, ...) = p. Each key must have as many
```

```
      values as there are parents. You can use this form always;
      the first two are just conveniences.

    In all cases the probability of X being false is left implicit,
    since it follows from P(X=true).

    >>> X = BayesNode('X', '', 0.2)
    >>> Y = BayesNode('Y', 'P', {T: 0.2, F: 0.7})
    >>> Z = BayesNode('Z', 'P Q',
    ...    {(T, T): 0.2, (T, F): 0.3, (F, T): 0.5, (F, F): 0.7})
    """
    if isinstance(parents, str):
        parents = parents.split()

    # We store the table always in the third form above.
    if isinstance(cpt, (float, int)):  # no parents, 0-tuple
        cpt = {(): cpt}
    elif isinstance(cpt, dict):
        # one parent, 1-tuple
        if cpt and isinstance(list(cpt.keys())[0], bool):
            cpt = {(v,): p for v, p in cpt.items()}

    assert isinstance(cpt, dict)
    for vs, p in cpt.items():
        assert isinstance(vs, tuple) and len(vs) == len(parents)
        assert all(isinstance(v, bool) for v in vs)
        assert 0 <= p <= 1

    self.variable = X
    self.parents = parents
    self.cpt = cpt
    self.children = []

def p(self, value, event):
    """Return the conditional probability
    P(X=value | parents=parent_values), where parent_values
    are the values of parents in event. (event must assign each
    parent a value.)
    >>> bn = BayesNode('X', 'Burglary', {T: 0.2, F: 0.625})
    >>> bn.p(False, {'Burglary': False, 'Earthquake': True})
    0.375"""
    assert isinstance(value, bool)
    ptrue = self.cpt[event_values(event, self.parents)]
    return ptrue if value else 1 - ptrue

def sample(self, event):
    """Sample from the distribution for this variable conditioned
    on event's values for parent_variables. That is, return True/False
    at random according with the conditional probability given the
    parents."""
    return probability(self.p(True, event))
```

```
    def __repr__(self):
        return repr((self.variable, ' '.join(self.parents)))


# defining bayes net class

class BayesNet:
    """Bayesian network containing only boolean-variable nodes."""

    def __init__(self, node_specs=None):
        """Nodes must be ordered with parents before children."""
        self.nodes = []
        self.variables = []
        node_specs = node_specs or []
        for node_spec in node_specs:
            self.add(node_spec)

    def add(self, node_spec):
        """Add a node to the net. Its parents must already be in the
        net, and its variable must not."""
        node = BayesNode(*node_spec)
        assert node.variable not in self.variables
        assert all((parent in self.variables) for parent in node.parents)
        self.nodes.append(node)
        self.variables.append(node.variable)
        for parent in node.parents:
            self.variable_node(parent).children.append(node)

    def variable_node(self, var):
        """Return the node for the variable named var.
        >>> burglary.variable_node('Burglary').variable
        'Burglary'"""
        for n in self.nodes:
            if n.variable == var:
                return n
        raise Exception("No such variable: {}".format(var))

    def variable_values(self, var):
        """Return the domain of var."""
        return [True, False]

    def __repr__(self):
        return 'BayesNet({0!r})'.format(self.nodes)


# defining functions

def enumerate_all(variables, e, bn):
    """Return the sum of those entries in P(variables | e{others})
    consistent with e, where P is the joint distribution represented
    by bn, and e{others} means e restricted to bn's other variables
```

```
    (the ones other than variables). Parents must precede children in variables."""
    if not variables:
        return 1.0
    Y, rest = variables[0], variables[1:]
    Ynode = bn.variable_node(Y)
    if Y in e:
        return Ynode.p(e[Y], e) * enumerate_all(rest, e, bn)
    else:
        return sum(Ynode.p(y, e) * enumerate_all(rest, extend(e, Y, y), bn)
                   for y in bn.variable_values(Y))

def enumeration_ask(X, e, bn):
    """Return the conditional probability distribution of variable X
    given evidence e, from BayesNet bn. [Figure 14.9]
    >>> enumeration_ask('Burglary', dict(JohnCalls=T, MaryCalls=T), burglary
    ...  ).show_approx()
    'False: 0.716, True: 0.284'"""
    assert X not in e, "Query variable must be distinct from evidence"
    Q = ProbDist(X)
    for xi in bn.variable_values(X):
        Q[xi] = enumerate_all(bn.variables, extend(e, X, xi), bn)
    return Q.normalize()



# REJECTION

def rejection_sampling(X, e, bn, N=10000):
    """Estimate the probability distribution of variable X given
    evidence e in BayesNet bn, using N samples.  [Figure 14.14]
    Raises a ZeroDivisionError if all the N samples are rejected,
    i.e., inconsistent with e.
    >>> random.seed(47)
    >>> rejection_sampling('Burglary', dict(JohnCalls=T, MaryCalls=T),
    ...   burglary, 10000).show_approx()
    'False: 0.7, True: 0.3'
    """
    counts = {x: 0 for x in bn.variable_values(X)}
    for j in range(N):
        sample = prior_sample(bn)
        if consistent_with(sample, e):
            counts[sample[X]] += 1
    return ProbDist(X, counts)

def consistent_with(event, evidence):
    """Is event consistent with the given evidence?"""
    return all(evidence.get(k, v) == v
               for k, v in event.items())

def sample(self, event):
        """Sample from the distribution for this variable conditioned
```

```
        on event's values for parent_variables. That is, return True/False
        at random according with the conditional probability given the
        parents."""
        return probability(self.p(True, event))

def prior_sample(bn):
    """Randomly sample from bn's full joint distribution. The result
    is a {variable: value} dict. [Figure 14.13]"""
    event = {}
    for node in bn.nodes:
        event[node.variable] = node.sample(event)
    return event




# WEIGHTED

def weighted_sample(bn, e):
    """Sample an event from bn that's consistent with the evidence e;
    return the event and its weight, the likelihood that the event
    accords to the evidence."""
    w = 1
    event = dict(e)
    for node in bn.nodes:
        Xi = node.variable
        if Xi in e:
            w *= node.p(e[Xi], event)
        else:
            event[Xi] = node.sample(event)
    return event, w

def likelihood_weighting(X, e, bn, N=10000):
    """Estimate the probability distribution of variable X given
    evidence e in BayesNet bn.  [Figure 14.15]
    >>> random.seed(1017)
    >>> likelihood_weighting('Burglary', dict(JohnCalls=T, MaryCalls=T),
    ...   burglary, 10000).show_approx()
    'False: 0.702, True: 0.298'
    """
    W = {x: 0 for x in bn.variable_values(X)}
    for j in range(N):
        sample, weight = weighted_sample(bn, e)
        W[sample[X]] += weight
    return ProbDist(X, W)


def probability(p):
    """Return true with probability p."""
    return p > random.uniform(0.0, 1.0)
```

```
def event_values(event, variables):
    """Return a tuple of the values of variables in event.
    >>> event_values ({'A': 10, 'B': 9, 'C': 8}, ['C', 'A'])
    (8, 10)
    >>> event_values ((1, 2), ['C', 'A'])
    (1, 2)
    """
    if isinstance(event, tuple) and len(event) == len(variables):
        return event
    else:
        return tuple([event[var] for var in variables])


def isclose(a, b, rel_tol=1e-9, abs_tol=0.0):
    return abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)




def extend(s, var, val):
    """Copy dict s and extend it by setting var to val; return copy."""
    return {**s, var: val}
```

Building the Bayesian Network

```
import random


# making the network with the functions defined above
visit_to_asia = BayesNode('Visit to Asia?', '', 0.01)


smoker = BayesNode('Smoker?', '', 0.5)
tuberculosis = BayesNode('Tuberculosis?', ['Visit to Asia?'],
                         {True: 0.05, False: 0.01 })
lung_cancer = BayesNode('Lung Cancer', ['Smoker?'], {True: 0.1, False: 0.01})
broncitis = BayesNode('Broncitis', ['Smoker?'], {True: 0.6, False: 0.3})
tb_or_lc = BayesNode('tb_or_lc', ['Tuberculosis?',
                                  'Lung Cancer'],
                     {(True, True): 1,(True, False): 1,
                      (False, True): 1, (False, False): 0})
positive_x_ray = BayesNode('Positive X-ray', ['tb_or_lc'],
                           {True: 0.98, False: 0.05})
dispnea = BayesNode('Dispnea', ['tb_or_lc', 'Broncitis'],
                    {(True, True): 0.9,(True, False): 0.7,
                     (False, True): 0.8, (False, False): 0.1})
```

```
lung_cancer = BayesNet([('Visit to Asia?', '', 0.01),
                ('Smoker?', '', 0.5),
                ('Tuberculosis?', ['Visit to Asia?'],
                 {True: 0.05, False: 0.01 }),
                ('Lung Cancer', ['Smoker?'],
                 {True: 0.1, False: 0.01}),
                ('Broncitis', ['Smoker?'],
                 {True: 0.6, False: 0.3}),
                ('tb_or_lc', ['Tuberculosis?', 'Lung Cancer'],
                 {(True, True): 1, (True, False): 1,
                  (False, True): 1,
                  (False,False): 0}),

                ('Positive X-ray', ['tb_or_lc'], {True: 0.98, False: 0.05}),
                ('Dispnea', ['tb_or_lc', 'Broncitis'],
                 {(True, True): 0.9,(True, False): 0.7,
                  (False, True): 0.8, (False, False): 0.1})
                ])


lw = likelihood_weighting('Lung Cancer',
                          dict(Broncitis = True), lung_cancer, 100).show_approx()


p = rejection_sampling('Lung Cancer', dict(Broncitis = True), lung_cancer, 100)


enum_ans = enumeration_ask('Lung Cancer',
                           {'Broncitis': True, 'Positive X-ray': True}, lung_cancer)
enum_ans[True]
```

```
     0.5528024667561036
```


0s
completed at 3:32 PM