COMP9418 Report

Task 1: Efficient d-separation test

The basic definition of d-separation involves looking for paths from X to Y and make sure one exists if it is not blocked by an inactive triple. The time complexity of this algorithm would be looking at every node and running a path finding algorithm to the destination. This would result in O(N(N+E)) for each individual X to an individual Y.

The more efficient algorithm, on the other hand, only requires one path finding algorithm. Most of the work relies on removing the nodes and edges. This is insignificant to the time complexity since it would take at most O(E), given that edges would always be more than nodes. Thus, the overall complexity of the algorithm is O(N+E)

I implemented a class Node for the implementation of individual nodes. It has a list of incoming nodes and outgoing nodes. The graph is implemented as a dictionary of nodes. I implemented functions for adding and removing edges as well as removing nodes.

The main challenge for this task is making the class. I think it serves as the backbone for the rest of the assignment given that the Node class can be used for other tasks. The algorithm itself is not as complex in my opinion. As mentioned above, the Node class has functions that could be useful in the next tasks.

Task 2: Estimate Bayesian Network parameters from data

I used similar methods to the tutorial. I merely collect all the data from the csv and counts the number of outcome occurrences for each variable and divide it by the total number of entries to get the percentage. This is stored in a dictionary with the variable as the key and another dictionary with outcome as the key and the percentage as value.

The joint distribution for this graph is
P(Age)P(Location)P(BreastDensity)P(BC|Age,Location)P(Metastasis|BC)P(MC|BC)
P(AD|BC)P(Mass|BC,BreastDensity)P(Lymphnodes|Metastasis)P(FibrTissueDev|AD)P(SkinRetract|BC,FibrTissueDev)P(NippleDischarge|BC,FibrTissueDev)P(Spiculation|FibrTissueDev)P(Size|Mass)P(Shape|Mass)P(Margin|Mass,Spiculation)

Task 3: Sampling

I modified the previous task's probability table as well as Node class. Each probability table is now stored inside individual nodes. For leaf nodes, it remains mostly unchanged with each outcome having a probability. For non-leaf nodes, the probability table is a dictionary with string key consists of a comma-separated value for the dependency's value. The value is a dictionary of outcomes and probabilities. The probability table is adjusted to take in these values for the forward sampling. I also do topological sort to go traverse through nodes for the forward sampling. This is important because variables need to go in order depending on the incoming nodes.

The way I implemented the sampling involves continuously decreasing the random generated number by the probability of the outcomes. If it goes below 0, the generated outcome would be the outcome that reduces the random number to below 0. It is similar to creating a range of values that would result in an outcome from one another.

I found a lot of difficulty deciding how to implement the probability of each distribution. I thought at first that dependencies wouldn't matter as much but the sample data is not making much sense since each variable is independent from one another. The approach that makes most sense to me is to have probability table for each variable and dependencies. I intended to use tuples as keys, but it seems that it is not doable in Python without more work. As a simple workaround, I instead converted the list of dependency outcomes into a string.

Since each entry would have all their variables checked, building a probability table would take the time to loop through both. Thus, the time complexity of the operation would be O(entry*variable number). In addition to this, we must calculate the time it took to generate samples. An individual sample would traverse through the graph in topological order. So, it would only take O(nodes). Given n numbers of sample, the time complexity is O(n*nodes). The total time complexity of generating the model and samples is, therefore, O(n*nodes) + O(entry*variable number).

The accuracy of the sample would grow with more variables given. This is because some variables are dependent on others. If we give less variable, we must assume that some variables are independent of others. Since this is not the case, it would reduce the accuracy of the generated samples.

The accuracy of the sample is equal to the accuracy when testing the training set. This means that the sample data can mimic outliers well enough and the data is similar to the training data.

Task 4: Classification

The prediction is like generating the sample data. Instead of generating data, I used the existing data and calculate the joint distribution for the 3 possible BC outcomes. The highest probability would be the predicted outcome. I used Laplace smoothing because despite it not being possible given the way sample data is generated, in a real-world situation, it would be necessary in case an outlier happens. Furthermore, it wouldn't affect the joint distribution as much anyway.

The classification error is calculated by counting the number of correctly classified data divided by the total data. The predictions are categorical data, so I did not find it necessary to overcomplicate things.

I compared the Bayesian Network Classifier with the KNN classifier. I thought given the dependencies, data with the same outcomes would be similar. Going by this logic, KNN seems like the best choice.

Bayes Network	91.3%
KNN (5 neighbors)	91.175%
KNN (10 neighbors)	90%
KNN (100 neighbors)	85.995%

I decided to start with 5 neighbors because any smaller would result in overfitting. The larger the neighbors, the more the outliers are falsely classified, it seems. Thus, it proves that looking for similarity is not better than counting the probability through dependencies.