CS 540 (Shavlik) HW 3 – Probabilistic Reasoning

Assigned: 10/20/16
Due: 11/8/16 at 11:55pm (not accepted after 11:55pm on 11/15/16)
Points: 150

Suggestion: you should consider doing this HW in a word processor since cut-and-pasting is likely to be useful.

You should do Problem 1 before the midterm since it involves material from Lecture 13, the last lecture whose topics might be on the midterm.

Your solutions to Problems 1-4 should be placed in HW3_p1_to_p4.pdf.

Problem 1: Full Joint Probability Distributions (15 points)

Consider this *full joint probability distribution* involving four Boolean-valued random variables (A-D). For this problem you can simply walk through the cells in the table below one-by-one and add up those probabilities where the expression for a given question below is true. I.e., you do not need to create "complete world states."

A	B	C	D	Prob
F	F	F	F	0.002
F	F	F	T	0.018
F	F	T	F	0.013
F	F	T	T	0.127
F	T	F	F	0.019
F	T	F	T	0.021
F	T	T	F	0.038
F	T	T	T	0.122
T	F	F	F	0.031
T	F	F	T	0.019
T	F	T	F	0.150
T	F	T	T	?
T	T	F	F	0.080
T	T	F	T	0.070
T	T	T	F	0.060
T	T	T	T	0.090

i. Compute *P(A = true and B = false and C = true and D = true).*

ii. Compute *P(B = false and C = false and D = true).*

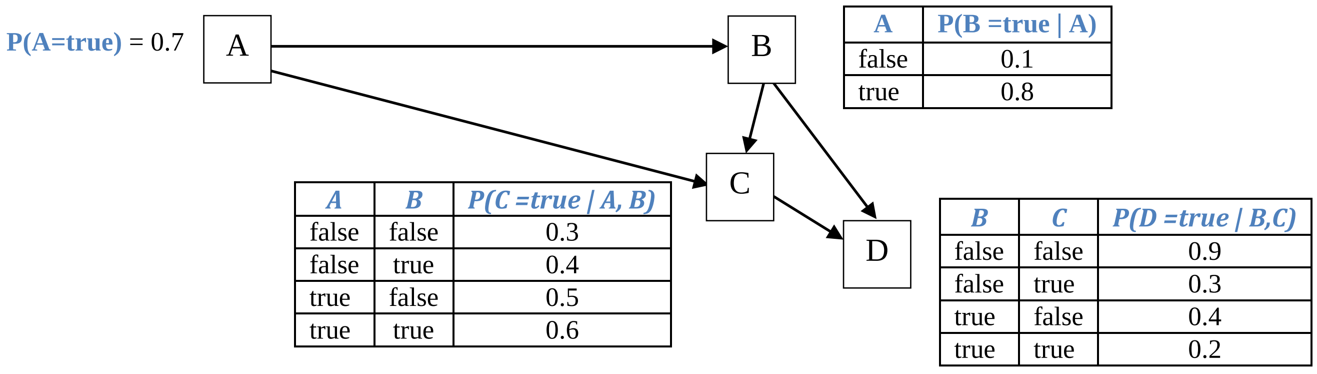
iii. Compute *P(A = true or B = false).*

iv. Compute *P(A = true | B = false and C = false and D = true).*

v. Compute *P(B = false and C = false and D = true | A = true).*

Problem 2: Bayesian Networks (20 points)

Consider the following Bayesian Network, where variables **A-D** are all Boolean-valued:



P(A=true) = 0.7

A	P(B =true | A)
false	0.1
true	0.8

A	*B*	*P(C =true | A, B)*
false	false	0.3
false	true	0.4
true	false	0.5
true	true	0.6

B	*C*	*P(D =true | B,C)*
false	false	0.9
false	true	0.3
true	false	0.4
true	true	0.2

Show your work for the following calculations.

i. Compute *P(A = true and B = true and C = true and D = true).*

ii. Compute *P(B = false and C = false and D = false).*

iii. Compute *P(A = true* | *B=true and C=true and D=false).*

iv. Compute *P(D= false* | *A=true and B = true and C = true).*
// Do this one without employing the *Markov Blanket* property, i.e., algebraically or
// numerically confirm that knowing *A=true* does not change the results from computing
// *P(D= false* | *B = true and C = true).*

v. Compute *P((A = true and D = true) or (B = true and C = true*)).

Problem 3: BNs and Full Joint Probability Tables (5 points)

Use the Bayes Net from Problem 2 to fill in the bottom quarter of the full joint probability table below (feel free to fill it all in, but to reduce the tedium we are only requiring you to fill in the bottom quarter). Show your work.

A	B	C	D	Prob
F	F	F	F	?
F	F	F	T	?
F	F	T	F	?
F	F	T	T	?
F	T	F	F	?
F	T	F	T	?
F	T	T	F	?
F	T	T	T	?
T	F	F	F	?
T	F	F	T	?
T	F	T	F	?
T	F	T	T	?
T	T	F	F	?
T	T	F	T	?
T	T	T	F	?
T	T	T	T	?

Problem 4: Bayes' Rule (20 points)

Define the following two variables about people:

shot	=	got flu shot last year
flu	=	caught flu this year

Assume we know from past experience that:

Prob(shot)	=	*0.7*
Prob(flu)	=	*0.2*
Prob(flu | shot)	=	*0.1*

i. Given someone did not get a *shot*, what is the probability he or she does gets the *flu*?

ii. Given you find out someone did not get the *flu*, what's the probability he or she got a *shot*?

Be sure to show and explain your calculations. Start by writing out the above questions as conditional probabilities.

In the general population, 57 in a 1,000,000 people have the dreaded *DislikeExams* disease. Fortunately, there is a test (*test4it)* for this disease that is 95% accurate. That is, if one has the disease, 95 times out of 100 *test4it* will turn out positive; if one does *not* have the disease, 5 times out of 100 the test will turn out positive.

iii. You take *test4it* and the results come back true. Use Bayesian reasoning to calculate the probability that you actually have *DislikeExams*. That is, compute:

Prob(DislikeExams = true | test4it = true)

Show your work; you may use *DE* for *DislikeExams* and *T4* for *test4it* if you wish.

Problem 5: Creating Probabilistic Reasoners that Play Nannon (90 points)

This problem involves writing Java code that implements three probabilistic reasoners to play the two-person board game called Nannon (http://nannon.com), which is a simplified version of the well-known game Backgammon (http://en.wikipedia.org/wiki/Backgammon). Instructions for Nannon are available at http://nannon.com/rules.html.

Here is how we will formulate the task. At each turn, whenever there is more than one legal move, your *chooseMove* method is given access to:

1) The *current board configuration.*
2) A *list of legal moves*; each move's effect is also provided, and you are able to determine the next board configuration from each move. (Explanations of which effects are computed for you appear in the *chooseMove* method of the provided RandomNannonPlayer and in the ManageMoveEffects.java file).

Your *chooseMove* method needs to return one of the legal moves. It should do this by using Bayes' Rule to estimate the odds each move will lead to a winning game, returning the one with the highest odds. That is, it should compute for each possible move:

$$\frac{\textit{Prob(will } \underline{\textit{win}} \textit{ game } | \textit{ current board, move, next board, and move's effect)}}{\textit{Prob(will } \underline{\textit{lose}} \textit{ game } | \textit{ current board, move, next board, and move's effect)}}$$

Your solution need not use ALL of these given's to estimate these probabilities, and you can choose to define whichever random variables you wish from the provided information. The specific design is up to you and we expect each student's solution to be unique.

You need to create three solutions. In one, you will create a full joint probability table. In the other two you will create two (Bayesian Networks, BNs); neither can be a BN equivalent to your full joint probability table. One BN should be Naive Bayes (NB) and the other needs to somehow go beyond the NB conditional-independence assumption (see notes). It is up to you to decide the specific random variables used and, for the non-naive Bayesian Network, which conditional independence assumptions you wish to make. The random variables in your three solutions need not be the same.

You need to place your three solutions in these files, where *YourMoodleLoginName* is your actual Moodle (i.e., your UWisc) login:

FullJointProbTablePlayer_YourMoodleLoginName.java

NaiveBayesNetPlayer_YourMoodleLoginName.java

BayesNetPlayer_YourMoodleLoginName.java

Copy all the Java files in http://pages.cs.wisc.edu/~shavlik/cs540/HWs/HW3/ to your working space. The provided PlayNannon.java, NannonPlayer.java, and RandomNannonPlayer.java files contain substantial details on what you need to do. You should start by reading the comments in them; I suggest you read the files in the order they appear in the previous sentence.

After the final HW due date, you can share you code with others and have your players play one another, since they have unique names. We might also run student code against one another during grading.

So how do you get the necessary information to compute these probabilities? After each game, your players' *updateStatistics* method is given information about the sequence of board configurations encountered and the moves chosen by your player in that game, as well as whether or not your player won that game. You should not try to figure out which moves where good or bad in any one specific games; instead, if a game was won, all moves in it should be considered good (i.e., led to a win) and if a game is lost all moves should be considered bad (led to a loss). Obviously some moves in losing games were good and vice versa, but because we are using statistics, we are robust to this 'noise.'

Your three players need to implement the *reportLearnedModel* method, which reports the value of the random variable (or values of the combination of random variables) where the following ratio is *largest* (i.e., most indicative of a win) and the *smallest* (i.e., most indicative of a loss):

prob(randomVariable(s) | win) / prob(randomVariable(s) | loss)

For your full-joint-prob table, randomVariables should be a setting of all the random variables other than the 'win' variable (i.e., loss = ¬ win). For your NB player, randomVariable(s) should be *one* variable other than *win*. For your Bayes Net approach, randomVariable(s) should be one of the non-NB entries in the product of probabilities you compute. (Recall that if we want to allow some dependencies among our random variables, the product in a non-naive Bayes Net calculation will include something like $p(A \mid B \wedge win) \times p(B \mid win)$, which is equivalent to $p(A \wedge B \mid win)$, as explained in class.)

The *reportLearnedModel* method is automatically called (by code we have written) after a run of *k* games completes when Nannon.*reportLearnedModels* is set to true.

It is fine to print more about what was learned, but the above requested information should be easy to find in your printout.

What needs to be turned in, in addition to the three Java files listed above (be sure to comment your code)? A report (in HW3_p5.pdf) containing:

1. A description of your design for your FullJointProbTablePlayer; should be 1-2 pages. Include an illustrative picture (the picture can be hand drawn). Don't draw the complete full-joint table, but provide a visualization of its contents.
2. A description of your design for your NaiveBayesNetPlayer; should be 1-2 pages. An illustrative picture of your Bayes Net is required (the picture can be hand drawn).
3. A description of your design for your BayesNetPlayer; should be 1-2 pages. An illustrative picture of your Bayes Net is required (the picture can be hand drawn). Focus your discussion on the differences from your NB player.
4. A 1-2 page report presenting and discussing how your three players performed against each other, as well as against the provided 'hand coded' and 'random moves' players.

Create a table where the columns are (change "Your" to "My" in your report):

YourFullJoint YourNB YourBN HandCoded Random

And the rows are

YourFullJoint

YourNB

YourBN

Feel the 'upper triangle' portion of this table with the results (the percentage of the time the column's label beat the row's label) on the 6 cells x 3 pieces per player board using 1M games after 1K 'burn-in'games.

Please type your report.

Here some statistics from my solution on the 6 cells x 3 pieces per player board. Note that results can vary depending on the 'seed' for the random-number generator. Also note that several students' solutions beat my solutions in previous cs540 classes.

33% random-moves	vs	67% my Full-Joint Player
47% hand-coded	vs	53% my Full-Joint Player
33% random-moves	vs	67% my Bayes Net
46% hand-coded	vs	54% my Bayes Net
39% random-moves	vs	61% my Naive Bayes
52% hand-coded	vs	48% my Naive Bayes

(so don't worry if your Bayes Net cannot beat the hand-coded player)

Note that we will test your NaiveBayesNetPlayer and your BayesNetPlayer under various conditions. The PlayNannon.java file provides details. Also note that you should not modify any of the provided Java files since when we test your solution we will be using the original versions of these files.

Feel free to create additional players based on genetic algorithms and/or a nearest-neighbor approach (or even decision trees/forests), but that is not required nor will any extra credit be given. I will be quite happy to talk to students about such approaches, though.

A GUI-Based Player

There is a GUI-based way to play against a trained computer opponent. If you haven't already, download the GUI_player.java, MiscForGUI.java, NannonGUI.java, and PlayingField.java. (I crudely hacked up some old GUI code I had written around 1998 called the AgentWorld, so don't look at the code - it has a lot of unused junk in it.)

To invoke this player (as X, use arg2 to be O) do this in PlayNannon:

```
  String   arg1 = "gui";
```

It will first silently play the specified number of burn-in games against your chosen computer-based player; the GUI-based player will make random moves. It will then pop up a Java window and allow you to use the mouse to play against the computerized player.

Legal moves are visually highlighted. If you press down on the mouse but don't release, you will see where a moveable piece will go. If you release quickly enough (ie, "click on") a moveable player, the move will happen. If you press down on a moveable piece but do NOT want to make that move, move the mouse off that player and then release it (or hold it down long enough so the press+release is not viewed as click).

Note you only get to select a move for those board states where you have a CHOICE of moves, so games might seem a bit haphazard.

I have only tested this in Windows, but it is simple Java and should run anywhere Java does.

In a related note, at the end of a match you can manually watch two computerized players play. The file PlayNannon.java's main has a line like this:

```
      Nannon.useGUItoWatch  = true;
```

Change true to false to have the GUI turned off.

One known bug: you can't kill the GUI using the upper-right X in Windows, but there is a QUIT button (or kill the PlayNannon java process).

Some Additional Tips

Here are some miscellaneous notes and tips about Nannon that came up in student discussions when I used Nannon in previous semesters.

1) In Java when you spec arrays, you need not use constants but can instead do something like this:

```
          myRandomVar_win
          = new int[NannonGameBoard.piecesPerPlayer][NannonGameBoard.cellsOnBoard];
        // I am simply illustrating variable-sized arrays here and don't use
        // something exactly like this in my code.
        // (The actual code uses 'accessor' methods for reading and writing these static vars.)
```

You should use NannonGameBoard.piecesPerPlayer and NannonGameBoard.cellsOnBoard in your BayesNet players so that it can handle any game size, but your FullJoint player only needs to handle NannonGameBoard.piecesPerPlayer = 3 and NannonGameBoard.cellsOnBoard = 6 (in Java the SIZE of an individual dimension can vary, but the NUMBER of dimensions cannot - at least as far as I know; possible some advanced feature of Java allows this - and this limitation makes it hard for the FullJoint player to handle various game sizes, plus for larger games, the

FullJoint table might cause you to run out of memory since the table grows exponentially with game size).

2) You can vary Nannon.numberOfGamesInBurnInPhase to run experiments via an 'accessor' function. I made this and other variables private and provided accessor functions, so that during grading we can alter the accessor functions to only allow the TA or me to change this variable. During grading, your player has to 'live with' the settings of this variable (it is ok to make random moves after the burn-in phase, but random moves are likely to increase the odds your player will lose).

Ditto for Nannon.gamesToPlay and Nannon.printProgress (be sure your turned-in player 'runs silently' unless the TA or I set Nannon.printProgress = true because excess printing will slow down things and clutter our output).

Ditto for NannonGameBoard.piecesPerPlayer and NannonGameBoard.cellsOnBoard. It would be rather chaotic if players could change the game in the middle of playing!

Good programming practice is to ALWAYS use accessor functions (ie, setters and getters) rather than making public variables, for reasons like these.

3) If a full joint prob table has *K* cells and we initially place a '1' in each cell to avoid having prob = 0, then technically the *numberWins* and *numberLosses* should initially be set to something like *K*/10, assuming the average game involves 5 world states and half are wins and half are losses, because world states do not overlap. But in the full joint table approach, *K* will be quite large and the real wins and losses will likely be small compared to *K*/10 (and, hence "washed out"). So I suggest you simply initially set *numberWins* and *numberLosses* to 1 (and hence the initial numberGames = 2 if you keep that as an explicit counter).

In the Bayes Net approaches, the various local probability tables can overlap, so one could initially set *numberWins* and *numberLosses* based on the LARGEST such table; this is still a heuristic rather than necessarily correct, but a reasonable approach. Initially setting *numberWins* = 1 and *numberLosses* = 1 is also fine here, given we are playing millions of games.

4) Remember that we COUNT things in our cells, but our formula uses PROB's. So don't forgot that *prob(win) = numberWins / (numberWins + numberLosses)* and if you have some conditional prob that, say, depends on three random variables ('features'), then you need to do *prob(F1=a and F2=b and F3=c | win) = countWins(F1=a and F2=b and F3=c) / numberWins* and *prob(F1=a and F2=b and F3=c | not win) = countLosses(F1=a and F2=b and F3=c) / numberLosses.* And remember to coerce one of the integers to a double so that you do not perform integer division, which would lead to all probabilities being zero!

5) I recommend you first create and debug your NB player, then cut-paste-and-extend it to create your non-NB player. Whether you do the FullJoint player first or last is up to you.

<u>Be sure to monitor the HW3 discussion in Piazza for suggestions, extra information, and (hopefully rarely, if at all) bug fixes.</u>