Lab1 Group Report

Aman Kumar Nayak , Ali Etminan , Mohsen Pirmoradiyan , Ahmed Ali Nawir Alhasan

9/13/2020

(1)Show that multiple runs of the hill-climbing algorithm can return non-equivalent Bayesian network (BN) structures. Explain why this happens.

The dataset consists of 8 discrete variables each of which is a binary feature:

- D (dyspnoea), a two-level factor with levels yes and no.
- T (tuberculosis), a two-level factor with levels yes and no.
- L (lung cancer), a two-level factor with levels yes and no.
- B (bronchitis), a two-level factor with levels yes and no.
- A (visit to Asia), a two-level factor with levels yes and no.
- S (smoking), a two-level factor with levels yes and no.
- X (chest X-ray), a two-level factor with levels yes and no.
- E (tuberculosis versus lung cancer/bronchitis), a two-level factor with levels yes and no.

Lauritzen and Spiegelhalter (1988) motivate this example as follows:

"Shortness-of-breath (dyspnoea) may be due to tuberculosis, lung cancer or bronchitis, or none of them, or more than one of them. A recent visit to Asia increases the chances of tuberculosis, while smoking is known to be a risk factor for both lung cancer and bronchitis. The results of a single chest X-ray do not discriminate between lung cancer and tuberculosis, as neither does the presence or absence of dyspnoea."

Standard learning algorithms are not able to recover the true structure of the network because of the presence of a node (E) with conditional probabilities equal to both 0 and 1. Monte Carlo tests seems to behave better than their parametric counterparts.

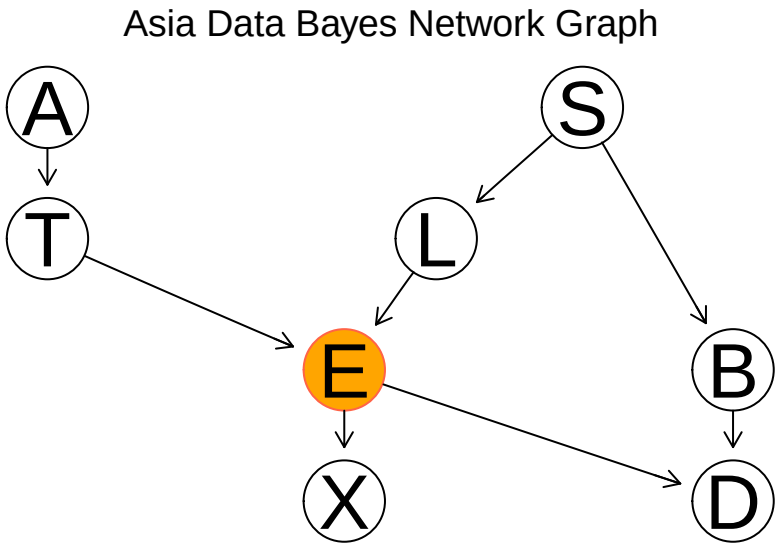
```
data("asia")
head(asia)
```

```
##    A   S   T  L   B   E   X   D
## 1 no yes  no no yes  no  no yes
## 2 no yes  no no  no  no  no  no
## 3 no  no yes no  no yes yes yes
## 4 no  no  no no yes  no  no yes
## 5 no  no  no no  no  no  no yes
## 6 no yes  no no  no  no  no yes
```

Based on the aforementioned explanations an initial structure is defined as follow:

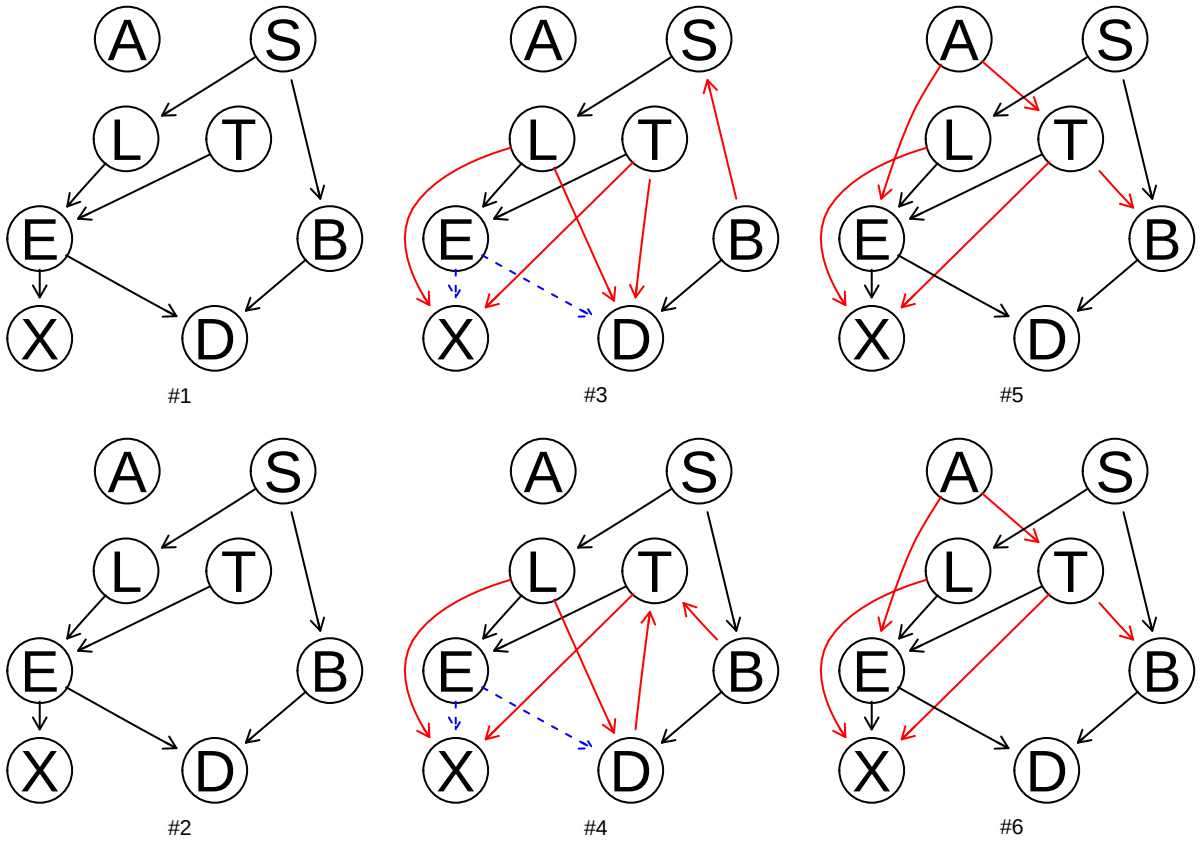
```
dag.True  = bnlearn::model2network(string = "[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")

bnlearn::graphviz.plot(dag.True, main = "Asia Data Bayes Network Graph"
                        #,  layout = "neato",
                        ,highlight = list(nodes = "E" ,  col = "tomato",
                                          fill = "orange") )
```



(1)

Running hill-climbing algorithm multiple times initialized with an empty graph:

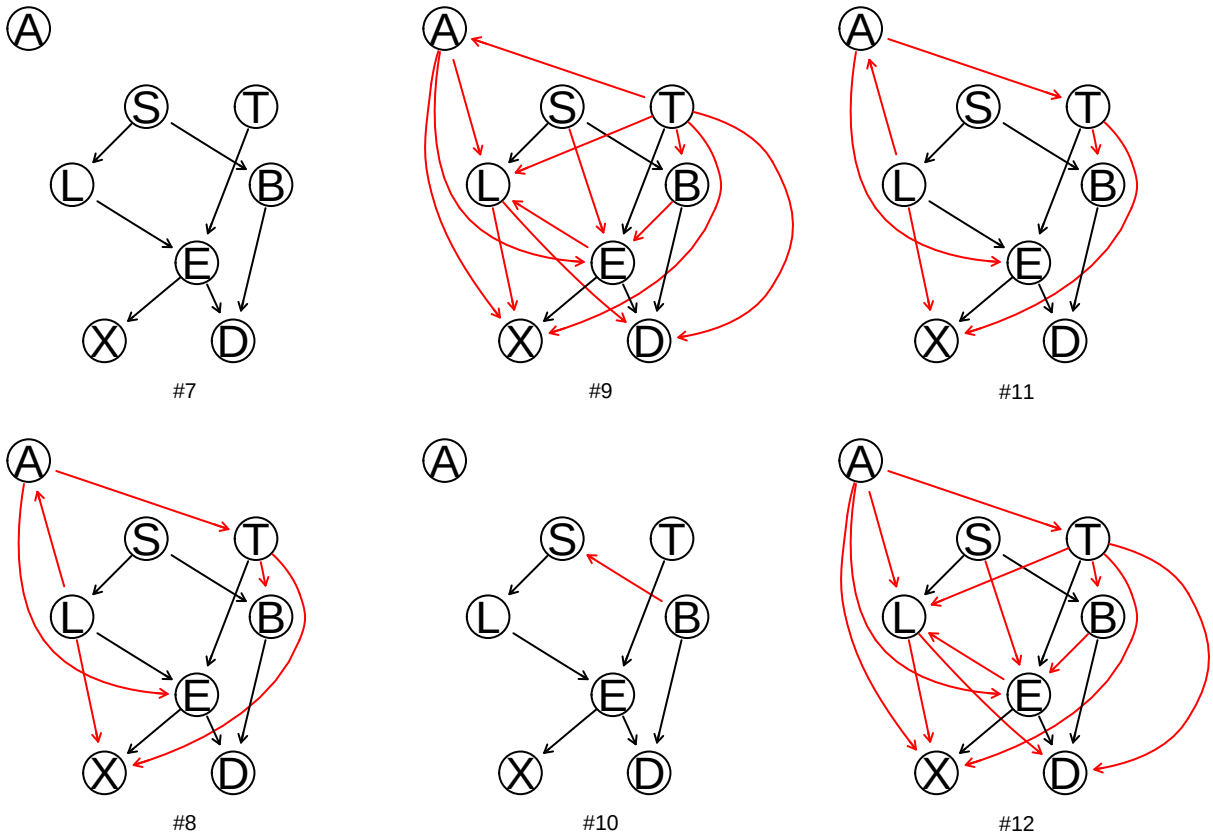



```
## Are model 1 and 2 equivalent:  TRUE

## Are model 3 and 4 equivalent:  Different number of directed/undirected arcs
```

In the first experiment, we tried six different runs changing the starting graph, the number of restarts, and different scores as shown in the plot above, we could see that the graphs are not always equal to each other.

Highlighted arcs are the ones that are different from the first model in the top left corner.




```
## Are model 7 and 9 equivalent:  Different number of directed/undirected arcs
```

```
## Are model 9 and 12 equivalent:  Different arc sets
```

In the second experiment, we try another 6 runs all using "bde" score but using different is "imaginary sample size", we notice the network become denser when increasing "iss" parameter because it gives more weight to the Dirichlet prior of the parameters which mean there is less certainty if there is a dependency between parameters so adding more arcs will avoid giving information about conditional independencies.

The different results are because the Hill-Climbing algorithm is a greedy algorithm and does not try all the possible outcomes and because it doesn't distinguish the direction of the arc in certain cases it will eventually get stuck in local minima and given the large search space that local minima could be different in each run and each configuration.

(2) Learn a BN from 80 % of the Asia dataset.

a)Learn both the structure and the parameters. Use any learning algorithm and settings that you consider appropriate.

```
#Splitting the data set into train and test
n = dim(asia)[1]
id = sample(1:n,floor(.8*n))
```

```
train = asia[id,]
test = asia[-id,]
```

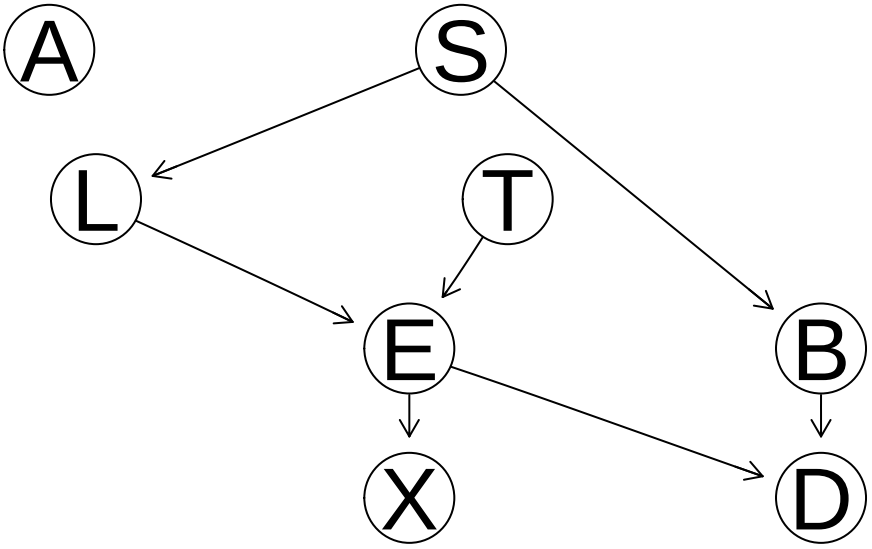
After splitting the data we'll start the learning procedure:

1)Structural learning implementing the hill-climbing algorithm:

```
#Implementing Score-based algorithm
train_DAG_hc = hc(train)
train_DAG_hc
```

```
##
##   Bayesian network learned via Score-based methods
##
##   model:
##    [A][S][T][L|S][B|S][E|T:L][X|E][D|B:E]
##   nodes:                                 8
##   arcs:                                  7
##     undirected arcs:                     0
##     directed arcs:                       7
##   average markov blanket size:           2.25
##   average neighbourhood size:            1.75
##   average branching factor:              0.88
##
##   learning algorithm:                    Hill-Climbing
##   score:                                 BIC (disc.)
##   penalization coefficient:              4.147025
##   tests used in the learning procedure:  77
##   optimized:                             TRUE
```

```
graphviz.plot(train_DAG_hc)
```



2)now we can fit the parameters:

```r
graph_fit = bn.fit(train_DAG_hc, train)
graph_fit
```

```
##
##   Bayesian network parameters
##
##   Parameters of node A (multinomial distribution)
##
## Conditional probability table:
##     no   yes
## 0.991 0.009
##
##   Parameters of node S (multinomial distribution)
##
## Conditional probability table:
##     no   yes
## 0.502 0.498
##
##   Parameters of node T (multinomial distribution)
##
## Conditional probability table:
##      no    yes
## 0.9915 0.0085
##
##   Parameters of node L (multinomial distribution)
##
## Conditional probability table:
##
##      S
## L             no        yes
##   no  0.98705179 0.88253012
##   yes 0.01294821 0.11746988
##
##   Parameters of node B (multinomial distribution)
##
## Conditional probability table:
##
##      S
## B            no       yes
##   no  0.7036853 0.2856426
##   yes 0.2963147 0.7143574
##
##   Parameters of node E (multinomial distribution)
##
## Conditional probability table:
##
## , , L = no
##
##      T
## E     no yes
##   no   1   0
##   yes  0   1
##
```

```
## , , L = yes
##
##      T
## E     no yes
##   no   0   0
##   yes  1   1
##
##
##   Parameters of node X (multinomial distribution)
##
## Conditional probability table:
##
##      E
## X              no         yes
##   no  0.955783230 0.006872852
##   yes 0.044216770 0.993127148
##
##   Parameters of node D (multinomial distribution)
##
## Conditional probability table:
##
## , , E = no
##
##      B
## D            no       yes
##   no  0.8962766 0.2132313
##   yes 0.1037234 0.7867687
##
## , , E = yes
##
##      B
## D            no       yes
##   no  0.2647059 0.1428571
##   yes 0.7352941 0.8571429
```

b)Use the BN learned to classify the remaining 20 % of the Asia dataset in two classes: S = yes and S = no. In other words, compute the posterior probability distribution of S for each case and classify it in the most likely class.

To do an exact inference for computing the posterior probability distribution of S, gRain package will be implemented. To do so we have to export the fitted bn object to gRain:

```
fitted_grain = as.grain(graph_fit)
fitted_grain
```

```
## Independence network: Compiled: TRUE Propagated: FALSE
##   Nodes: chr [1:8] "A" "S" "T" "L" "B" "E" "X" "D"
```

The next step is compiling the graph to be moralized and triangulated:

```
fitted_grain_comp = compile(fitted_grain, propagate = TRUE)
summary(fitted_grain_comp)
```

```
## Independence network: Compiled: TRUE Propagated: TRUE
##  Nodes : Named chr [1:8] "A" "S" "T" "L" "B" "E" "X" "D"
##  - attr(*, "names")= chr [1:8] "A" "S" "T" "L" ...
##  Number of cliques:                 6
##  Maximal clique size:               3
##  Maximal state space in cliques:    8
```

Now we have to set the evidences for each case and request the probability at node S:

```
n = nrow(test)
found_nodes = c("A", "T", "L", "B", "E", "X", "D")
result = vector(length = n)
for (i in 1:n) {
  dag_find = setEvidence(fitted_grain_comp,
                         nodes = found_nodes,
                         states = c(as.character(test$A[i]),
                                    as.character(test$T[i]),
                                    as.character(test$L[i]),
                                    as.character(test$B[i]),
                                    as.character(test$E[i]),
                                    as.character(test$X[i]),
                                    as.character(test$D[i])))
  s_quary = querygrain(dag_find, nodes="S")
  result[i] = ifelse(s_quary$S["no"] > s_quary$S["yes"], "no", "yes")

}
```

c)Report the confusion matrix, i.e.,true/false positives/negatives.

```
table(Actual=test$S, Predict=result)
```

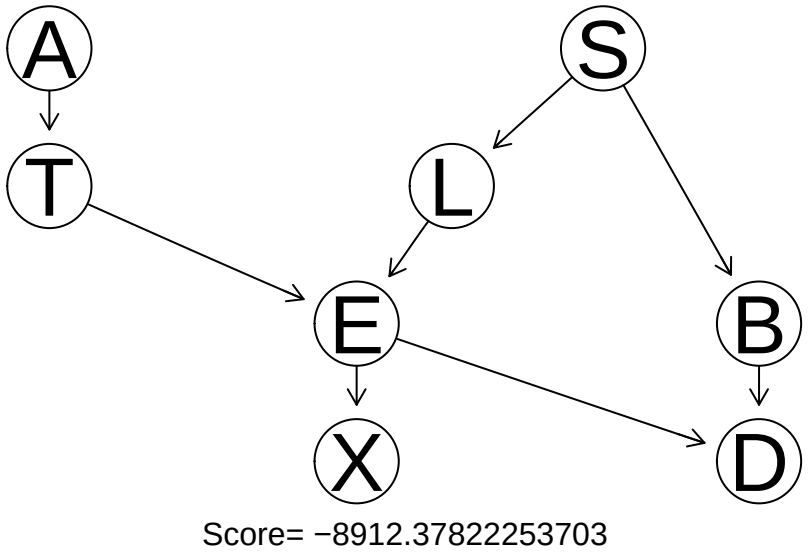
```
##       Predict
## Actual  no yes
##    no  320 157
##    yes 118 405
```

```
cat("ErrorRate:", mean(test$S != result))
```

```
## ErrorRate: 0.275
```

d)Compare your results with those of the true Asia BN.

```
true.dag = bnlearn::model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
graphviz.plot(true.dag,
              sub =paste("Score=",
                         c(as.character(bnlearn::score(true.dag, train))))
             )
```



Let's train the parameters and export the bn object to gRain:

```
true_fit = bn.fit(true.dag, train, method='mle')
true_grain = as.grain(true_fit)
true_grain_comp = compile(true_grain, propagate = TRUE)
```

Now we have to set the evidences for each case and request the probability at node S:

```
true_result = vector(length = n)
for (i in 1:n) {
  dag_find_true = setEvidence(true_grain_comp,
                          nodes = found_nodes,
                          states = c(as.character(test$A[i]),
                                     as.character(test$T[i]),
                                     as.character(test$L[i]),
                                     as.character(test$B[i]),
                                     as.character(test$E[i]),
                                     as.character(test$X[i]),
                                     as.character(test$D[i])))
  s_quary = querygrain(dag_find_true, nodes="S")
  true_result[i] = ifelse(s_quary$S["no"] > s_quary$S["yes"], "no", "yes")

}
```

Confusion matrix and error rate:

```r
table(Actual=test$S, Predict=true_result)
```

```
##       Predict
## Actual  no yes
##    no  320 157
##    yes 118 405
```

```r
cat("ErrorRate:", mean(test$S != true_result))
```

```
## ErrorRate: 0.275
```

The results for both network seem to be the same.

(3) In the previous exercise, you classified the variable S given observations for all the rest of the variables. Now, you are asked to classify S given observations only for the so-called Markov blanket of S, i.e. its parents plus its children plus the parents of its children minus S itself. Report again the confusion matrix.

First obtaining the markov blanket of the node S:

```r
m_blanket = mb(graph_fit, "S")
m_blanket
```

```
## [1] "L" "B"
```

Doing the inference and returning the confusion matrix:

```r
mb_result = vector(length = n)
for (i in 1:n) {
  dag_find_mb = setEvidence(fitted_grain_comp,
                            nodes = m_blanket,
                            states = c(as.character(test$L[i]),
                                       as.character(test$B[i])))
  s_quary = querygrain(dag_find_mb, nodes="S")
  mb_result[i] = ifelse(s_quary$S["no"] > s_quary$S["yes"], "no", "yes")

}


table(Actual=test$S, Predict=mb_result)
```

```
##       Predict
## Actual  no yes
##    no  320 157
##    yes 118 405
```

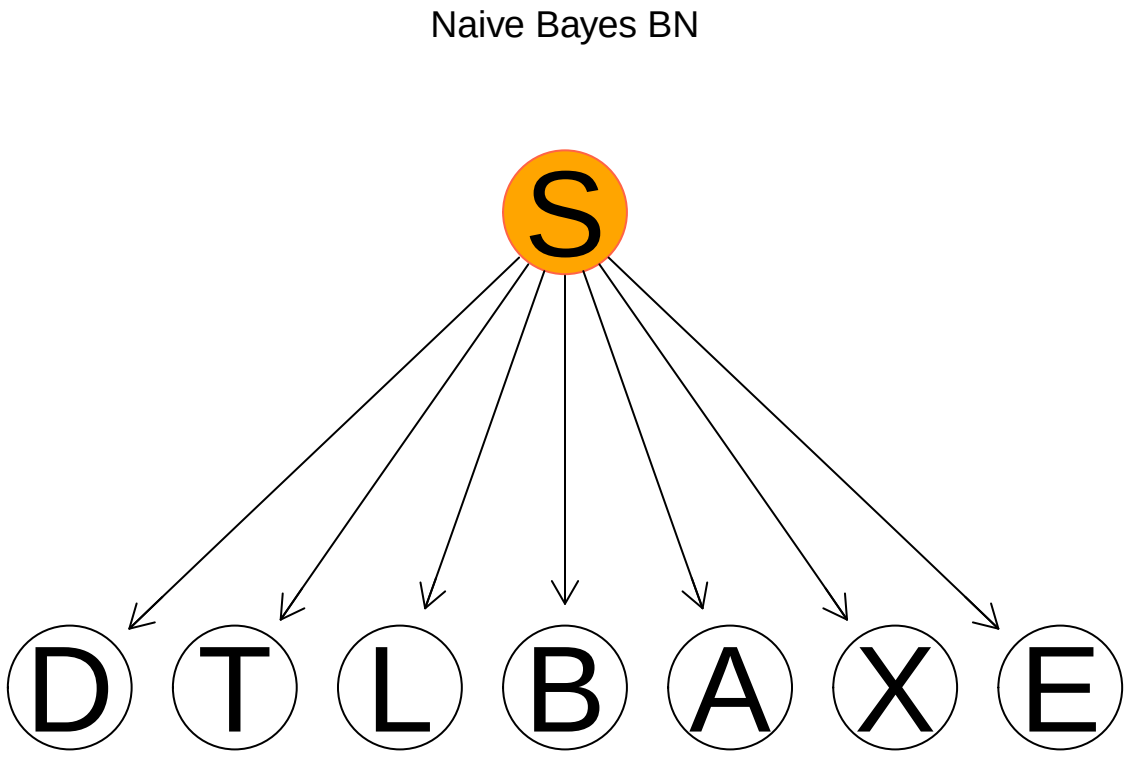
```r
cat("ErrorRate:", mean(test$S != mb_result))
```

```
## ErrorRate: 0.275
```

The result is as same as the previous results.

(4) Repeat the exercise (2) using a naive Bayes classifier, i.e. the predictive variables are independent given the class variable. Model the naive Bayes classifier as a BN. You have to create the BN by hand.

Naive Bayes Assumption : In Naive Bayes, we use Bayes Theorem with strong Naive assumption we consider independence between features. Since here we have features namely "D" , "T", "L", "B", "A", "X", "E" while target is "S", we will assume all features are independent of each other.




```
## Confusion Matrix for Naive Bayes BN for S

##      prediction
##          no   yes
##   no  0.344 0.133
##   yes 0.186 0.337

## Misclassification rate for node S Naive Bayes BN 0.319
```

(5) Explain why you obtain the same or different results in the exercises (2-4).

The following table shows the error rate for the 3 different structures:

```
#repeating code block for report
naive_bayes = model2network("[S][A|S][T|S][L|S][B|S][E|S][X|S][D|S]")
nb_fit = bn.fit(naive_bayes, train, method='mle')
```

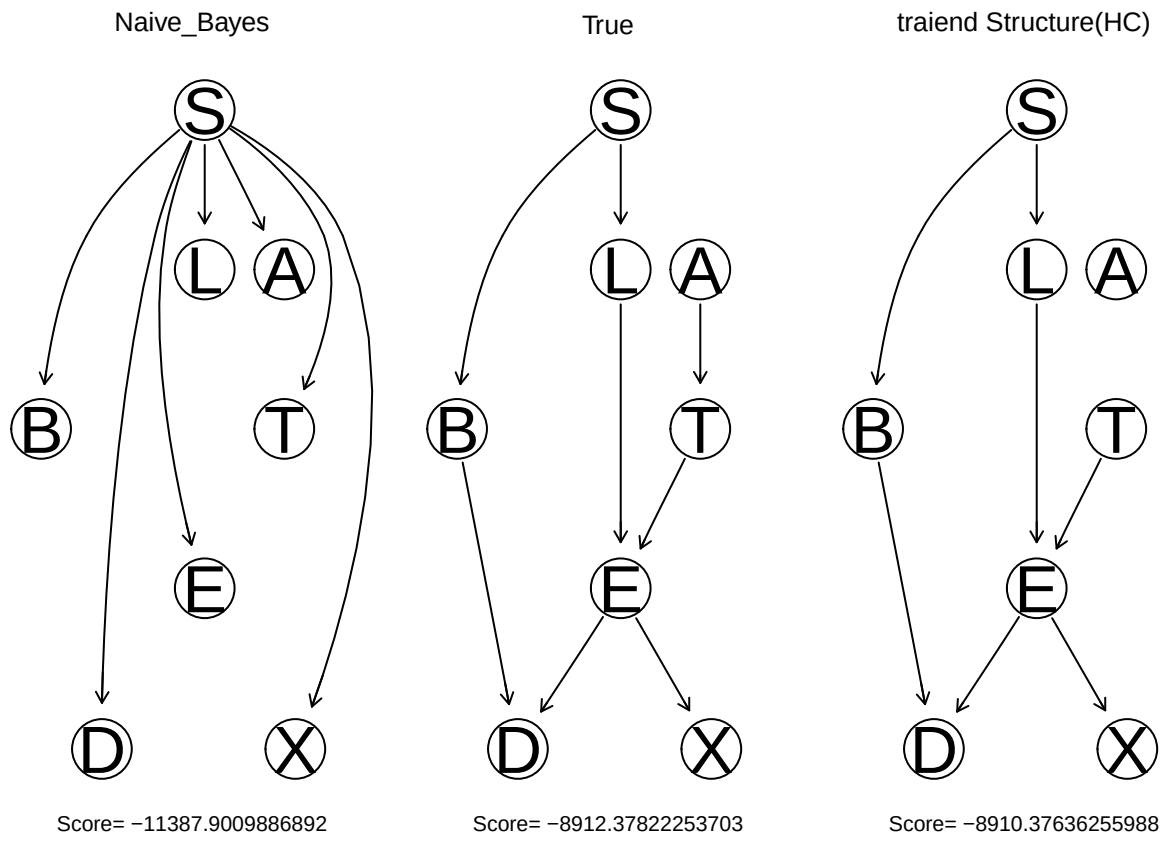
```
nb_grain = as.grain(nb_fit)
nb_grain_comp = compile(nb_grain, propagate = TRUE)
nb_result = vector(length = n)
for (i in 1:n) {
  dag_find_nb = setEvidence(nb_grain_comp,
                            nodes = found_nodes,
                            states = c(as.character(test$A[i]),
                                       as.character(test$T[i]),
                                       as.character(test$L[i]),
                                       as.character(test$B[i]),
                                       as.character(test$E[i]),
                                       as.character(test$X[i]),
                                       as.character(test$D[i])))
  s_quary = querygrain(dag_find_true, nodes="S")
  nb_result[i] = ifelse(s_quary$S["no"] > s_quary$S["yes"], "no", "yes")

}
#repeating code block for report


compare_result = data.frame("learned_structure"=mean(test$S != result),
                            "True_structure"=mean(test$S != true_result),
                            "Naive_Bayes"=mean(test$S != nb_result),
                            "Markov blanket"= mean(test$S != mb_result))
row.names(compare_result) = "Error Rate"
knitr::kable(compare_result)
```

	learned_structure	True_structure	Naive_Bayes	Markov.blanket
Error Rate	0.275	0.275	0.523	0.275

```
par(mfrow=c(1,3))
graphviz.compare(NaiveBayesBN, true.dag, train_DAG_hc,
                 diff = "none",
                 main = c('Naive_Bayes', 'True', 'traiend Structure(HC)'),
                 sub = paste("Score=",c(as.character(bnlearn::score(naive_bayes,train)),
                                        as.character(bnlearn::score(true.dag, train)),
                                        as.character(bnlearn::score(train_DAG_hc, train))
                                        )))
```



Naive-bayes assumes that all the predictors are conditionally independent given S which is far from the true independencies. As a result the marginal probability distribution for a S is different from that is obtained by the true network., and thus a different probabilistic inference will be obtained. The trained graph, however, is similar to the true one. Their log-likelihood score on the train data are so close to each other. In other words, in term of returning the data independence they are equivalent, and therefore the resulted marginal distribution of S in both graph would be similar. The result for Markov blanket is also the same as that of the true and learned graphs. The reason is that in both true and fitted structures S is a common parent of L and B, thus given L and B all the paths would be blocked and S is independent of the others. So knowing about L and B is enough to infer about S and knowledge about the other variables add no more information about S. In other words, S is d-separated from other variables by its Markov blanket(B,L).

Appendix

```
knitr::opts_chunk$set(echo = TRUE)
rm(list = ls())
library(bnlearn)
library(gRain)
library(Rgraphviz)
data("asia")
head(asia)

dag.True  = bnlearn::model2network(string = "[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")

bnlearn::graphviz.plot(dag.True, main = "Asia Data Bayes Network Graph"
                       #,  layout = "neato",
                       ,highlight = list(nodes = "E" ,  col = "tomato",
                                         fill = "orange") )
```

```r
set.seed(123)
init_graph <- random.graph(nodes = colnames(asia))

# First Experiment
model1 <- hc(x = asia, start = NULL, restart = 1, score = "bic")
model2 <- hc(x = asia, start = NULL, restart = 10, score = "bic")
model3 <- hc(x = asia, start = init_graph, restart = 1, score = "bic")
model4 <- hc(x = asia, start = init_graph, restart = 10, score = "bic")
model5 <- hc(x = asia, start = NULL, restart = 1, score = "bde", iss = 5)
model6 <- hc(x = asia, start = init_graph, restart = 10, score = "bde", iss = 5)


par(mfcol = c(2,3))
graphviz.compare(model1, model2, model3, model4, model5, model6,
                 sub = c('#1', '#2', '#3', '#4', '#5', '#6'))

cat(paste("Are model 1 and 2 equivalent: ", all.equal(model1, model2)))
cat(paste("Are model 3 and 4 equivalent: ", all.equal(model3, model4)))


# Second Experiment
model7 <- hc(x = asia, start = NULL, restart = 100, score = "bde", iss = 1)
model8 <- hc(x = asia, start = NULL, restart = 100, score = "bde", iss = 10)
model9 <- hc(x = asia, start = NULL, restart = 100, score = "bde", iss = 100)
model10 <- hc(x = asia, start = init_graph, restart = 100, score = "bde", iss = 1)
model11 <- hc(x = asia, start = init_graph, restart = 100, score = "bde", iss = 10)
model12 <- hc(x = asia, start = init_graph, restart = 100, score = "bde", iss = 100)

par(mfcol = c(2,3))
graphviz.compare(model7, model8, model9, model10, model11, model12,
                 sub = c('#7', '#8', '#9', '#10', '#11', '#12'))

cat(paste("Are model 7 and 9 equivalent: ", all.equal(model7, model9)))
cat(paste("Are model 9 and 12 equivalent: ", all.equal(model9, model12)))

#Splitting the data set into train and test
n = dim(asia)[1]
id = sample(1:n,floor(.8*n))
train = asia[id,]
test = asia[-id,]
#Implementing Score-based algorithm
train_DAG_hc = hc(train)
train_DAG_hc
graphviz.plot(train_DAG_hc)
graph_fit = bn.fit(train_DAG_hc, train)
graph_fit
fitted_grain = as.grain(graph_fit)
fitted_grain
fitted_grain_comp = compile(fitted_grain, propagate = TRUE)
summary(fitted_grain_comp)
n = nrow(test)
found_nodes = c("A", "T", "L", "B", "E", "X", "D")
result = vector(length = n)
```

```
for (i in 1:n) {
  dag_find = setEvidence(fitted_grain_comp,
                         nodes = found_nodes,
                         states = c(as.character(test$A[i]),
                                    as.character(test$T[i]),
                                    as.character(test$L[i]),
                                    as.character(test$B[i]),
                                    as.character(test$E[i]),
                                    as.character(test$X[i]),
                                    as.character(test$D[i])))
  s_quary = querygrain(dag_find, nodes="S")
  result[i] = ifelse(s_quary$S["no"] > s_quary$S["yes"], "no", "yes")

}

table(Actual=test$S, Predict=result)
cat("ErrorRate:", mean(test$S != result))
true.dag = bnlearn::model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
graphviz.plot(true.dag,
              sub =paste("Score=",
                         c(as.character(bnlearn::score(true.dag, train))))
             )
true_fit = bn.fit(true.dag, train, method='mle')
true_grain = as.grain(true_fit)
true_grain_comp = compile(true_grain, propagate = TRUE)
true_result = vector(length = n)
for (i in 1:n) {
  dag_find_true = setEvidence(true_grain_comp,
                         nodes = found_nodes,
                         states = c(as.character(test$A[i]),
                                    as.character(test$T[i]),
                                    as.character(test$L[i]),
                                    as.character(test$B[i]),
                                    as.character(test$E[i]),
                                    as.character(test$X[i]),
                                    as.character(test$D[i])))
  s_quary = querygrain(dag_find_true, nodes="S")
  true_result[i] = ifelse(s_quary$S["no"] > s_quary$S["yes"], "no", "yes")

}

table(Actual=test$S, Predict=true_result)
cat("ErrorRate:", mean(test$S != true_result))
m_blanket = mb(graph_fit, "S")
m_blanket
mb_result = vector(length = n)
for (i in 1:n) {
  dag_find_mb = setEvidence(fitted_grain_comp,
                         nodes = m_blanket,
                         states = c(as.character(test$L[i]),
                                    as.character(test$B[i])))
  s_quary = querygrain(dag_find_mb, nodes="S")
  mb_result[i] = ifelse(s_quary$S["no"] > s_quary$S["yes"], "no", "yes")
```

```
}


table(Actual=test$S, Predict=mb_result)
cat("ErrorRate:", mean(test$S != mb_result))
#Common function written as part of individual report

fnPredict = function(trainData , testData , DAGraph.bnObject , predNode = "S" ,
                     Method = "bayes" , markovB = FALSE , markovObj = NULL ,
                     returnDataFrame = FALSE){

  #fit bn model to training Data
  modelFit = bnlearn::bn.fit(x = DAGraph.bnObject , data = trainData ,
                             method = Method)

  # The main data structure in gRain is the grain class, which stores a fitted Bayesian network as a li
  # conditional probability tables (much like bnlearn's bn.fit objects) and makes it possible for setEv
  # and querygrain() to perform posterior inference via belief propagation.  #via docs

  # #converting to grain object and compiling
  compiled_Grain = gRbase::compile(object = bnlearn::as.grain(modelFit))
  #compiled_Grain = bnlearn::as.grain(modelFit)


  #fetch index of predNode in order to remove that while making predictions


  if(markovB == FALSE)
    {
      indx = which(colnames(testData) == predNode)
      nodesName = colnames(testData[-indx])
    }else{
      nodesName = as.vector(markovObj)
    }

  prediction = numeric(nrow(testData))

  #Added to see changing probabilities such that we can validate confusion matrix
  if(returnDataFrame == TRUE)
    dfPredTest = data.frame(matrix(data = NA , ncol = 2 , nrow = nrow(testData)))


  for(i in 1:nrow(testData))
  {

    #In setEvidence function
    #nodes : A vector of nodes; those nodes for which the (conditional) distribution is requested.
    #states : A vector of states (of the nodes given by 'nodes')

    #fetching ith row of data set to predicted without target
    #extending logic for markov blanket case model

    if(markovB == FALSE){
```

```
      testDataRow.State = as.vector(unname(unlist(testData[i, -indx])))

    }else{
      testDataRow.State = as.vector(unname(unlist(testData[i, nodesName])))
    }

    # if(i == 1)
    #   print(testDataRow.State)

    Evidence_Grain = gRain::setEvidence(object = compiled_Grain ,
                                        nodes = nodesName ,
                                        states = testDataRow.State
                                        )

    queryGrain = querygrain(object = Evidence_Grain,
                            #object = compiled_Grain , evidence = Evidence_Grain ,
                             nodes = predNode ,
                             type = "marginal"
                            )

    qG = as.vector(unname(unlist(queryGrain)))
    #qG[1] = No and qG[2] = Yes
    #Added for visual validation of result @removed from final result
    if(returnDataFrame == TRUE)
    {
      dfPredTest[i,] = qG
    }

    prediction[i] = ifelse(qG[2] > qG[1]   ,  "yes" , "no")

  }#for

  #confusion matrix
  confusionMatrix = prop.table(table(test$S , prediction))

  if(returnDataFrame == TRUE)
  {
    return(list("Cmat" = confusionMatrix , "df" = dfPredTest))
  }else {
          return(confusionMatrix)
        }

}#fnPredict




NaiveBayesBN = bnlearn::empty.graph(nodes = c("D" , "T", "L", "B", "A",  "X", "E" , "S"))
NaiveBayesBN = bnlearn::set.arc(NaiveBayesBN, from = "S" , to = "D" )
NaiveBayesBN = bnlearn::set.arc(NaiveBayesBN, from = "S" , to = "T" )
NaiveBayesBN = bnlearn::set.arc(NaiveBayesBN, from = "S" , to = "L" )
NaiveBayesBN = bnlearn::set.arc(NaiveBayesBN, from = "S" , to = "B" )
NaiveBayesBN = bnlearn::set.arc(NaiveBayesBN, from = "S" , to = "A" )
```

```
NaiveBayesBN = bnlearn::set.arc(NaiveBayesBN, from = "S" , to = "X" )
NaiveBayesBN = bnlearn::set.arc(NaiveBayesBN, from = "S" , to = "E" )


bnlearn::graphviz.plot(NaiveBayesBN, main = "Naive Bayes BN"
                       ,highlight = list(nodes = "S" ,  col = "tomato",
                                         fill = "orange") )


#result might appear different from individual report as data split is changed here

NaiveBayesBNPred = fnPredict(trainData = train , testData = test , DAGraph.bnObject = NaiveBayesBN
                 , predNode = "S" , Method = "mle" , markovB = FALSE , returnDataFrame = FALSE )

cat("Confusion Matrix for Naive Bayes BN for S ")
cat("\n")
NaiveBayesBNPred
cat("\n")
MiscalNaiveB = 1 - sum(diag(NaiveBayesBNPred))/sum(NaiveBayesBNPred)
cat("Misclassification rate for node S Naive Bayes BN" , MiscalNaiveB)
cat("\n")

#repeating code block for report
naive_bayes = model2network("[S][A|S][T|S][L|S][B|S][E|S][X|S][D|S]")
nb_fit = bn.fit(naive_bayes, train, method='mle')
nb_grain = as.grain(nb_fit)
nb_grain_comp = compile(nb_grain, propagate = TRUE)
nb_result = vector(length = n)
for (i in 1:n) {
  dag_find_nb = setEvidence(nb_grain_comp,
                        nodes = found_nodes,
                        states = c(as.character(test$A[i]),
                                   as.character(test$T[i]),
                                   as.character(test$L[i]),
                                   as.character(test$B[i]),
                                   as.character(test$E[i]),
                                   as.character(test$X[i]),
                                   as.character(test$D[i])))
  s_quary = querygrain(dag_find_true, nodes="S")
  nb_result[i] = ifelse(s_quary$S["no"] > s_quary$S["yes"], "no", "yes")

}
#repeating code block for report


compare_result = data.frame("learned_structure"=mean(test$S != result),
                            "True_structure"=mean(test$S != true_result),
                            "Naive_Bayes"=mean(test$S != nb_result),
                            "Markov blanket"= mean(test$S != mb_result))
row.names(compare_result) = "Error Rate"
knitr::kable(compare_result)
par(mfrow=c(1,3))
graphviz.compare(NaiveBayesBN, true.dag, train_DAG_hc,
```

```
                  diff = "none",
                  main = c('Naive_Bayes', 'True', 'traiend Structure(HC)'),
                  sub = paste("Score=",c(as.character(bnlearn::score(naive_bayes,train)),
                                         as.character(bnlearn::score(true.dag, train)),
                                         as.character(bnlearn::score(train_DAG_hc, train))
                                         )))
```