Aruup818@student.liu.se

Lab1

Arun Uppugunduri

2020-09-13

In this assignment we are working with classifying the asia dataset. The task si to classify the variable S smoker given the other. The classification is to be solved by fitting the data to a bayesian network. We are trying to classify the variable S = Smoker [yes/no] by looking at the observations of the other nodes.

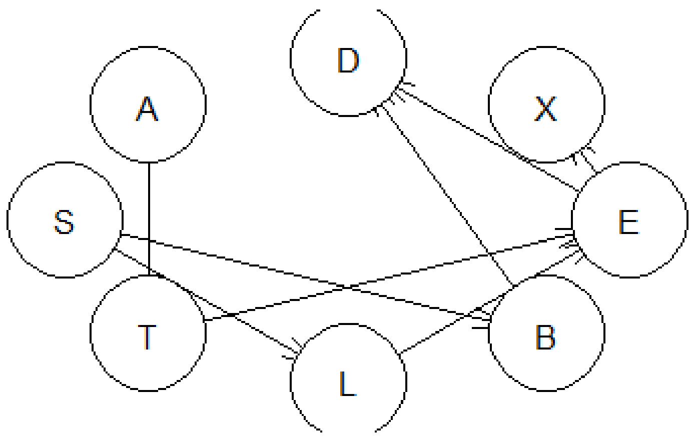
```
## Warning: package 'bnlearn' was built under R version 3.6.3

##
## Attaching package: 'bnlearn'

## The following object is masked from 'package:stats':
##
##     sigma

##    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
```

Below can be seen the true structure of the network. The first part of this lab lies in applying the hill-climbing algorithm in order to learn this network structure

```
variables = colnames(asia)
true_network = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]", orde
ring = variables)
plot(true_network, main = "True network")
```

Aruup818@student.liu.se

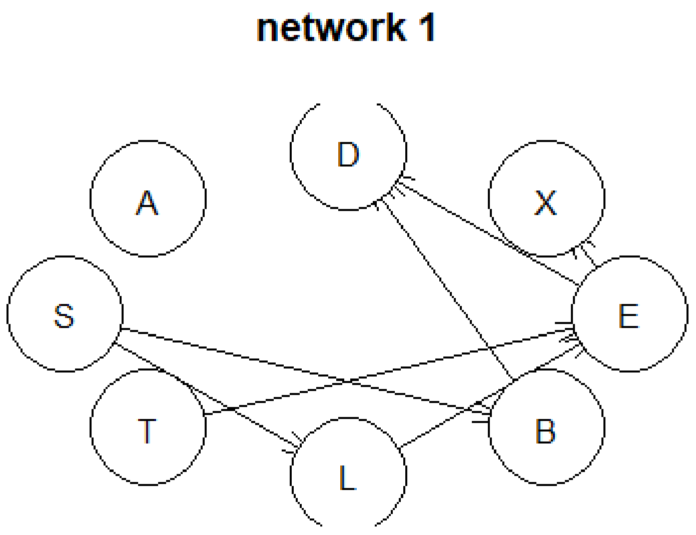
True network



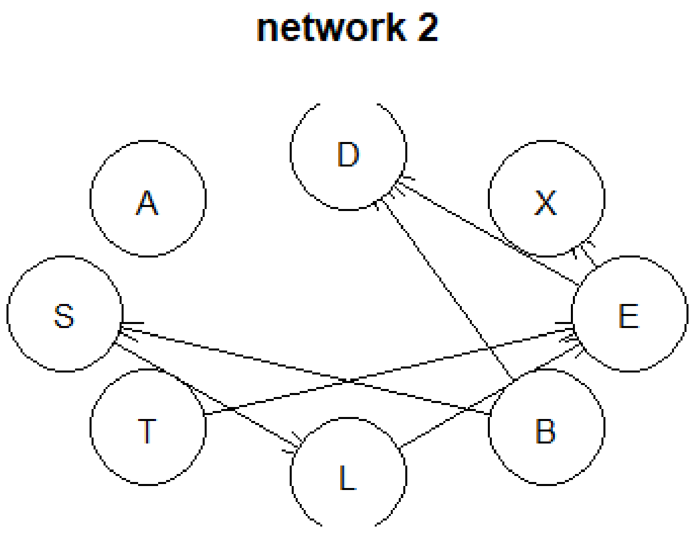
Task 1 Show that multiple runs of the hill-climbing algorithms can return non-equivalent Bayesian network (BN) structures

The hill-climbing algorithm is a iterative algorithm which adds/eliminates edges between nodes in order to find the optimal tree. The optimal tree chosen is that which has the highest posterior porbability. Using the hill climbing algorithm there is a risk of getting stuck in a local optimum. Depending on extra input like initial structure, score to be used and amount of restarts we can thereby end up with non-equivalent bayesian network structure. This means that that there is no unambiguity in the model.

```r
#Network 1
network1 = hc(asia, restart = 1, score = "bic")
plot(network1, main = "network 1")
```



```
cpdag(network1)

##
##   Bayesian network learned via Score-based methods
##
##   model:
##     [partially directed graph]
##   nodes:                                 8
##   arcs:                                  7
##     undirected arcs:                     2
##     directed arcs:                       5
##   average markov blanket size:           2.25
##   average neighbourhood size:            1.75
##   average branching factor:              0.62
##
##   learning algorithm:                    Hill-Climbing
##   score:                                 BIC (disc.)
##   penalization coefficient:              4.258597
##   tests used in the learning procedure:  91
##   optimized:                             TRUE

#Network 2
network2 = hc(asia, restart = 100, score = "bic")
plot(network2, main = "network 2")
```

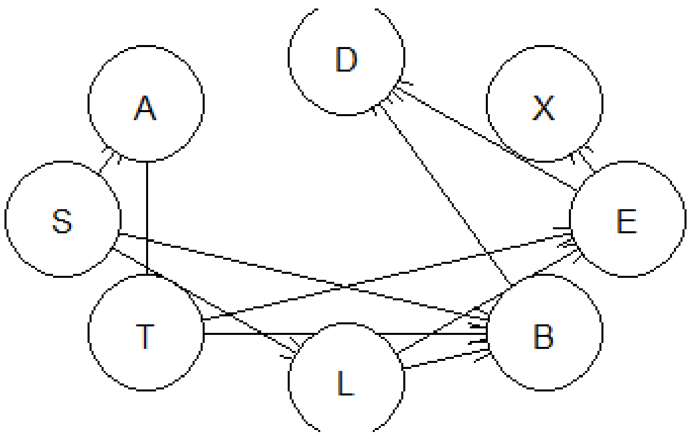


```
cpdag(network2)

##
##   Bayesian network learned via Score-based methods
##
##   model:
##     [partially directed graph]
##   nodes:                                 8
##   arcs:                                  7
##     undirected arcs:                     2
##     directed arcs:                       5
##   average markov blanket size:           2.25
##   average neighbourhood size:            1.75
##   average branching factor:              0.62
##
##   learning algorithm:                    Hill-Climbing
##   score:                                 BIC (disc.)
##   penalization coefficient:              4.258597
##   tests used in the learning procedure:  1808
##   optimized:                             TRUE

## Score for network 1, restart = 1 -11107.29

## Score for network 2, restart = 100 -11107.29

## [1] "Different arc sets"
```

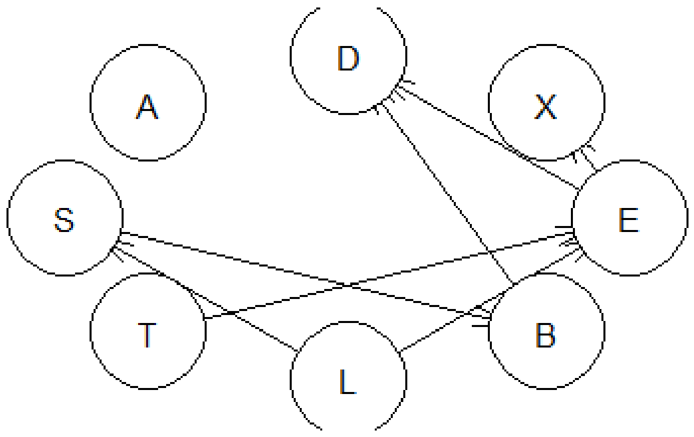
Aruup818@student.liu.se

By changing the value for restart we modify how many random restarts of the algorithm should be used. In this example we can howver conclude that it does NOT result in non-equivalent network strutures. We know this by comparing the structure. Since only the direction of an edge is what differs this does not change any form of independence/dependence. This can also be seen by comparing the scores. Since the scooring criteria assign the same score to equivalent structures.

```
#Network 3
network3 = hc(asia, score = "aic", restart = 100)
plot(network3)
```



```
cpdag(network3)

##
##   Bayesian network learned via Score-based methods
##
##   model:
##     [partially directed graph]
##   nodes:                                 8
##   arcs:                                  11
##     undirected arcs:                     3
##     directed arcs:                       8
##   average markov blanket size:           3.50
##   average neighbourhood size:            2.75
##   average branching factor:              1.00
##
##   learning algorithm:                    Hill-Climbing
##   score:                                 AIC (disc.)
##   penalization coefficient:              1
##   tests used in the learning procedure:  1877
##   optimized:                             TRUE

#Network 4
network4 = hc(asia, score = "bic", restart = 100)
plot(network4)
```



```
cpdag(network4)

##
##   Bayesian network learned via Score-based methods
##
##   model:
##     [partially directed graph]
##   nodes:                                 8
##   arcs:                                  7
##     undirected arcs:                     2
##     directed arcs:                       5
##   average markov blanket size:           2.25
##   average neighbourhood size:            1.75
##   average branching factor:              0.62
##
##   learning algorithm:                    Hill-Climbing
##   score:                                 BIC (disc.)
##   penalization coefficient:              4.258597
##   tests used in the learning procedure:  1829
##   optimized:                             TRUE

#Comparing network 3 and 4
all.equal(network3, network4)

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

## Score for network 3, score = aic, restart = 100 => Score:  -11129.57

## Score for network 4, score = bic restart = 100, => Score: -11107.29
```

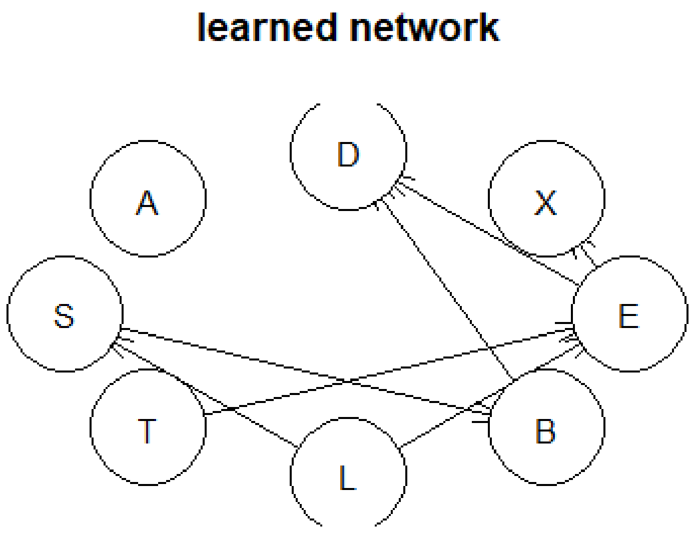
In this step i compared the resutls from using different scores. By looking at the new plots produced we can easily conclude that the two models created are different , in other words non-equivalent. This can also be stated by looking at their scores. The best values to select in this case would thereby be network 4 using the default bic Bayesian Information Criterion score. This is because it has the highest score but we also notice that it's graph show an independence of A. Since the hill climbing algorithm does not produce any false independences then we know that this is something which we can trust and there is no use in modelling extra unesseary parameters.

Task 2 Learn a BN with 80 % of the data. Learn both its structure and its parameters. Use any learning algorithm and setting that you consider appropriate. Use the BN learned to classify the remaining20 % test data into two classes S = [yes/no]. In other words, compute the posterior probabilitydistribution of S for each case and classify it in the most likely class

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

I learn the Bayesian network structure by applying the hill climbing algorithm. I use the settings which in the previous step resulted in the best score.

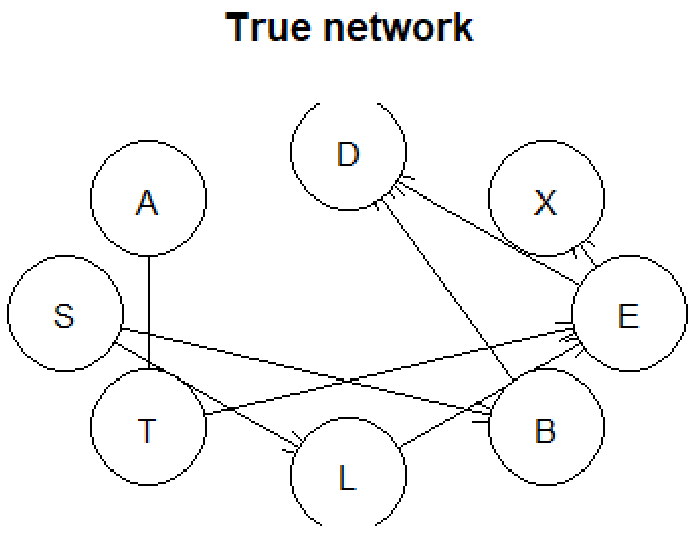
Aruup818@student.liu.se



Then we creat the true network structue using the given structure

```
true_network = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]", orde
ring = variables)
plot(true_network, main = "True network")
```

Aruup818@student.liu.se



Having created the network we then fit the parameters of the bayesian network conditional on its structure. We do this by fitting the train data to the netork structure

```
bn_fit = bn.fit(bn_network,
                data = train)
true_bn_fit = bn.fit(true_network,
                     data = train)
```

In order to make inference we have to convert the fit to a grain and compile it. The function as.grain converts the bn.fit object to a grain. The grain class stores a fitted bayesian network as a list of conditional probability tables. This makes it possible for setEvidence and querygrain to perform posterior inference via belief propagation. The grain does not allow conditional probabilities to be NaN. This happens when estimationg them via a maximum likelihood and some parent ocnfigurations are not observed in the data. Grain will then replace NaN with uniform distribution. In this case the data is however complete and this is not a problem which we have to take into account.

```
#Convert to a grain object
bn_fit_gRain = as.grain(bn_fit)

## Loading required namespace: gRain

#For the true network
bn_fit_true = as.grain(true_bn_fit)

library(gRain)
```

```
## Warning: package 'gRain' was built under R version 3.6.3

## Loading required package: gRbase

## Warning: package 'gRbase' was built under R version 3.6.3

##
## Attaching package: 'gRbase'

## The following objects are masked from 'package:bnlearn':
##
##     ancestors, children, parents

#Compile the BN which means creating teh junction tree and establishing cliqu
e potentials
junction_tree = compile(bn_fit_gRain)
#For the true network
junction_tree_true = compile(bn_fit_true)
```

In order to be able to make predictions on the new test data we had to implement an own prediction function. In this function we make use of the funcitons setEvidence and querygrain. Using setEvidence we extract the conditional probability tables for a node(s) which can be in states X. What the function does is that it takes all the data and assigns it to the observation nodes. THis means that the node we are tring to classify should not be included. Once this is extracted we can then apply querygrain in order to extract the conditional for a node (prediction node) given the evidence extracted above. Using this conditional probability we then assign each observation S = "yes" if the probability for p(S) = "yes" > 0.5

```
predict_test = function(BN_model, testData, observation_nodes, prediction_var
iable){
  prediction = rep(0, length(testData))
 for(i in 1:nrow(testData)){
   node_state = NULL
   for (k in observation_nodes){
     node_state[k] = if(testData[i,k] == "yes") "yes" else "no"
   }

   #Assign the new data points for each observation to a node
   # observation nodes should be all the nodes except the conditional node we
are doing inference on
   # In this case this is S
   evidence = setEvidence(object = BN_model,
                          nodes = observation_nodes,
                          states = node_state)

   #obtain the conditional distribution for a node given the evidence obtaine
d above. In this case we get the conditional
   #distribution for S which is the prediction variable
   conditional_distribution = querygrain(object = evidence,
                                         nodes = prediction_variable)$S
   #For each observation (row in the data set) we classify the prediction var
iable as yes/no depending on what
   #probability is larger

   prediction[i] = if(conditional_distribution["yes"] > 0.5) "yes" else "no"
 }
  return(prediction)
}


test_prediction = predict_test(BN_model = junction_tree,
             testData = test,
             observation_nodes = c("A", "T", "L", "B", "E", "X", "D"),
             prediction_variable = c("S"))

table(test_prediction, test$S)

##
## test_prediction  no yes
##             no  337 121
##             yes 176 366

test_prediction_true = predict_test(BN_model = junction_tree_true,
                              testData = test,
                              observation_nodes = c("A", "T", "L", "B", "E",
"X", "D"),
                              prediction_variable = c("S"))

table(test_prediction_true, test$S)

##
## test_prediction_true  no yes
##                  no  337 121
##                  yes 176 366
```

We notice that the prediction using the true structure vs the learned structure results in the same confusion matrix.

Task 3 The same task as before but that the classification of S should now be made given observations only for the so called Maakov bnlanket of S, i.e its parents plus its children plus the parents of its children minus S itself

Extract Marakov blaket from the fitted network structure generated by the hill climb algorithm as well as the fitted network structure created from the true model

```
mb_hc = mb(x = bn_fit,
           node = c("S"))

mb_true = mb(x = true_bn_fit,
             node = c("S"))
#Learned network
cat(mb_hc)

## L B

#True network
cat(mb_true)

## L B
```

The resulting marakov blanket for the two networks is as we can see actually the same. This can also be viewed and verified through the graph where the nodes L and B are the only ones that are connected to prediction variable S. We then make the new predictions but only using the marakov blanket as the obseration nodes. This can be seen as that we are only using certain nodes and certain data to classify S since the other have no relation to S.

```
test_prediction_mb = predict_test(BN_model = junction_tree,
                               testData = test,
                               observation_nodes = mb_hc,
                               prediction_variable = c("S"))

table(test_prediction_mb, test$S)

##
## test_prediction_mb  no yes
##                no  337 121
##                yes 176 366

mean(test_prediction_mb != test$S)

## [1] 0.297

#True network structure
#For the true graph
test_prediction_true_mb = predict_test(BN_model = junction_tree_true,
                                    testData = test,
                                    observation_nodes = mb_true,
                                    prediction_variable = c("S"))

table(test_prediction_true, test$S)
mean(test_prediction_true_mb != test$S)
```

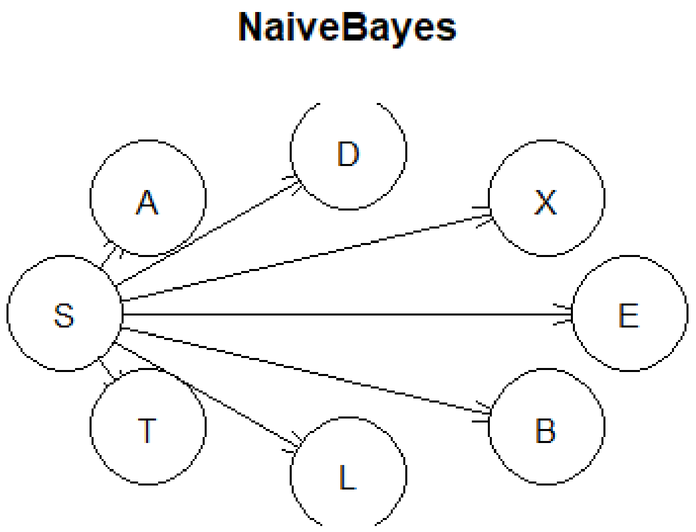
Using the markov blanket we only make computations using the data belonging to the nodes B and L since they are the parents/children. We see that the resulting matrix is the same and this makes sense. By looking at the graph we can see that the node S is independent from all the nodes given B, L. Adding thhe other nodes would therfore only create more computation but would not gain any more information.

Task 4 Repeat the previous excercise using a naiveBayes classifier, i.e the predictive variables are independent given the class variable

Aruup818@student.liu.se

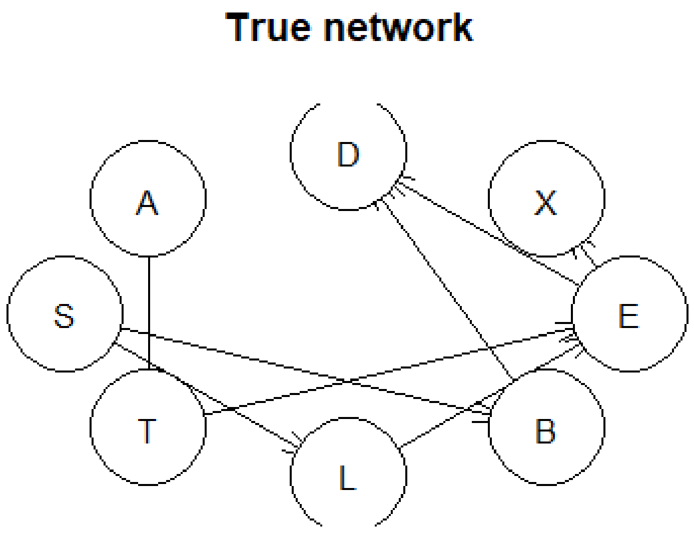
Create an bayesian network having structure as if all its predictive variables are independent given the class variable.

```
BN_bayes =  model2network("[S][A|S][T|S][L|S][B|S][E|S][X|S][D|S]", ordering
= variables)
plot(BN_bayes, main = "NaiveBayes")
```



```
plot(true_network, main = "True network")
```

Aruup818@student.liu.se



Having created the network we do the same precedures as before of fitting the parameters, converting to grain and compiling the junction tree. Finally we make the prediction using the new naivebayes model and compare it to the true model.

```
#Naive bayes classifier
table(Observartion = test$S, Classification = prediction_bayes)
mean(test$S != prediction_bayes)

#True network structure
table(Observartion = test$S, Classification = prediction_true)
mean(test$S != prediction_true)
```

Task 5 Explain why you obtain the same or different results in the exercise 2-4.

In task 2 where we compared the learned structure using the default score "bic" with restart = 100 we ended up with confusion matrices which where identical. This is due to the fact that S only have the nodes B and L as its marakov blanket. The only difference which could be seen when comparing the two plots was for edges between nodes which where not connected to S. Furthermore we saw another example of this is task 3 when only the marakov blanket was used as the observed variables. Since here, it produced the exakt same confusion matrices yet again. This would suggest that this is a result which is reached either way if we specify the marakov blanket from before or not.

Comparing the naiveBayes network with the true network we can se quite large differences in the structure. When comparing the confusion matrice produced and its missclassification rate we can observe that the naivebayes classifier does a worse job that our original model. This could be explained by the fact that the assumtion of independence between prediction variables does not hold. Looking at the plot for the real network we can see that there are variables which would not be independent gives S.

Comparing

Code

```
#Install the necessary
if (!requireNamespace("BiocManager", quietly = TRUE))
install.packages("BiocManager")
install.packages("bnlearn")
BiocManager::install("RBGL")
BiocManager::install("Rgraphviz")
BiocManager::install("gRain")
library(bnlearn)
library(gRain)

### TASK 1 ###
#Read the data to be used
data(asia)
cat("Data structure: ", colnames(asia))
# Variables = A S T L B E X D
variables = colnames(asia)
set.seed(12345)
#Task 1: Run the hill climbing algorithm on the data. This will return a Baye
sian
#network structure of class bn. By modifying the parameters score and restart
# we will examine how the bayesian network is affected.

###General network
network = hc(asia)
#Ways of examining the network
plot(network)
arcs(network)
vstructs(network)
cpdag(network)

#Network 1
network1 = hc(asia, restart = 1, score = "bic")
plot(network1)
cpdag(network1)
cat("Score for network 1, restart = 1, score = bci")
score(network1, asia)

#Network 2
network2 = hc(asia, restart = 100, score = "bic")
plot(network2)
cpdag(network2)
score(network2, asia)
```

```
print(all.equal(network1, network2))
# By changing the number of restarts we in general avoid getting stuck into a
# local optimum. By increasing the amount of restart the amount of tests is
# greatly increased from 91 to 1832. Since the score is however the same
# then the two classes are score equivalent. This means that the scoring crit
eria assign the same score
# to equivalent structures. In this example, meerly changing the restart does
therfore not create non
# equivalent BN in this case

#Network 3
network3 = hc(asia, score = "aic", restart = 100)
plot(network3)
cpdag(network3)
cat("BN loglike, restart = 100 => Score: ",
score(network3, asia))

#Network 4
network4 = hc(asia, score = "bic", restart = 100)
plot(network4)
score(network4, asia)
cpdag(network4)

#Comparing network 3 and 4
all.equal(network3, network4)

#Both in the plot and through the above all.equal we see that there are
# 2 different models which have been created where they have a different
#number of directed and undirected arcs. We can moreover verify that
#There are non-equivalent classes created since their scores differ.
# looking at the score we we prefer the bic since this gives a less negative
score
# looking at the graph we can also see that A is independent in the bic netwo
rk and since the
# hill climbing algorithm does not produce fake independences we know that we
can trust this and is therfore prefereed
# since is gives us less parameters to estimate




### Task 2 ###
# Divide data into train/test and use it to learn the structure and parameter
s
# of a network. Use the created BN to classify the remaining test dataset int
o two classes
# S = yes and S = no. Using a scoring algorithm this results in computing the
posterior probability dsitribution for S
# for each class and classify the most likely class.
```

Aruup818@student.liu.se

```
### Functions to be used
#bn.fit
#as.grain

###Functions for approximate inference
#prop.table
#table
#cpdist

### gRain package
# compile
# setFinding
# querygrain

set.seed(12345)
n = nrow(asia)
id = sample(n, floor(n*0.8))
train = asia[id,]
test = asia[-id,]

#Learn BN structure using parameters of your choice
set.seed(12345)
bn_network = hc(train, score = "bic", restart = 100)
plot(bn_network, main = "learned network")

#Create the true BN for the Asia dataset
true_network = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]", orde
ring = variables)
plot(true_network, main = "True network")

#Fit the parameters of a bayesian notwork conditional on its structure
bn_fit = bn.fit(bn_network,
                data = train)
true_bn_fit = bn.fit(true_network,
                     data = train)

library(gRain)
# In order to make inference we have to convert the fit to a BN and compile i
t
#As.grain conversts the bn.fit object to a grain. The grain class stores a fi
tted bayesian network as a list of conditional
# probability tables. This makes it possible for setEvidence and querygrain t
o perform posterior inference via belief
# propagation. The grain does not allow conditional probabilities to be NaN.
This happens when estimationg them via a
# maximum likelihood and some parent ocnfigurations are not observed in the d
ata. Grain will then replace NaN with uniform
# distribution
```

```
bn_fit_gRain = as.grain(bn_fit)
bn_fit_gRain
#For the true network
bn_fit_true = as.grain(true_bn_fit)
bn_fit_true


#Compile the BN which means creating teh junction tree and establishing cliqu
e potentials
junction_tree = compile(bn_fit_gRain)
junction_tree
#For the true network
junction_tree_true = compile(bn_fit_true)


   #Using setEvidende we extract the conditional probability tables for node(
s) which can be in states x
   # setEvidence allows to specifivction of hard evidence and likelihood evid
ence for variables.
   # It attache a nodeto all the
   #(bn_fit, nodes = asia[-2], states = "S")

   #query an independence network i.e obtain the conditional distribution of
a set of variables given evidnce on other
   # variables (in most cases)
querygrain(junction_tree)

#prop.table(table(querygrain(junction_tree)))

# Make predictin on the test data

predict_test = function(BN_model, testData, observation_nodes, prediction_var
iable){
  prediction = rep(0, length(testData))
 for(i in 1:nrow(testData)){
   node_state = NULL
   for (k in observation_nodes){
     node_state[k] = if(testData[i,k] == "yes") "yes" else "no"
   }

   #Assign the new data points for each observation to a node
   # observation nodes should be all the nodes except the conditional node we
are doing inference on
   # In this case this is S
   evidence = setEvidence(object = BN_model,
                          nodes = observation_nodes,
                          states = node_state)

   #obtain the conditional distribution for a node given the evidence obtaine
d above. In this case we get the conditional
   #distribution for S which is the prediction variable
   conditional_distribution = querygrain(object = evidence,
                                         nodes = prediction_variable)$S

   #For each observation (row in the data set) we classify the prediction var
iable as yes/no depending on what
   #probability is larger

   prediction[i] = if(conditional_distribution["yes"] > 0.5) "yes" else "no"
 }
  return(prediction)
}


var = c("A", "T", "L", "B", "E", "X", "D")

test_prediction = predict_test(BN_model = junction_tree,
             testData = test,
             observation_nodes = c("A", "T", "L", "B", "E", "X", "D"),
             prediction_variable = c("S"))

table(test_prediction, test$S)

#For the true graph
test_prediction_true = predict_test(BN_model = junction_tree_true,
                              testData = test,
                              observation_nodes = c("A", "T", "L", "B", "E",
"X", "D"),
                              prediction_variable = c("S"))

table(test_prediction_true, test$S)




### Task 3 ###
# The same excercise as before but classify the S only for the Marakov Blanke
t of S. i.e its parents plus its children plus the
# parents of of its children minus S itseld. Reports again the confusion matr
ix

#Useful function
# mb => miscelalaneous utilities: assign or extract various quantities from a
n object of class bn of bn.fit

# Extract Marakov blaket from the fitted network structure generated by the h
ill climb algorithm as well as
# the fitted network structure created from the true model
```

```
mb_hc = mb(x = bn_fit,
           node = c("S"))

mb_true = mb(x = true_bn_fit,
             node = c("S"))

#Make the predictions and create the new confusion matrices
test_prediction_mb = predict_test(BN_model = junction_tree,
                                  testData = test,
                                  observation_nodes = mb_hc,
                                  prediction_variable = c("S"))

table(test_prediction_mb, test$S)
mean(test_prediction_mb != test$S)

#For the true graph
test_prediction_true_mb = predict_test(BN_model = junction_tree_true,
                                       testData = test,
                                       observation_nodes = mb_true,
                                       prediction_variable = c("S"))

table(test_prediction_true, test$S)

#Using the markov blanket we only make computations using the data belonging
to the nodes B and L since they are
# the parents/children. We see that the resulting matrix is the same and this
makes sense. By looking at the graph we can
# see that the node S is independent from all the nodes given B, L. Adding th
he other nodes would therfore only create
# more computation but would not gain any moreinformation.



### TASK 4###
#Repeat the previous excercise but insted using a
#bayes classifier. We are assuming that the predictive
#variables are independent given the class variable
#Model the NB classifier as a BN

# Create the naive bayes network structur under the asumption
# That all predictor variables are independent given
# S.
BN_bayes =  model2network("[S][A|S][T|S][L|S][B|S][E|S][X|S][D|S]")

# Fit the parameters to the network
BN_fit_bayes = bn.fit(BN_bayes, data = train)
plot(BN_bayes)
```

```
#Convert the BN to a gRain object and compile
# to a junction tree.
BN_fit_bayes_grain = as.grain(BN_fit_bayes)
junction_tree_bayes = compile(BN_fit_bayes_grain)

#Classify the test set using the naive baye classifier
prediction_bayes = predict_test(junction_tree_bayes,
                                testData = test,
                                observation_nodes = c("A", "T", "L", "B", "E"
, "X", "D"),
                                prediction_variable = c("S"))

table(Observartion = test$S, Classification = prediction_bayes)
mean(test$S != prediction_bayes)
# Results in a missclassificartio rate of 0.334

prediction_true = predict_test(junction_tree_true,
                                testData = test,
                                observation_nodes = c("A", "T", "L", "B", "E"
, "X", "D"),
                                prediction_variable = c("S"))

table(Observartion = test$S, Classification = prediction_true)
mean(test$S != prediction_true)
# Looking at the true structure we know that the predictor varibles are actua
lly
# not independent given the class variables. Since this asumption is wrong it
will
# therfore not classify as well as the true network structure
```