Project 3 Report Runhua Gao

Program Hierarchy Description:

Net Parser:
XmlFile(stands for a bayesnet) -> xmlParser.py(Parser File and output a bayesnet object)

Algorithm:
Input(query variable, evidence, net, N(times for sampling algorithm)) -> output a True and False normalized probability distribution

Basic Class:
bayesnet(naïve baynes net class):
Property:
variablesroom: stores all random variables of the net in key,value of dictionary as the key is variable's name and value is the variable(make it easy to use variable from out environment)
nodes: return all net nodes in topological order, an important property in both enumeration and sampling algorithms
eg: extract the random variable from net: if you want to extract variable "A" in net1, and you have the net1 object, just use it as following:
A = net1["A"]

rv(random variable):
Property:
self.name: random variable's name, str
self.domain: random variable's value space, typically consists of True and False for this project
self.childNodes, self.children: store the node's childNodes in net in class of rv and str separately(self.children stores all its childNodes'name)
self.parentNodes,self.parents: same as above, store node's parent nodes in rv and str(child.parents stores variable's parentnode name)
self.table: variable's probability distribution according to the xml file

Methods:

addChildNode:(RV),:add a variable to its childNode space

addParentNode(RV): add a variable to its parentNode space

kthproba: return kth value in variable's domain

domainsize: return the size of variable's domain space

addProba(Table): bind variable to a table that stores the information of its probability information

evidence_values(evidence): extract information from a evidence(in class of dict), return a new dictionary only stores the value of it's parentNodes' value

eg: in alarm net, J.evidence_values({A:True,B:True}) = {A:True}

calProba(value,evidence): calculate the probability of the value given evidence, actually is to extract info from random variable's table

sample(event): return a sample of the variable based on the event(go through a random process)

Table(Used for Xml Parser, not a practical class, each table is binded with a randomvariable):

Property:

state(rv): to denote the random variable that this table describes

given(rvs): the state's parentNodes

query: the probability distribution information, in class of dictionary and each query is in format as following:

query = {"state": value, "evidence": evidence, "probability": proba}

eg: in alarm net, one of A's query is as following:

{'state': True, 'probability': 0.95, 'evidence': {'B': True, 'E': True}}

Methods:

processItems(): actually process a definition element in xml file, set the instance's query property

processNormalItems(): process the normal definition tag in alarm file- and wet-grass file

(eg: DEFINITION>
<FOR>M</FOR>
<GIVEN>A</GIVEN>
<TABLE>
<!-- M !A -->
<!-- A --> 0.70 0.30

<!-- !A --> 0.01 0.99
</TABLE>
</DEFINITION>)

processOneLineItem(): process the specific definition tag in dog-out tag in dog-out file

(eg: <DEFINITION>
<FOR>hear-bark</FOR>
<GIVEN>dog-out</GIVEN>
<TABLE>0.7 0.3 0.01 0.99 </TABLE>
</DEFINITION>)

Function:

create query(query,evidence,proba): given a query variable, a evidence and a probability, return the info of this probability distribution in class of dictionary and in a standard format(mention in Table.query property)

probability(p): return a sample given a probability

(eg: for variable A in alarm net, return True or False given a probability, used for sampling algorithms)

XmlParser.py: use xml.dom.minidom as the parser

Methods:

getVariables(root): given the document root object, return all its random variables

getDeifinitions(root):given the document root object, return all its definition tags, (in class Table mentioned above)

parseTableTag,parseTable: some auxiliary functions that help to parse definition tags

parsefile(filename):main function, input a file name and return a bayesnet object the file stands for s

Parser result screen shot: print out each variable's name, parent Node,and probability distribution

```
B
[]
{'state': True, 'evidence': {}, 'probability': 0.001}
{'state': False, 'evidence': {}, 'probability': 0.999}
E
[]
{'state': True, 'evidence': {}, 'probability': 0.002}
{'state': False, 'evidence': {}, 'probability': 0.998}
A
[B, E]
{'state': True, 'probability': 0.95, 'evidence': {'B': True, 'E': True}}
{'state': False, 'probability': 0.05, 'evidence': {'B': True, 'E': True}}
{'state': True, 'probability': 0.94, 'evidence': {'B': True, 'E': False}}
{'state': False, 'probability': 0.06, 'evidence': {'B': True, 'E': False}}
{'state': True, 'probability': 0.29, 'evidence': {'B': False, 'E': True}}
{'state': False, 'probability': 0.71, 'evidence': {'B': False, 'E': True}}
{'state': True, 'probability': 0.001, 'evidence': {'B': False, 'E': False}}
{'state': False, 'probability': 0.999, 'evidence': {'B': False, 'E': False}}
J
[A]
{'state': True, 'probability': 0.9, 'evidence': {'A': True}}
{'state': False, 'probability': 0.1, 'evidence': {'A': True}}
{'state': True, 'probability': 0.05, 'evidence': {'A': False}}
{'state': False, 'probability': 0.95, 'evidence': {'A': False}}
```

Enumeration(Enumeration.py): implement Enumeration Algorithm

Methods:

enumerate(queryvariable,evidence,bayesnet): input a query variable, the evidence and a bayesnet object, return the probability distribution of the query variable given evidence use Enumeration algorithm

enumerateall(variables, evidence): a recursive function that do multiple production between all random variable of the net

extend(evidence,variable,value): extend the evidence offered by add variable=value to the evidence

Details Discussion:

Here comes up a core concept called “evidence fixed”: which means when go through all variables of the net,

if the variable is already in evidence given, directly calculate it’s probability of value in the evidence based on evidence(use rv.calProba(value,evidence))

if the variable is an unobserved variable(not in the evidence), then go through all its value in domain to get the probability distribution and add them up to. AIMA Equation 13.9

The problem of enumeration is about time complexity as it in short will go through whole joint probability distribution. Although it may be not

clear on this project as the net given has a small size, but when come across a large net, this will lead to a slow calculation performance.

Code:

```
def enumerateall(vars, evidence):
    if not vars:
        return 1
    else:
        first, rest = vars[0], vars[1:]
        if first.name in evidence:
            value = evidence[first.name]
            return first.calProba(value, evidence) * enumerateall(rest, evidence)
        else:
            summation = 0
            for value in first.domain:
                summation += first.calProba(value, evidence) * enumerateall(rest,
                                                                extend(evidence, first.name, value))
            return summation
```

Sampling Algorithm:

Basic Function:

Priorsampling(bayesnet): produce a sample randomly by assign values to all variables of the bayesnet given

details:

go through all variables in topological order(from top to bottom), and accmulatively add variable=value to event, sample later variables based on the event produced

For example, in alarm net, firstly we sample variable B and E(these two variables are at the top) , if the event now is {B:True, E:True} then we sample variable A based on the evidence above(look at the file when B and E are both True, then A will probably be True), then do same thing for J & M as these two variables need A in evidence to sample.

The sample order: B-E-A-J-M(B&E, J&M could change order as they are at the same level) is called "topological order".

Rejection Sampling(query,evidence,net,N):

Sample the net N times, for each sample produced, if it is consistent with the evidence given(here, consistent means for each key,value in evidence, it must has same value of key in sample)

If sample produced is consistent with evidence, select the value of query in sample, add it to the counts(here actually is doing the conditional probability calculation).

Finally, normalize the counts and return probability distribution of query variable.

Code:

```
def rejectionsample(query, event, bayesnet, N):
    """
    :param query: the query variable
    :param event: the evidence
    :param bayesnet: the baysnet
    :param N: do N times sampling
    :return: the normalized probability distribution by sampling
    """
    distribute = {status: 0 for status in bayesnet[query].domain}
    for _ in range(N):
        sample = priorsample(bayesnet)
        if consistent(sample, event):
            distribute[sample[query]] += 1
    rv.normalize(distribute)
    return distribute
```

Discuss:

The problem of rejection sample I came across in doing experiment is that when the N(sample times is small, for example, less than 100) and the evidence given dose not match the sample that prior sample function produced well(for example, if the evidence given is B = True in alarm net, it is rarely the sample will be consistent with evidence because evidence given is such a event with low probability).

The solution to this kind of problem is to assign a great value to sample times(default value is 25000 for all sampling algorithms)

Likehood Sampling:

Do not reject any sample, instead assign each sample a weight which stands for the probability that this kind of sample could happen based on evidence given.

Details Discussion: the sample process is somehow as same as prior sampling: Go through all the variables in net in topological order(from top to bottom), there are two cases:

if the variable is in evidence given, then update weight(intitial weight is 1) by multiple weight and variable.calProba(node value in evidence,evidence), this stands for probability that the sample will appear

if variable not in evidence, then just sample it base on the event produced(this is a cumulative process, update the weight or sample variable for each variable in net)

Code representation:

```
def weightedsample(bayesnet, evidence):
    w = 1
    event = dict(evidence)
    for node in bayesnet.nodes:
        if node.name in evidence:
            w = w * node.calProba(evidence[node.name], event)
        else:
            event[node.name] = node.sample(event)
    return event, w
```

for each sample and its weight, update query's value probability distribution by add the weight of sample
Code representation:

```
def likehoodweighting(query, evidence, baysnet, N):
    """
    :param query: query variable, str
    :param baysnet: baysnet, from xmlfile
    :param N: sample times
    :return: the normalized probability distribution in bayesn
    """
    assert query not in evidence
    W = {x: 0 for x in baysnet[query].domain}
    for _ in range(N):
        sample, proba = weightedsample(baysnet, evidence)
        W[sample[query]] += proba
    rv.normalize(W)
    return W
```

Finally, do normalization on W, return it

Discuss:

The problem of likehood weighting sampling algorithm is some how as same as rejection sampling, which also depends on the evidence given. If the evidence itself is with low probability.For example in alarm net, if evidence given is B=True, E = True, which leads to a low weight for each sample(the weight will be less than 0.001*0.002), may result diffculty in normalization process finally(the weight is so small that it is hard to do normalization).

The solution here is as same as Rejection sample, set a great value of default sample times to make it easy to normalize the probability distribution.

Gibbs Sampling:

As for likehood sampling, to sample each variable in net we just care about its parent nodes(in topological order), however, in Gibbs sampling, to sample each node we also take its childnodes into consideration.

Gibbs sampling in some degree is just to sample one variable each time

Gibbs sampling process is as following:

Firstly, initialize a state(fixed evidence and initialize all other variables randomly)

To produce each sample, what we only care about are just those unobservable variables. For each non evidence variable, go through its domain, separately update variable=value to the state(update the initial state produces firstly), then calculate the probability by multiplication current query variable's probability and all it's childNodes probability given the state(evidence).

Then add this probability to the probability distribution, sample current query variable based on the normalization of whole probability.

For example, if the query is A , the evidence is B=True for net 1, the process will go as following:

Initialize the state for example as {B:True,E:True,A:False,J:True,M:True}

Then for those non evidence variables[E,A,J,M](in topological order),

firstly sample E: as E is at the top of net, update E = True, E = False separately to the state, add the probability of each case to a blank dictionary then normalize it.

Then sample E by normalization of probability calculated, update E=sampled value to the initialized state then add query variable's value in state, update each non evidence variable based on updated state(for example, the next variable to be sampled is A and we sample A based on the state updated by E).

update each non evidence variable one time for each sample, get the query's value in state and then normalize these appearance, get the probability distribution of query variable.

Code Representation:

```
def gibbssampling(query, evidence, bayesnet, N):
    assert query not in evidence
    counts = {x: 0 for x in bayesnet[query].domain}
    Z = [r.name for r in bayesnet.nodes if r.name not in evidence]
    state = dict(evidence)
    for z in Z:
        state[z] = choice(bayesnet[z].domain)
    for _ in range(N):
        for z in Z:
            state[z] = markovsample(z, state, bayesnet)
            counts[state[query]] += 1
    rv.normalize(counts)
    return query, counts


def markovsample(query, evidence, bayesnet):
    centralnode = bayesnet[query]
    Q = {}
    for value in bayesnet[query].domain:
        currevent = extend(evidence, query, value)
        markovchainProb = [node.calProba(currevent[node.name], currevent) for node in centralnode.childNodes]
        Q[value] = centralnode.calProba(value, evidence) * calProduct(markovchainProb)
    rv.normalize(Q)
    return probability(Q[True])
```

Bonus Question: **Variable Elimination**

The idea of variable elimination is to simplify process of calculation based on Enumeration algorithm, which is to store the result of probability distribution, in other words, this algorithm will go in reversed topological order(from bottom to top).

Here we need create a new class called *Factor(Page 533 AIMA),* which stores the probability distribution of a variable.

Calculation process is as following:
Firstly, go through all variables in reversed topological order, for each node, create a factor instance for it by listing all probability distribution info of a variable(for example, in alarm net, query variable is J, and A = True is in evidence, then Factor(A) should be (0.9,0.1) else if A not in Evidence, the factors should be(0.9,0.1,0.05,0.95))

Here some detail works need to be done. For those variables that are not the query variable and not in evidence, we need to sum out the variable by do production between factors and then do marginalization on non evidence variable, here is what we called " variable elimination ", sum out those non observed variables to get the new factors.

At the end, do production on all factors left and normalize it to get probability distribution of query variable.

Some support Functions:
allevents(variables,bayesnet,event): produce all event by extending variables to event given

eventvalues(variables,event): extract value of variables in evidence, return a tuple storing these information

make factor(query,bayesnet,evidence): make a factor based on the query and given evidence(to clarify if query's parent nodes are in evidence given)

ishidden(query,evidence, variables): clarify if a variable is not the query variable and not in evidence given

sumout Function & sum method in Factor class: do sum out work on some queries for a specific variable

pointerwise function & method in Factor class: do production between two factors as described in textbook

To Run the Program, Please do as following:

Run each Algorithm file separately(I write a process input function in RunProgram.py to preprocess the input from keyboard to make it usable for sampling and enumeration algorithm)

For example, if you want to run Enumeration algorithm given that the query variable is A, file is aima-alarm, evidence is B is True and M is False, please input following command line in shell:
python Enumeration.py aima-alarm.xml A B T M F
Same mechanism for other algorithms

Algorithms Files Dictionary:

- EnumerationInference.py -> Enumeration algorithm
- RejectionSampling.py -> Rejection Sample
- LikehoodWeighting.py -> LikeHood Sample
- GibbsSampling.py -> Gibss Sample
- VariableElimination.py -> VariableElimination

The command line should in format of:

python [Algorithm file name] [net file name] [queryname] [evidence][value] [evidence][value]

The value here could only be T(stands for True) or F(stands for False)

Some Tips:

The xmlParser is only designed for example files, do not support other format files(such as the random variable has more than 2 values in domain), the value of the random could only be True or False.

ProjectLink: https://github.com/RunhuaGao/AIProjects/tree/master/Project%203