CSC442 Intro to AI Project 3: Uncertain Inference

Contributors: Jeet Thaker (jthaker), Richard Magnotti (rmagnott)

CONTRIBUTIONS

Jeet Ketan Thaker:

-Likelihood sampling
-Rejection sampling
-XML parser (most)
-Writeup

Rich Magnotti:

-Exact inference
-Driver/input
-XML parser (some)
-Writeup

INTRODUCTION

In this project our goal was to program a piece of software to calculate probabilities under uncertain conditions. We did this by coding in Python Bayesian Network techniques outlined in the AIMA book. Specifically, we were interested in Bayesian exact inference, and approximate inference utilizing rejection sampling and likelihood sampling.

CAVEATS:

There is one vital consideration we needed to take into account when parsing the xml files. We found the probabilities from the CSC 442 course were typically in reverse binary order (e.g. for 3 variable truth values the order would be: TTT, TTF, TFT, TFF, FTT, FTF, FFT, FFF). Therefore, we built our parser to follow that structure. The probabilities for *aima-alarm* and *aima-wet-grass* xml files were listed in table format, but we changed them to the typical streamlined order. Similarly, *dog-problem* listed probabilities in some different order in which each pair of consecutive probabilities summed to 1 (e.g. [TT, FT], [TF, FF]), with each pair corresponding to a probability of some query given some evidence, and the negation of either the evidence or the query. So, we manually reordered the probability lists into reverse binary.

METHODS:

I XML Parser

The bayesian network has been provided in an XML format. We need a data structure to store all relevant information of a bayesian network. We use a graph in python consisting of nodes of class *bayesian_network.* Each nose has the following attributes:

Name (string)
Value (integer)
Children (list)
Parent (list)
Probability_Distribution (list)

From the XML file we need to get information about the names of variables used, their relationship to each other and their probability distributions.

For parsing XML files, we make use of the library *xml.etree.ElementTree*. Let this module be called ET.

1. We pass the XML file as an argument to ET.parse and store the node in a variable called *tree*
2. For all the tag names *variable* we extract the *name* information from them and append these names to a list *l1*
3. Next we create nodes of class *bayesian_network* with the names of the variables that we got from the XML file and store these nodes in a list *l2* for later use.
4. For storing the probability information we need to change the given XML files a bit. We change the <TABLE> information and make sure that all the probabilities are in the same line.
5. We search for the <DEFINITION> tag and in it we look at the <FOR> information and pull up the corresponding node from our *l2* list. We also pull up nodes corresponding to <GIVEN> information and assign these nodes as parents to our original node. We also assign our original node as the child of each of these parent nodes.
6. From the <TABLE> information inside definition we pass the information to a function called space_parser, which takes in this string and parses it based on whitespaces. It converts strings to floats and appends to a list *l3.* We then make the original nodes Probability_Distribution as this list *l3.*
7. Steps 5 and 6 are performed for all the definition tags.
8. Finally we create a ‘root’ node and assign its children a list of nodes which had no parents from our *l2* list. We also set the parent of each of these nodes as the ‘root’ node. Now this ‘root’ node contains all the information of our bayesian network.
9. Additionally, we included a function here ‘Linearize()’. This function sorts the nodes in order of dependence/topologically in order to avoid recalculating parent probabilities repeatedly.

Ⅱ Exact Inference Enumerator and Driver

The Python module ‘*BN_Driver.py*’ is the main function which handles the auxiliary function calls to attain the desired probabilities. This module accepts user input from the command line in the following order according to the order prescribed in the assignment rubric:

‘*python3 fileName.py bayesianXML.xml query_variable evidence_variables*’

The module then passes the command line arguments to ‘*xml_to_bn.py*’ where the desired xml file is parsed, and the Bayesian network is created then returned as a list of nodes to the driver module.

Once the network is returned, the driver calls the exact inference module and passes the other command line user input arguments to the module ‘*Bayesian_Enumerator.py*’ with the following data types:

- query X → list: [variable_name, (tuple of values X can take)],
- evidence → dictionary containing all the evidence variables and their truth values
- ordered network → list containing all the nodes in the Bayesian network in topological order

This algorithm is a fairly straightforward recreation of AIMA Fig. 14.9, with some meaningful additions. The sequence breakdown is as follows in a stepwise fashion (note: all the following steps are for the case where the variable Y (*var[0]* in the code) is not in the evidence. However, the steps for the case where Y is in the evidence are so similar it did not warrant a separate explanation):

1. Because the Bayesian network is initiated with each node having *value* 0, we update the network values according to the evidence passed in the current call.
2. Find the parent nodes of the variable Y and their truth values.
3. Convert the truth value combination of Y and its parents from binary decimal.
4. Use that decimal value to find the element of the probability list that belongs to Y.
5. Multiply the calculated probability by a recursive call to the same function with the updated Bayesian network and remaining variable name list.

Once this recursive function call stack is executed completely, steps 1-5 will be repeated for the negated query. Ultimately we will be left with 2 float values representing the final probabilities for P(query|evidence) and P(~query|evidence), which we then use to find the normalization constant ⍺ (alpha). Then we normalize the two probabilities and return the distribution to the driver module and printed.

Ⅲ Approximate Inference and Driver

The file 'BN_Driver_Approximate.py' is the main module that drives approximate inference. To run this module, it expects command line arguments as given in the assignment :

```
python (BN_Driver_Approximate.py) (number_of_samples) (filename.xml) (query)
(evidence)
```

We have implemented two sampling methods for this project. One is rejection sampling and the other is Likelihood Weighting Sampling.

<u>Pseudocode for Rejection Sampling</u>

1) We initialize count = 0 , event_count = 0 and number = 0.
2) While number < number_of_runs , which is provided by the user, we do the following :
    a) We traverse through the bayesian network once in topological order, flipping nodes as we go along. The flip is done in accordance to the probability distribution of the node which is stored as node.probs.
    b) We take into consideration the value of parents before flipping a node to ensure that the samples drawn are consistent.
    c) Once this traversal is over, we add 1 to number and check if the evidence variables match the values given by the user , if so then we add 1 to count.
    d) If the above is TRUE then we look at the query variable , if its value is 1 then we add 1 to event_count.
3) We return event_count / count once we get out of the while loop.

<u>Pseudocode for Likelihood Sampling</u>

1) It is exactly the same as rejection sampling with a few differences.

2) We traverse the network just as before flipping nodes as we go along. Before each traversal begins we set a variable weight to be 1. When we encounter an evidence variable we set the value of the variable to be equal to the value provided by the user.
3) We draw the probability given its parents that the value of the variable would be equal to the value provided and set the weight as the multiplication of the previous weight value and the probability we found out.
4) We set count = count + weight. And IF the query variable is true THEN we set the event_count = event_count + weight
5) We return event_count / count once we get out of the while loop.

EXPERIMENTATION

We ran some evaluations in order to both ensure the efficacy of our respective inference methods, as well as verify their accuracy by comparing the values we attained. We ran the following 10 tests:

Likelihood Sampling:

aima-alarm1.xml	Rich	Jeet	**Likelihood** Sampling Sample Size	% Difference
1) P(M | j, ~a)	0.01	0.010	100,000	0.672
		0.00996	500,000	0.4355
		0.0100	1,000,000	0.0729
2) P(A | ~m, ~e)	0.00588	0.00069	100,000	17.10
		0.000587	500,000	0.258
		0.000591	1,000,000	0.465
3) P(E | ~j, e)	0.00189	0.00156	100,000	5.374
		0.00140	500,000	5.375
		0.00149	1,000,000	0.418
dog-problem.xml				
4) P(Dog-Out | ~family-out, light-on)	0.3067	0.307	100,000	0.0033

		0.3063 0.0.3062	500,000 1,000,000	0.119 0.1722
5) P(Family-Out | dog-out, ~bowel-problem, hear-bark)	0.34615	0.3436 0.3470 0.3470	100,000 500,000 1,000,000	0.7401 0.257 0.246
6) P(Bowel-Problem | light-on)	0.01	0.099 0.0104 0.0101	100,000 500,000 1,000,000	0.612 4.03 1.66
7) P(Hear-Bark | bowel-problem, ~family-out)	0.6793	0.683 0.6796 0.6793	100,000 500,000 1,000,000	0.55 0.055 0.013
aima-wet-grass.xml				
8) P(S | r, w)	0.1944	0.1934 0.1947 0.1944	100,000 500,000 1,000,000	0.546 0.132 0.032
9) P(C | ~w)	0.361	0.359 0.3622 0.3607	100,000 500,000 1,000,000	0.5388 0.357 0.078
10) P(R | ~w, ~c)	0.02439	0.02407 0.0243 0.02451	100,000 500,000 1,000,000	1.31 0.151 0.528

Rejection Sampling:

aima-alarm1.xml	Rich	Jeet	Rejection Sampling Sample Size	% Difference
1) P(M | j, ~a)	0.01	0.012	100,000	21.33
		0.0101	500,000	0.649
		0.01002	1,000,000	0.206
2) P(A | ~m, ~e)	0.00588	0.000598	100,000	1.644
		0.00062	500,000	5.399
		0.000574	1,000,000	2.526
3) P(E | ~j, e)	0.00189	0.00136	100,000	8.299
		0.00151	500,000	1.628
		0.00145	1,000,000	2.134
dog-problem.xml				
4) P(Dog-Out | ~family-out, light-on)	0.3067	0.303	100,000	1.313
		0.306	500,000	0.343
		0.3098	1,000,000	1.0255
5) P(Family-Out | dog-out, ~bowel-problem, hear-bark)	0.3461	0.3462	100,000	0.0215
		0.3445	500,000	0.469
		0.3456	1,000,000	0.1538
6) P(Bowel-Problem | light-on)	0.01	0.01	100,000	0.11
		0.0101	500,000	1.583
		0.0098	1,000,000	1.97
7) P(Hear-Bark | bowel-problem, ~family-out)	0.6793	0.6807	100,000	0.211
		0.678	500,000	0.0921

		0.6811	1,000,000	0.2684
aima-wet-grass.xml				
8) P(S | r, w)	0.1944	0.1933	100,000	0.602
		0.1933	500,000	0.593
		0.1943	1,000,000	0.059
9) P(C | ~w)	0.361	0.365	100,000	1.28
		0.3609	500,000	0.013
		0.3603	1,000,000	0.176
10) P(R | ~w, ~c)	0.02439	0.02388	100,000	2.08
		0.0234	500,000	3.93
		0.0240	1,000,000	1.47

CONCLUSION

Exact inference does not speed up the probability finding process , bayesian networks are just a way to represent all probability information in a compact form. The reason we can compress probability information is due to the causal nature of the universe. We can make assumptions about conditional probabilities which help us in finding the whole probability distribution from just a handful of facts.

Since exact inference does not provide a speed up , we use sampling in order to approximate the answer. The hope is that as we increase the number of samples the approximation keeps on getting better and better and in the limit sampling methods would give you the same answer as would exact inference.

From out preliminary search , we find that across all problems the average number of samples needed by ‘Likelihood Sampling’ in order to get an percent difference between actual probability and the approximate value below 1% is 270,000.