732A96: Advance Machine Learning

LAB 1: Graphical Models

Arian Barakat, ariba405
Rebin Hosini, rebho150
Hao chi Kiang, haoki222
Joshua Hudson, joshu107
Carles Sans Fuentes, carsa564

Background

The purpose of the lab is to put in practice some of the concepts covered in the lectures. You can use any data set you like, e.g. you own data, data from public repositories, or data included in the R packages bnlearn and gRain. Check for instance https://www.bnlearn.com/documentation. The learning.test, asia or alarm data sets should suffice. Some questions may be easier to solve with one data set than with the others and, thus, you may need to try with different data sets.

Question 1

Question:
Show that multiple runs of the hill-climbing algorithm can return non-equivalent DAGs. Explain why this happens. Hint: Check the function hc in the bnlearn package. Note that you can specify the initial structure, the number of random restarts, the score, and the equivalent sample size (a.k.a imaginary sample size) in the BDeu score. Use these options to answer the question. You may also want to use the functions plot, arcs, vstructs, cpdag and all.equal.

Answer:

For this assignment we'll use the *alarm* dataset as it has the most number of nodes/variables among the three datasets, which increases the search space for the **hc()**-algorithm. The initial structure will be set to NULL (since we have no knowledge about the data as of this moment), the score criteria is set to the default (BIC), number of random restarts to 10 and the seed to 12345.

The key to undertand why we obtain non-equivalent graphs is that the hc-algorithm is a greedy algorithm that adds, removes or reverses edges between nodes in order to maximize the chosen score-criteria. By definition, two graphs are equivalent graphs if they represent the same dependencies (they have the same adjacencies and unshielded colliders), which isn't the case in the two constructed structures seen in figure 1

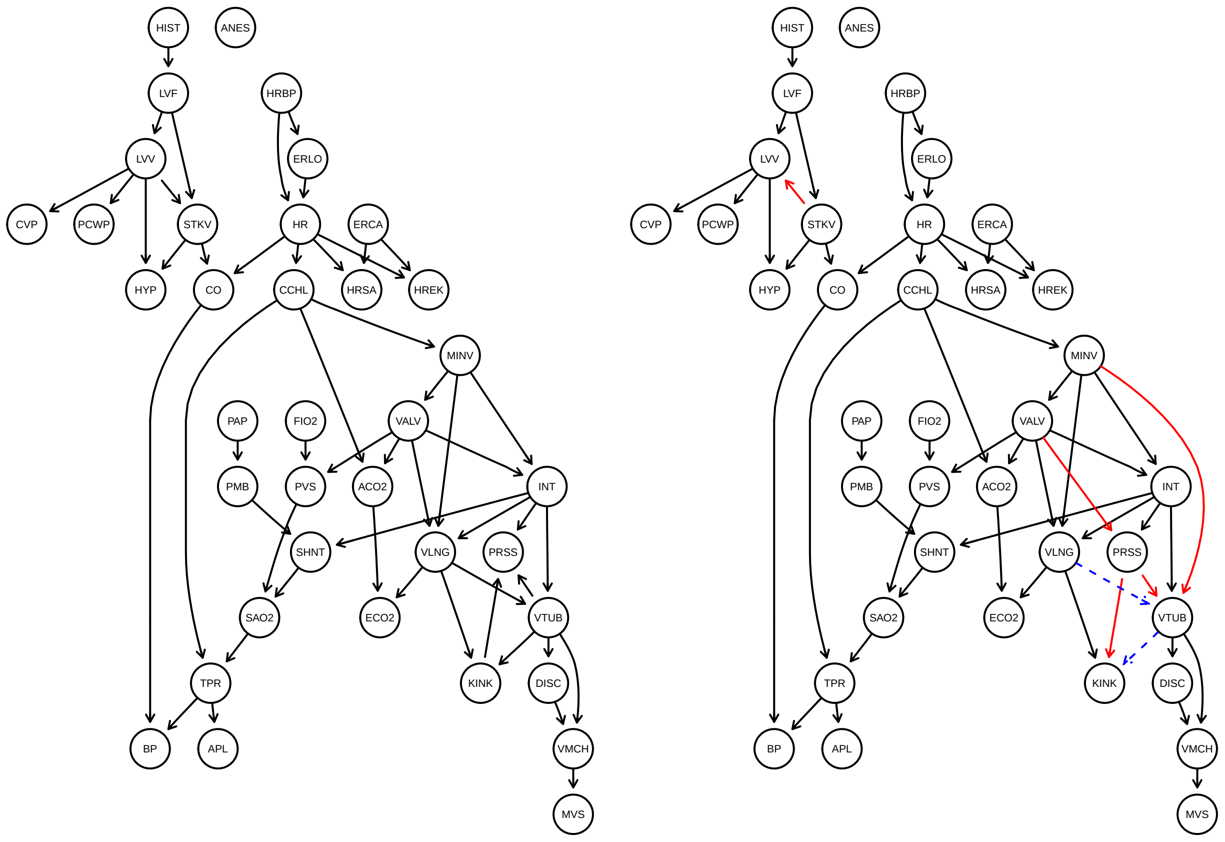

Figure 1: Bayesian Networks, Difference between non-equivalent DAGs

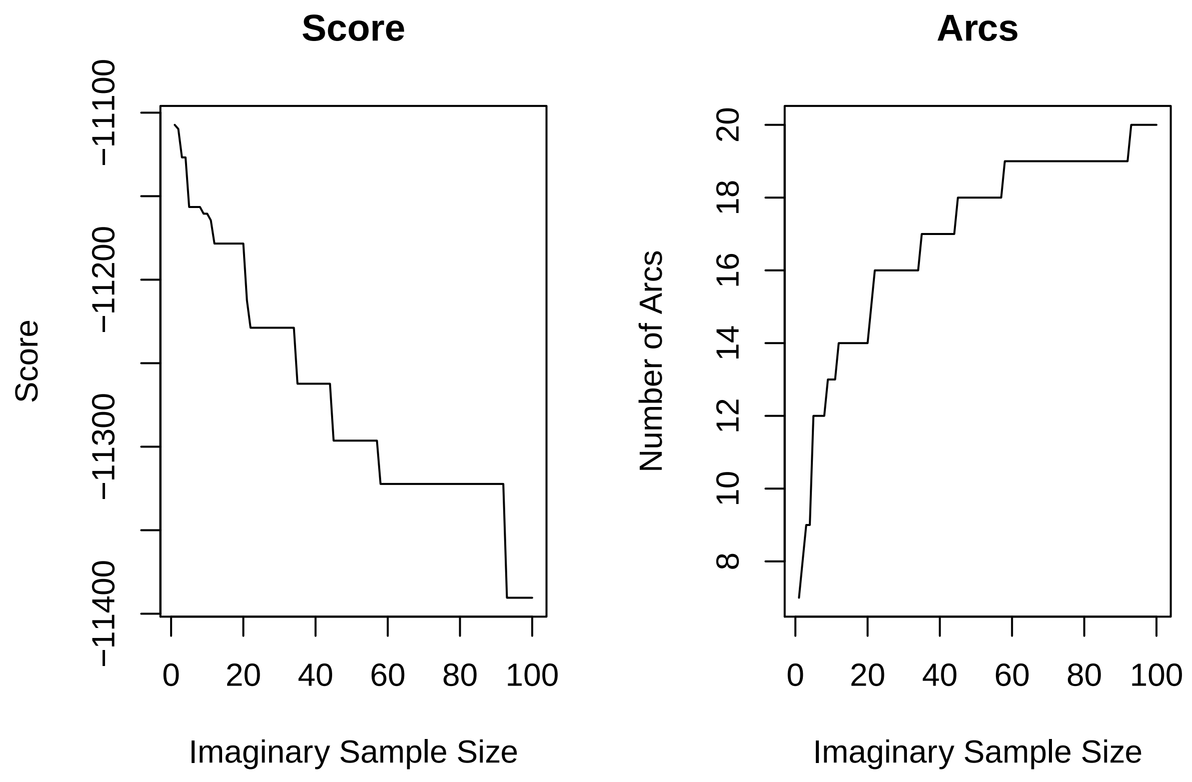

Figure 2: Graph score (BDeu) and arc density with respect to Imaginary Sample Size

Question 2

Question:
Show that increasing the equivalent sample size (a.k.a imaginary sample size) in the BDeu score decreases regularization. Explain why this happens. Hint: Run some structure learning algorithm (e.g. check the function hc in the bnlearn package) multiple times and show that it tends to end in more densely connected DAGs when large imaginary sample sizes are used. Or produce a histogram of the scores of a random sample of DAGs with different imaginary sample sizes and see if they come closer or not one to another (e.g. check the functions hist, random.graph, sapply and score in the bnlearn and core packages).

Answer:

Figure 2 presents the result and from the figure we can see that as the user-defined imaginary sample size (iss) increases the graph grows more densely but at the same time decreases in score. To explain this we will use a simple example where we use the balanced beta distribution as the prior (instead of the dirichlet prior) $beta(\alpha, \beta)$, where $\alpha + \beta = ISS$. A large value of ISS implies that we are very certain that the distribution is centered around 0.5. If this is the case we know that

$$p(A|B=0) \sim 0.5$$
$$p(A|B=1) \sim 0.5$$

Given this, we might argue that a model without an edge between the nodes A and B is similar to a model with an edge between the nodes. However, since we rather have causality in a model we'll chose the latter one.With this in mind, the graph will grow more densely.

Morever, the BDeu score is a function of the graph's dimensions and as the dimensions increase (a more dense graph), the score gets more penalized.

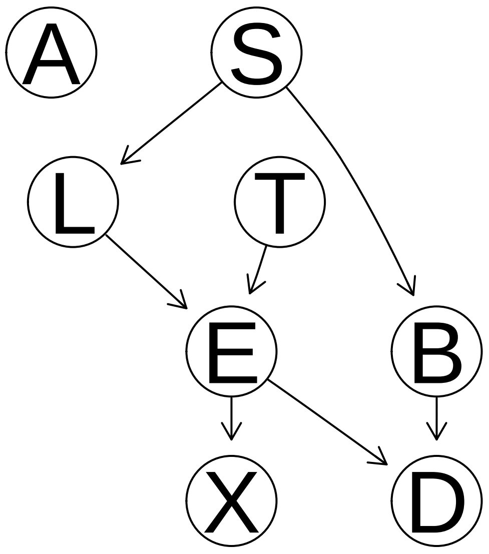

Figure 3: Bayesian Network using Hill-Climbing, Asia Data

Question 3

Question:

You already know the LS algorithm for inference in BNs, which is an exact algorithm. There are also approximate algorithms for when the exact ones are too demanding computationally. Compare the answers given to some queries by exact and approximate inference algorithms. When running the approximate algorithm several times, why may we obtain different answers? Is the approximate algorithm equally accurate when the query includes no observed nodes and when it includes some observed nodes? Hint: For exact inference, you may need the functions bn.fit and as.grain from the bnlearn package, and the functions compile, setFinding and querygrain from the package gRain. For approximate inference, you may need the functions prop.table, table and cpdist from the bnlearn package.

Answer:

From the learned structure in figure 3 we can construct some queries that will be investigated in this assignment, seen in table 1.

Figure 4 shows the values obtained from the exact inference as well as the approximate inference for a given number of iteration. From the plot(s) we can see that the approximate algorithm gives us different values in every iterations as it is based on markov-chain simulations.

Morever, as we add more evidence (observed nodes) we can see that the approximate algorithm (inference) gets less accurate as it is based on observations that are compatible with our evidence.

Table 1: Queries

Event	Evidence
L = Yes	None
L = Yes	S = Yes
E = Yes	L = Yes & T = No

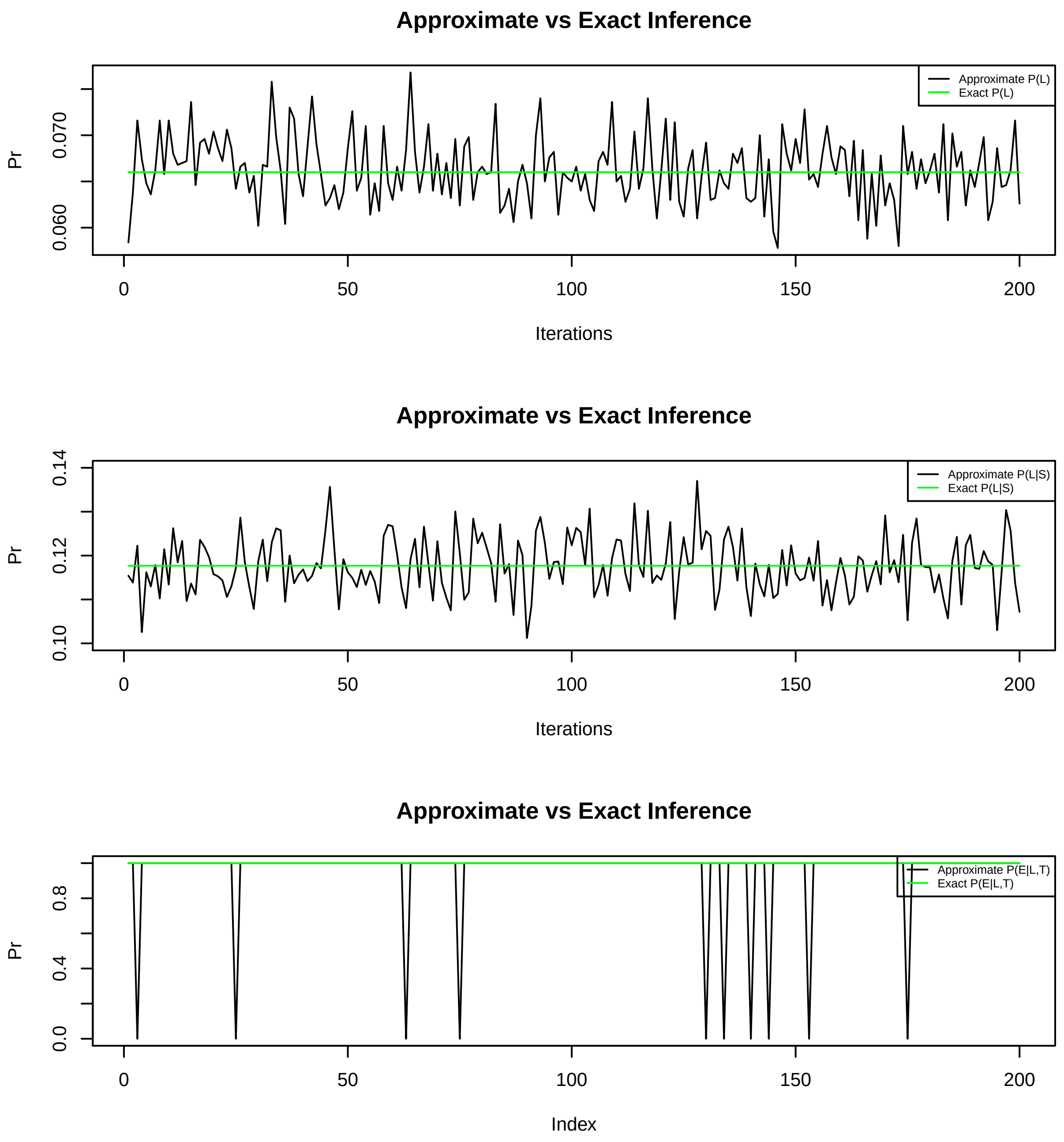

Figure 4: Approximate and Exact Answer with respect to iterations for different Queries

Question 4

Question:

There are 29281 DAGs with five nodes. Compute approximately the fraction of the 29281 DAGs that represent different independence models. In the light of the result obtained, would you say that it is preferable to perform structure learning in the space of DAGs or in the space of essential graphs? Hint: You do not need to produce the 29281 DAGs. Instead you can sample DAGs uniformly, convert each DAG in the sample to its essential graph (a.k.a completed partial DAG or CPDAG), and then count how many different essential graphs you have left. You can do all this with the functions random.graph, cpdag, lapply, unique and length in the bnlearn and core packages. Try different values for the parameters every and burn.in in the function random.graph.

Answer:

By running the code we obtain an estimate number of unique essential graphs equal to 52.547 percent of all the created graphs with 5 nodes. From the result one could argue that is better to learn a network structure from the search space of essential graph. One could also argue that this approach would make score based algorithms less prone to get caught in local minima.

Appendix

```r
# Setup

library(bnlearn)
source("..//Source/biocLite.R")
#source("http://bioconductor.org/biocLite.R")
#biocLite(c("graph", "Rgraphviz", "RBGL"))
library(gRain)
library(stringr)
library(knitr)


# Question 1

set.seed(12345)
countinue <- TRUE
while(countinue){
  hc1 <- hc(alarm, restart = 10)
  hc2 <- hc(alarm, restart = 10)
  continue <- ifelse(all.equal(vstructs(hc1), vstructs(hc2)) == TRUE, TRUE, FALSE)
  if (continue !=TRUE){
    par(mfrow = c(1,2))
    graphviz.compare(hc1, hc2)
    par(mfrow = c(1,1))
    break
  }
}



# Question 2

totSampleSize <- 100
scores <- vector("numeric", length = totSampleSize)
nrArcs <- vector("numeric", length = totSampleSize)

for(i in 1:totSampleSize){
  learned_DAG <- hc(asia, score = "bde", iss = i)
  scores[i] <- score(learned_DAG, asia)
  nrArcs[i] <- nrow(arcs(learned_DAG))
}

par(mfrow = c(1,2))
plot(y = scores, x = 1:totSampleSize,
     type = "l", main = "Score",
     xlab = "Imaginary Sample Size",
     ylab = "Score")
plot(y = nrArcs, x = 1:totSampleSize,
     type = "l", main = "Arcs",
     xlab = "Imaginary Sample Size",
     ylab = "Number of Arcs")
par(mfrow = c(1,1))
```

```r
# Question 3


# Learn Structure
BN_structure <- hc(asia)

# Fitting Network
BN_fitted <- bn.fit(BN_structure, method = "mle", asia)
graphviz.plot(BN_fitted) # Plot Network to see dependencies



probTable <- data.frame("Event" = c("L = Yes", "L = Yes", "E = Yes"),
                        "Evidence" = c("None", "S = Yes", "L = Yes & T = No"))
kable(probTable, caption = "Queries")

#Get conditional probability queries: probability of lungcancer given smoking.
smokeProb <- cpquery(BN_fitted, event = (L == "yes"),
                     evidence = (S == "yes"))

#Smoke probabilities unconditional approxpar$L,
unConSmokeApprox <- cpquery(BN_fitted, event = (L == "yes"),
                            evidence = TRUE)

#Get conditional probability queries: probability of shortness of breath (dyspnoea)
#given smoking & visit to asia
dyspnoeaProb <- cpquery(BN_fitted, event = (D == "yes"),
                        evidence = (A == "yes") & (S == "yes"))

#Get unconditional probability queries: probability of shortness of breath (dyspnoea)
#given smoking & visit to asia
unCondyspnoeaProb <- cpquery(BN_fitted, event = (D == "yes"),
                             evidence = TRUE)


#Exact Inference with gRain (use the same tree and parameters from the bnlearn)

# Creating Junction tree
junTree <- compile(as.grain(BN_fitted))

# Setting Evidence
smokeEvidence <- setFinding(junTree, nodes = "S", states = "yes")
dyspnoeaEvidence <- setFinding(junTree, nodes = c("A","S"), states = c("yes","yes"))

# Get probabilities (unconditional)
smokeProbExact <- querygrain(junTree, nodes = "L")$L

#Get probabilities (conditional)
smokeProbExactCon <- querygrain(smokeEvidence, nodes = "L")$L
dyspnoeaProbExactCon <- querygrain(dyspnoeaEvidence, nodes = "D")$D
```

```r
#Approximate plot
approxPlot <- function(n=100, BN_structure, BN_fitted){
  suppressMessages(suppressWarnings(require("gRain")))
  approxNet <- BN_structure
  approxPar <- BN_fitted
  junTree <- compile(as.grain(approxPar))
  smokeEvidence <- setFinding(junTree, nodes = "S", states = "yes")
  smokeProb <- c()
  smokeProb2 <- c()
  smokeProb3 <- c()
  smokeprobCon <- c()
  smokeProbCon2 <- c()
  smokeprobUncon <- c()

  for(i in 1:n){
    smokeProb[i] <- cpquery(approxPar, event = (L == "yes"),
                            evidence = (S == "yes"))
    smokeProb2[i] <- cpquery(approxPar, event = (L == "yes"),
                            evidence = TRUE)
    smokeProb3[i] <- cpquery(approxPar, event = (E == "yes"),
                            evidence = (L == "yes") & (T == "yes"))

    smokeEvidence <- setFinding(junTree, nodes = "S", states = "yes")
    smokeEvidence2 <- setFinding(junTree, nodes = c("L","T"), states = c("yes","no"))
    smokeprobCon[i] <- querygrain(smokeEvidence, nodes = "L")$L[2]
    smokeProbCon2[i] <- querygrain(smokeEvidence2, nodes = "E")$E[2]

    smokeprobUncon[i] <- querygrain(junTree, nodes = "L")$L[2]
  }

  return(data.frame("Conditional" = smokeProb,
                    "Unconditional" = smokeProb2,
                    "Conditional Exact" = smokeprobCon,
                    "Unconditional Exact" = smokeprobUncon,
                    "Conditional 2"  = smokeProb3,
                    "Conditional Exact 2" = smokeProbCon2))
}


par(mfrow = c(3,1))
#Unconditional
plot(approxPlot(n = 200, BN_structure = BN_structure, BN_fitted = BN_fitted)[,2], type = "l", col = "bla
      xlab = "Iterations",main = "Approximate vs Exact Inference", ylab = "Pr")
lines(approxPlot(n = 200, BN_structure = BN_structure, BN_fitted = BN_fitted)[,4], type = "l", col = "g
legend("topright", legend = c("Approximate P(L)", "Exact P(L)"),
       lty = c(1,1), col = c("black", "green"),
       cex = 0.6)

#Conditional
plot(approxPlot(n = 200, BN_structure = BN_structure, BN_fitted = BN_fitted)[,1], type = "l", xlab = "I
     ylim = c(0.1, 0.14), main = "Approximate vs Exact Inference",
     ylab = "Pr")
lines(approxPlot(n = 200, BN_structure = BN_structure, BN_fitted = BN_fitted)[,3], type = "l", col = "g
legend("topright", legend = c("Approximate P(L|S)", "Exact P(L|S)"),
       lty = c(1,1), col = c("black", "green"),
       cex = 0.6)

#2 Conditions
plot(approxPlot(n = 200, BN_structure = BN_structure, BN_fitted = BN_fitted)[,5], type = "l", col = "bla
     main = "Approximate vs Exact Inference", ylab = "Pr")
lines(approxPlot(n = 200, BN_structure = BN_structure, BN_fitted = BN_fitted)[,6], type = "l", col = "g
legend("topright", legend = c("Approximate P(E|L,T)", "Exact P(E|L,T)"),
       lty = c(1,1), col = c("black", "green"),
       cex = 0.6)
par(mfrow = c(1,1))








# Question 4

numberOfGraphs <- 25000
sampledDAGs <- random.graph(nodes = LETTERS[1:5],
                              num = numberOfGraphs,
                              method = "ic-dag",
                              burn.in = 12 * length(LETTERS[1:5])^2,
                              every = 1)
uniqueDAGs <- unique(sampledDAGs)
equivDAGs <- lapply(uniqueDAGs, cpdag)
percentUnique <- round((length(unique(equivDAGs))/length(uniqueDAGs))*100, digits = 3)
```