Vikgu708

Lab 1 TDDE15

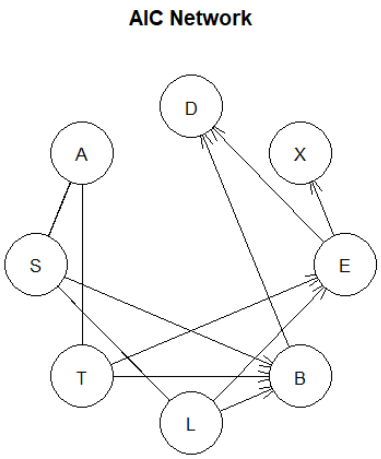
Question 1

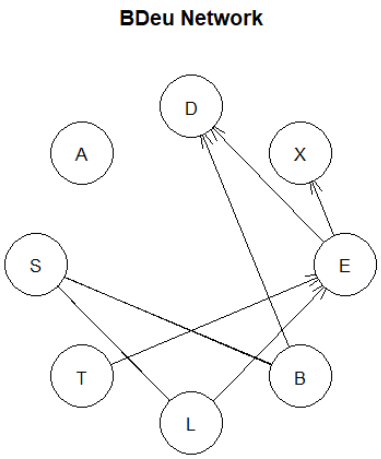
To answer the question I chose to study to different scores for the iterations in the hill climbing method. I then compare the two resulting equivalence networks graphically and with all.equal.

Code

```
library("bnlearn")

data("asia")

set.seed(12345)

network1 = cpdag(hc(asia, score ="aic"))
network2 = cpdag(hc(asia, score ="bde"))

plot(network1, main="AIC Network")
plot(network2, main="BDeu Network")


print(all.equal(network1,network2))
```

Output






```
[1] "Different number of directed/undirected arcs"
```

Reasoning

The two resulting networks are clearly different, where the AIC network has noticeably more dependencies between variables compared to the network that used BDeu scoring for the hill climbing algorithm. The reasoning for this is that the different scorings can result in different local optima as the optimization problem is not convex, and different scores could result in different decisions from the same initial starting point.

Question 2

I first used the PC algorithm for finding the graph structure. Then the parameters were learned using bn.fit. Inference was carried out using querygrain R-package.

Code

```
## Question 2
if (!require(gRain)) {
  source("https://bioconductor.org/biocLite.R")
  biocLite("RBGL")
}

library(gRain)


getFittedBN = function(trainData){
  graphStructure = pc.stable(trainData)
  fittedBN = bn.fit(cextend(graphStructure), trainData)
  return(fittedBN)
}

getTrueBN = function(trainData){
  graphStructure = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
  fittedBN = bn.fit(cextend(graphStructure), trainData)
  return(fittedBN)
}

predictBN = function(fittedBN, testData, missingNode){
  grainBN = as.grain(fittedBN)
  grainTree <- compile(grainBN)
  preds = rep(NA,nrow(testData))
  state = NULL
  for(i in 1:nrow(testData)){
    for (j in 1:ncol(testData)){
      state[j] = ifelse(testData[i,j]=="yes","yes","no")
    }
    evidence <- setEvidence(object = grainTree,
                            nodes = colnames(testData),
                            states = state)


    probs = querygrain(object = evidence,
                nodes = missingNode)$S

    preds[i] = ifelse(probs["yes"]>0.5,"yes","no")
  }
  return(preds)
}
```

```
index = sample(seq(0,nrow(asia)), round(nrow(asia)*0.8), replace = F)
train = asia[index,]
test = asia[-index,]
testWithMissing = subset(test, select=-S)


fittedBN = getFittedBN(train)
fittedBN.true = getTrueBN(train)

confMatrix = table(test$S,predictBN(fittedBN, testWithMissing, "S"))
confMatrix.true = table(test$S,predictBN(fittedBN.true, testWithMissing, "S"))

print("Network structure learned using P.C")
print(confMatrix)
print(paste("Accuracy: ", sum(diag(confMatrix))/sum(confMatrix)))

print("True network structure")
print(confMatrix.true)
print(paste("Accuracy: ", sum(diag(confMatrix.true))/sum(confMatrix.true)))
```

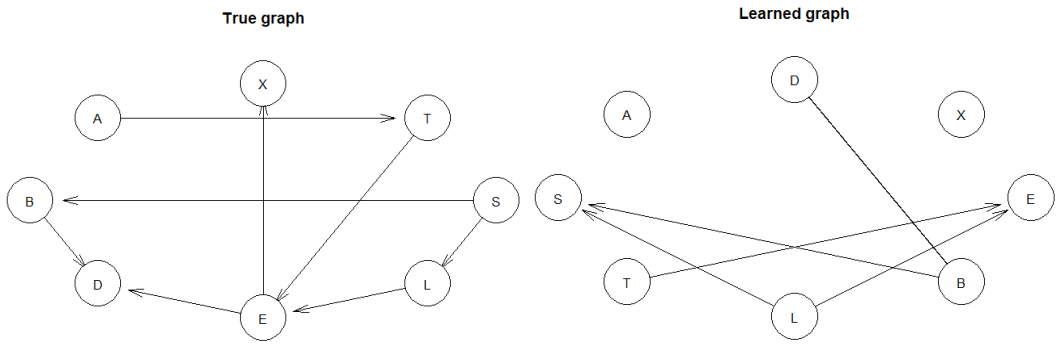
Output

```
[1] "Network structure learned using P.C"

      no yes
  no  320 166
  yes 120 395
[1] "Accuracy:  0.714285714285714"


[1] "True network structure"

      no yes
  no  320 166
  yes 120 395
[1] "Accuracy:  0.714285714285714"
```

Reasoning

The confusion matrix produced from learning the graph and using the correct graphs are the same. The model does not learn the correct graph, as can be seen by plotting the learned model and the true model side by side:

Vikgu708



The learned graph seems to be able to make predictions for S as well as with the true graph. My reasoning for this is that the learned graph includes the correct markov blanket for S = B and L. Inference is carried out with B and L always present, which makes S independent on the rest of the parameters and thus the wrong representation of the edges from the other variables does not matter for predicting S. This means both models only uses B and L to make inference on S, given all other nodes than S.

Both models uses the same nodes for inference, but the conditional distributions for S for the two graphs needs to be estimated equally (or very similar) to produce the same predictions, meaning

$$(1)\ p\left(S\middle|L,B,G_{false}\right) = p(S|L,B,G_{true})$$

The conditional on the graph is only illustrative to show which graph is concerned. However, both graphs uses the same data. The only difference is which parameters are learned. The models learn different parameters as the direction of the edges from S to L and B are opposite between the graphs.

The learned graph will learn the parameters for $p\left(S\middle|L,B,G_{false}\right)$ using maximum likelihood and can thus use this parameter immediately to produce inference. However, the true graph will learn $p(L|S,G_{true})$ and $p(B|S,G_{true})$ and produce inference with Bayes rule to get $(2)\ p(S|L,B,G_{true}) = \frac{p(S,L,B|G_{true})}{p(L|G_{true})p(B|G_{true})} = \frac{p(B|S,G_{true})p(L|S,G_{true})p(S|G_{true})}{p(L|G_{true})*p(B|G_{true})}$.

I suppose that the two expressions in (1) are equal if all factors p(S), p(L) and p(B) in (2) are approximated close to the real values so that the Bayes rule holds. Which is not unlikely considering the large training set and that the true graph is given resulting in more realistic estimations. I am not sure how to prove this formally, but I suppose one could plug in the ML-expression for all parameters above and see if (2) holds for the maximum likelihood. Then (1) also follows, as the same data is used for estimation in both graphs.

Vikgu708

Question 3

The code was mostly copied from question 2, with the only alteration being an addition of an argument evidenceNodes to the predict function, which produces inference given only the evidenceNodes. EvidenceNodes was then set to the market blanket of S in the function call, using the mb-function.

Code

```
library("bnlearn")

data("asia")

set.seed(12345)

if (!require(gRain)) {
  source("https://bioconductor.org/biocLite.R")
  biocLite("RBGL")
}


library(gRain)

getFittedBN = function(trainData){
  graphStructure = pc.stable(trainData)
  fittedBN = bn.fit(cextend(graphStructure), trainData)
  return(fittedBN)
}

getTrueBN = function(trainData){
  graphStructure = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
  fittedBN = bn.fit(cextend(graphStructure), trainData)
  return(fittedBN)
}

predictBN = function(fittedBN, testData, missingNode, evidenceNodes){
  grainBN = as.grain(fittedBN)
  grainTree <- compile(grainBN)
  preds = rep(NA,nrow(testData))
  state = NULL
  for(i in 1:nrow(testData)){
    for (j in evidenceNodes){
      state[j] = ifelse(testData[i,j]=="yes","yes","no")
    }
    evidence <- setEvidence(object = grainTree,
                            nodes = evidenceNodes,
                            states = state, details=0)


    probs = querygrain(object = evidence,
                       nodes = missingNode)$S
```

```
    preds[i] = ifelse(probs["yes"]>0.5,"yes","no")
  }
  return(preds)
}

index = sample(seq(0,nrow(asia)), round(nrow(asia)*0.8), replace = F)
train = asia[index,]
test = asia[-index,]
testWithMissing = subset(test, select=-S)


fittedBN = getFittedBN(train)
fittedBN.true = getTrueBN(train)

confMatrix = table(test$S,predictBN(fittedBN, testWithMissing, "S", mb(fittedB
N,"S")))
confMatrix.true = table(test$S, predictBN(fittedBN.true, testWithMissing,"S",
mb(fittedBN.true, "S")))

print("Network structure learned using P.C")
print(confMatrix)
print(paste("Accuracy: ", sum(diag(confMatrix))/sum(confMatrix)))

print("True network structure")
print(confMatrix.true)
print(paste("Accuracy: ", sum(diag(confMatrix.true))/sum(confMatrix.true)))
```

Output

```
[1] "Network structure learned using P.C"

      no yes
  no  320 166
  yes 120 395
[1] "Accuracy:  0.714285714285714"


[1] "True network structure"

      no yes
  no  320 166
  yes 120 395
[1] "Accuracy:  0.714285714285714"
```

Question 4

Naïve Bayes assumes conditional independence on all variables, meaning $p(x_i, x_j|C) = p(x_i|C)p(x_j|C) \forall i, j$. For a Bayesian network, this implies no edges between variables any two non-class variables (where class variables are the variable to be predicted). This could mean simply a graph with no edge. However, then the bayes clasasifier would simply output p(S) as the likelihoods would be the same for each class (p(A|S)*p(B|S)…. = p(A)*p(B)… when graph has no edges). Therefor, we like to encode as much information from the factors into predicting S without producing an edge between two non-S factors. Therefor, we make each factor dependent on S. Note that we could not have done it the other way around, I.E making S dependent on all factors, as this would make all factors dependent on S as S would be a un-shielded collider for all other factors.

Code

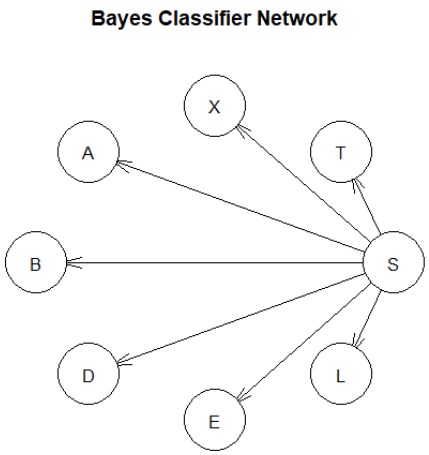
The code is mostly copied from question 2, using only the code for the true network but altering the model structure to `("[S][A|S][T|S][L|S][B|S][D|S][E|S][X|S]")`

```
library("bnlearn")

data("asia")

set.seed(12345)

if (!require(gRain)) {
  source("https://bioconductor.org/biocLite.R")
  biocLite("RBGL")
}


library(gRain)

getFittedBN = function(trainData){
  graphStructure = model2network("[S][A|S][T|S][L|S][B|S][D|S][E|S][X|S]")
  plot(graphStructure, main="Bayes Classifier Network")
  fittedBN = bn.fit(cextend(graphStructure), trainData)
  return(fittedBN)
}


predictBN = function(fittedBN, testData, missingNode){
  grainBN = as.grain(fittedBN)
  grainTree <- compile(grainBN)
  preds = rep(NA,nrow(testData))
  state = NULL
  for(i in 1:nrow(testData)){
    for (j in 1:ncol(testData)){
      state[j] = ifelse(testData[i,j]=="yes","yes","no")
    }
    evidence <- setEvidence(object = grainTree,
                            nodes = colnames(testData),
                            states = state)
```

```
    probs = querygrain(object = evidence,
                       nodes = missingNode)$S

    preds[i] = ifelse(probs["yes"]>0.5,"yes","no")
  }
  return(preds)
}

index = sample(seq(0,nrow(asia)), round(nrow(asia)*0.8), replace = F)
train = asia[index,]
test = asia[-index,]
testWithMissing = subset(test, select=-S)


fittedBN = getFittedBN(train)

confMatrix = table(test$S,predictBN(fittedBN, testWithMissing, "S"))

print("Bayes Classifier")
print(confMatrix)
print(paste("Accuracy: ", sum(diag(confMatrix))/sum(confMatrix)))
```

Output

```
[1] "Bayes Classifier"

      no yes
  no  348 138
  yes 187 328
[1] "Accuracy:  0.675324675324675"
```



Vikgu708

Question 5

Question 2 vs 3

Given the market blanket of S, S is independent on all other factors outside the market blanket. This means that if the market blanket is given, the addition of more nodes does not give any more information for inference on S, meaning that predicting S given S's Markov blanket or given all nodes results in the same predictions. This explains why question 2 and 3 results in the exact same confusion tables.

Question 4 vs 2 and 3

Question 4 simplifies the model by not considering conditional dependence between variables other than S. This is probably why the model produces a somewhat lower accuracy than the other graphs, as less information is included in the inference.

Another possible reason for lower accuracy is misrepresentation of conditional dependencies. The true model has the property that S is independent on all other nodes given B and L, meaning that if we have evidence for B and L no other factors give any further information. Therefor, considering other factors as well as B and L for predicting S is faulty and could produce unreliable results. Similar issues might appear when estimating parameters with the naïve bayes assumption, as the graph is far from the true graph. This is probably the main difference in question 4 vs 2 and 3: all graphs in Q2 and Q3 has the same markov blanket for S and therefor the same conditional dependencies used for inference on S (which happens to be the true graph structure), however naïve bayes assumption does not include this property over S.