732A96: Advance Machine Learning

LAB 1: Graphical Models

Arian Barakat/ariba405

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 question, 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) and the number of random restarts to 1.

From the result, we can observe that we obtain a non-equivalent graph already after 3 runs and that different arc(s) is: LVV->PCWP. According to the definition, two graphs are equivalent graphs if the represent the same dependencies (they have the same adjacencies and unshielded colliders), which isn't the case in the two constructed graphs as seen figure 1.

The reason why we obtain non-equivalent graphs is due to the fact the hill-climbing algorithm is a greedy algorithm. The algorithm adds, removes and reverses edges in order to maximize the score criteria. Due to the greedy property, the possibility of getting stuck in a local minimum is present. In our case, the large search space makes the algorithm more prone to get caught in local minima as the different combinations of nodes introduce relatively more complex landscape.

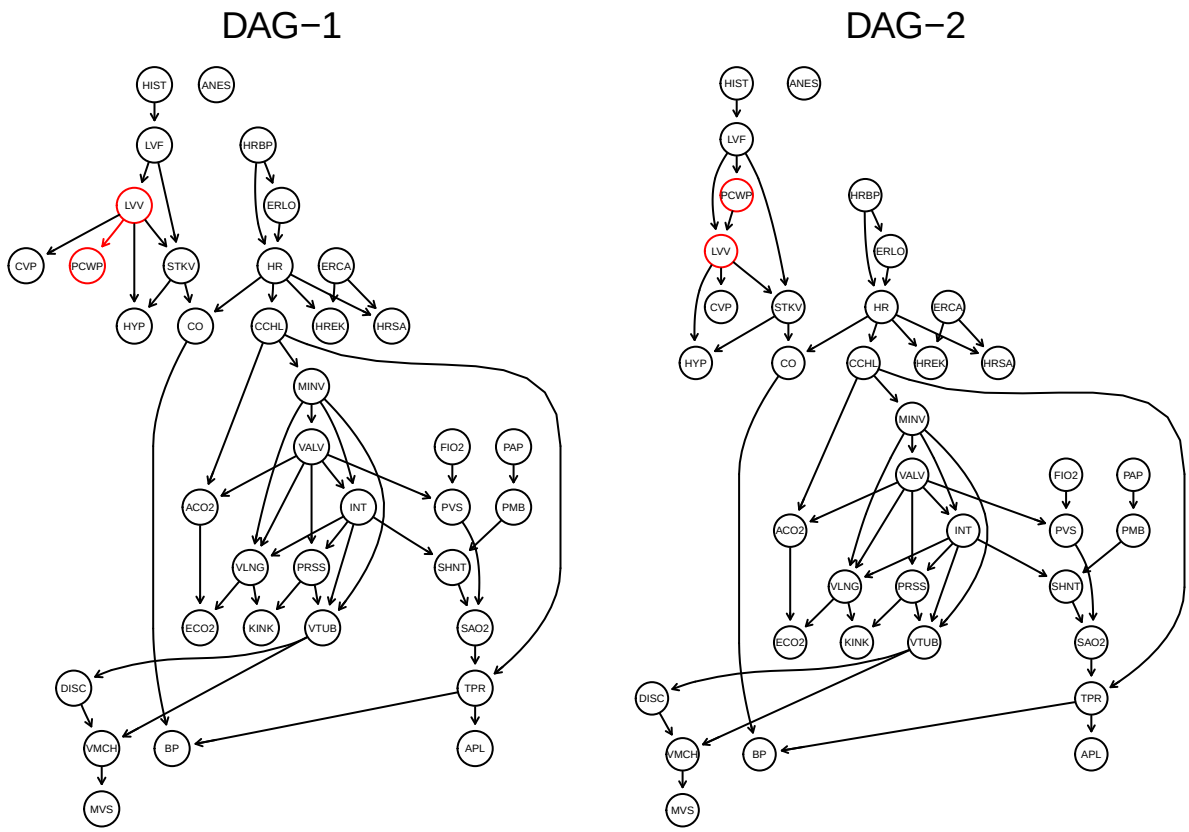

Figure 1: Bayesian Networks, non-equivalent graph DAG-1 and DAG-2

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:

From figure 2 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 the score. The explanation for this behavior is that the BDeu score, which requires the prior parameter (equivalent/imaginary sample size), is sensitive to values of the parameter iss.

In a general setting the prior takes the form of a balanced/uniform Dirichlet distribution (for multinomial data), however in case of binary outcome, the prior can be collapsed to a beta distribution. Higher values of ISS implies that we are very certain about that the parameter distribution is centered around 0.5 (in the case of binary variable/nodes), which implies that the posterior distribution of the parameter will coincide with the prior as iss $\rightarrow \infty$. In other words, as iss increases:

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

Given this, we might argue that a model without an edge between nodes A and B is similar to a model with an edge between the same nodes. However, in such a case, Bayesian Networks has the tendency to favor the presence of an edge, which makes the graph grow more densely. As the graph grows more densely, the score will be penalized as a result of the larger dimensions.

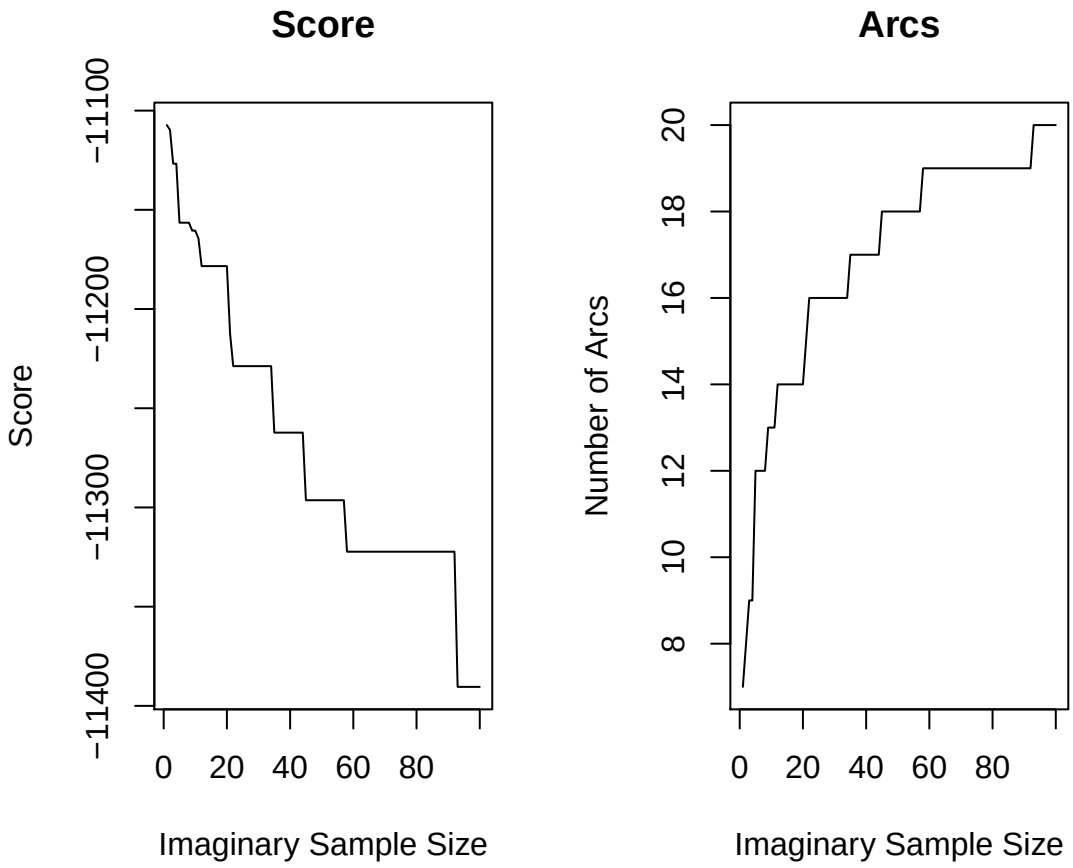

Figure 2: Graph score and arc density with respect to Imaginary Sample Size

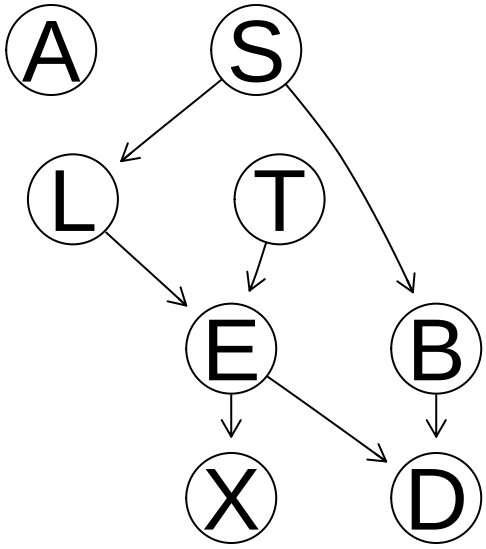

Figure 3: Bayesian Network using Hill-Climbing, Asia Data

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 DAG in figure 3 we can construct some queries, which are shown in table 1.

Table 1: Queries

Event	Evidence
(L == 'yes')	(S == 'yes')
(X == 'no')	(S == 'yes') & (T == 'yes')
(B == 'yes')	(NULL)

Figure 4 present the result for the queries using the exact approach but also the approximate approach. For every query, the approximate approach has been run 1000 times. From the plot(s) we can see that the approximate algorithm gives us different values in every iteration. The key to understanding why the approximate approach gives us different answers is the fact the algorithm is using Monte-Carlo simulations to obtain the result of our query.

The plots also reveal that as we add more evidence (observed nodes), the approximate approach has the tendency to get less accurate, which especially can be seen in the second graph in figure 4. The reason behind this behavior is that the algorithm is based on observations that are compatible with the provided evidence.

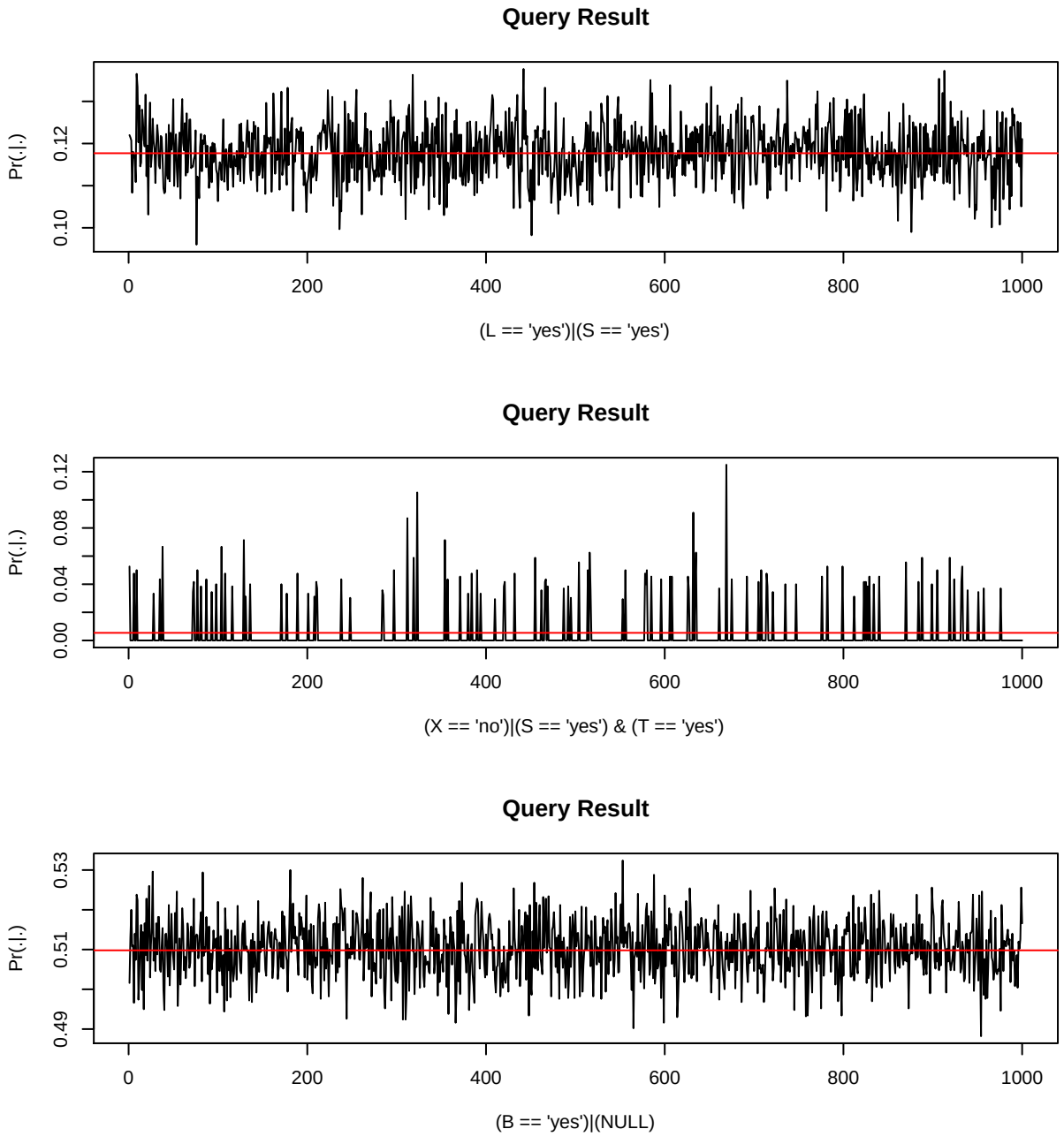

Figure 4: Approximate and Exant Answer with respect to iterations for different Queries, n = 1000

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:

See code in appendix.

By running the code we obtain a number of unique essential graphs equal to 51.997 percent of all the created graphs with 5 nodes. Given this, one could argue that it's preferred to perform structure learning in the space of essential graphs instead of DAGs, which would in a sense make score based algorithms less prone to get caught in local minima.

Appendix

```
# Setup

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


# Question 1

data("learning.test")
data("asia")
data("alarm")



count <- 0
is_equivalent <- TRUE
DAG <- list()
set.seed(12345)

while(is_equivalent){
  count <- count + 1
  DAG[[1]] <- hc(alarm, start = NULL, restart = 1)
  DAG[[2]] <- hc(alarm, start = NULL, restart = 1)
  test <- all.equal(cpdag(DAG[[1]]),cpdag(DAG[[2]]))
  is_equivalent <- ifelse(any(test == "TRUE"), TRUE, FALSE)
}

DAG_arcs <- lapply(DAG, function(x){
  x_arcs <- arcs(x)
  out <- apply(x_arcs, MARGIN = 1, FUN = function(x){paste(x, collapse = "->")})
  return(out)
})

differentArcs <- setdiff(DAG_arcs[[1]], DAG_arcs[[2]])


par(mfrow = c(1,2))
graphviz.plot(DAG[[1]],
              main = "DAG-1",
              highlight = list(arcs =
                               matrix(c("LVV", "PCWP"),
                                      ncol = 2,
                                      dimnames =
                                        list(NULL,
                                             c("from","to"))),
                               nodes = c("LVV", "PCWP")))
graphviz.plot(DAG[[2]],
```

```
             main = "DAG-2",
             highlight = list(nodes = c("LVV", "PCWP")))
par(mfrow = c(1,1))

# 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))


#Question 3


DAG_asia <- hc(asia)
DAG_asia_fit <- bn.fit(DAG_asia, data = asia, method = "mle")

graphviz.plot(DAG_asia_fit)

# These queries have been decided from the learned structure
queries <- matrix(data = c("(L == 'yes')", "(S == 'yes')",
                           "(X == 'no')", "(S == 'yes') & (T == 'yes')",
                           "(B == 'yes')", "(NULL)"),
                  ncol = 2,
                  byrow = TRUE,
                  dimnames = list(NULL, c("Event", "Evidence")))

knitr::kable(queries, caption = "Queries")



# Exact
junctionTree_asia <- compile(as.grain(DAG_asia_fit))

queryExact <- vector("numeric", length = nrow(queries))
for(i in 1:nrow(queries)){
```

```
  if(str_detect(queries[i,2], "&")){
    evid <- str_split(queries[i,2], pattern = "&", simplify = TRUE)
    evid <- str_extract_all(evid, pattern = "[A-z+]", simplify = TRUE)
  } else{
    evid <- str_extract_all(queries[i,2], pattern = "[A-z+]", simplify = TRUE)
  }


  if(str_c(evid, collapse = "") == "NULL"){
    # No evidence to set

    event <- str_extract_all(queries[i,1], pattern = "[A-z+]", simplify = TRUE)
    queryRes <- querygrain(junctionTree_asia, nodes = event[1])
    queryExact[i] <- queryRes[[1]][str_c(event[-1], collapse = "")]

  } else{
    currentEvidence <- setEvidence(junctionTree_asia,
                                   nodes = evid[,1],
                                   states = apply(evid[,-1, drop = FALSE],
                                                  1,
                                                  function(x){str_c(x, collapse = "")}))

      event <- str_extract_all(queries[i,1], pattern = "[A-z+]", simplify = TRUE)
      queryRes <- querygrain(currentEvidence, nodes = event[1])
      queryExact[i] <- queryRes[[1]][str_c(event[-1], collapse = "")]
  }
}

# Approximate

queryApprox <- list()
for(i in 1:nrow(queries)){

  evid <- str_extract_all(queries[i,2], pattern = "[A-z+]", simplify = TRUE)
  if(str_c(evid, collapse = "") == "NULL"){
    queryApprox[[i]] <- sapply(1:1000, function(i){cpquery(DAG_asia_fit,
                            event = eval(parse(text = queries[i, 1])),
                            evidence = TRUE)})

  } else {
      queryApprox[[i]] <- sapply(1:1000,FUN = function(i){cpquery(DAG_asia_fit,
                            event = eval(parse(text = queries[i, 1])),
                            evidence = eval(parse(text = queries[i, 2])))})

  }

}
```

```r
queries_name <-  apply(queries,MARGIN = 1,
                       FUN = function(x){str_c(x, collapse = "|")})

par(mfrow = c(3,1))
for(i in 1:nrow(queries)){
  plot(y = queryApprox[[i]],
       x = 1:1000,
       type = "l",
       main = "Query Result",
       xlab = queries_name[i],
       ylab = "Pr(.|.)")
  abline(h = queryExact[i], col = "red")
}
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)
```