Lab 1 - Advanced Machine Learning

Phillip Hölscher

14 9 2019

Questions

The purpose of the lab is to put in practice some of the concepts covered in the lectures.

Question 1

Show that multiple runs of the hill-climbing algorithm can return non-equivalent Bayesian network (BN) structures. Explain why this happens. Use the Asia dataset which is in- cluded in the *bnlearn* package. To load the data, run *data("asia")*.

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. You may want to use these options to answer the question. You may also want to use the functions *plot, arcs, vstructs, cpdag* and *all.equal*.

Results Question 1

In this task I will examine the parameters **restart** and **score.** In total I have created 4 Bayesian Network objects, all with different parameter entries. I will compare the output with each other. At the beginning I will set a score and distinguish the parameter restart. I will examine these two Bayesian Network objects with the following functions.

- plot
- all.equal
- score
- print of all object information

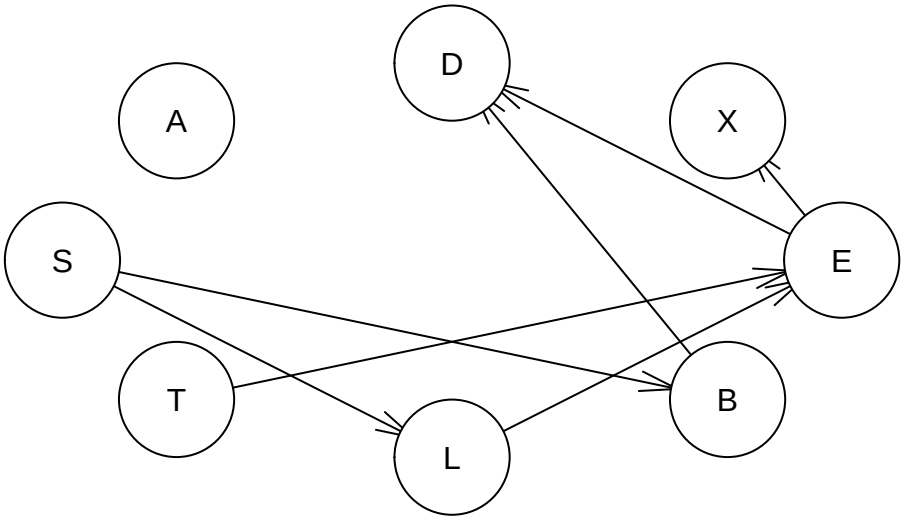
- Parameter: *restart*

Creat Bayesian Network objects

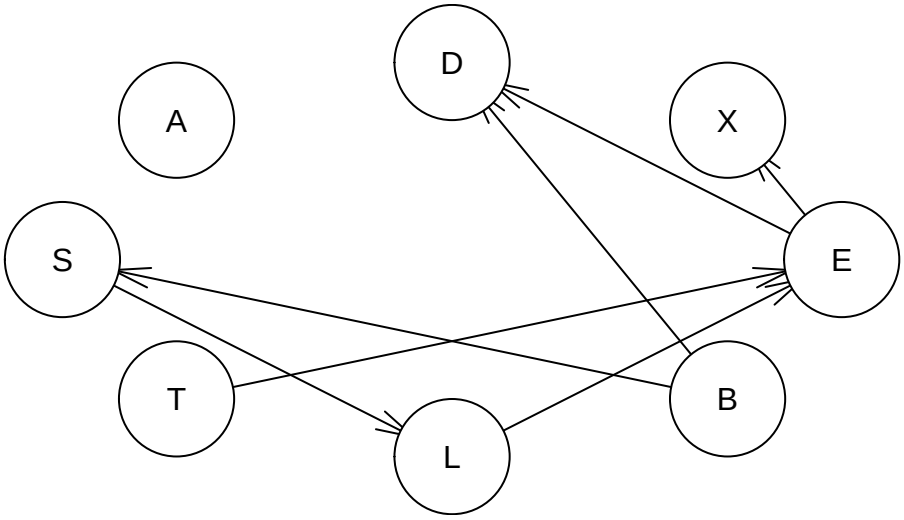
```
# create object 1
hc_o1 = hc(x = data,
           score = "bde",
           restart = 1)

# create object 2
hc_o2 = hc(x = data,
           score = "bde",
           restart = 100)
```

- Plot

```
## BN 1 - score: bde - restart: 100
```




```
## BN 2 - score: bde - restart: 1
```



In this example we can see that the parameter restart can make a difference to model creation. The connection between “S”, “L” and “B” is different.

- all.equal

```
all.equal(hc_o1,hc_o2)
```

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

With these functions we can compare the models with each other and get in the given case an indication about the respective differences. The analysis through the visualization we have already noticed that the arrows have a different connection. This means that the generated model is different. In the later course of the interpretation I will go into this point in more detail.

- Score

```
## BN 1 - score: bde - restart: 1 - score value:
```

```
## [1] -11107.29
```

```
## BN 2 - score: bde - restart: 100 - score value:
```

```
## [1] -11107.29
```

If we compare the scores of the two models, however, we see that they are the same. Otherwise we should have seen this in the **all.equal** function.

- Object information

```
## BN 1 - score: bde - restart: 1
```

```
##
##   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:                                 Bayesian Dirichlet (BDe)
##   graph prior:                           Uniform
##   imaginary sample size:                 1
##   tests used in the learning procedure:  91
##   optimized:                             TRUE
```

```
## BN 2 - score: bde - restart: 100
```

```
##
##   Bayesian network learned via Score-based methods
##
##   model:
##    [A][T][B][S|B][L|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:                                  Bayesian Dirichlet (BDe)
##   graph prior:                            Uniform
##   imaginary sample size:                  1
##   tests used in the learning procedure:   1729
##   optimized:                              TRUE
```

Under the point model we can see what I mentioned in advance. Already in the visualization we could see that a different model was created by the different initialization of restart. We can also state that BN1 needed less test to learn the procedure. This makes sense because BN1 only had a restart.

Interpretation of the parameter restart: The hill climbing algorithm can get into a local minimum/maximum, by the parameter restart the algorithm is listed more often. This restart is random, with an increase of the restart the probability is reduced to get stuck in a local minimum/maximum. The tradeoff, however, is related to the runtime of the algorithm. Since the problem is not very complex, the hill climbing algorithm can be started 100 times without any problems.

Therefore in the next part I will work with parameter value of restart of 100.

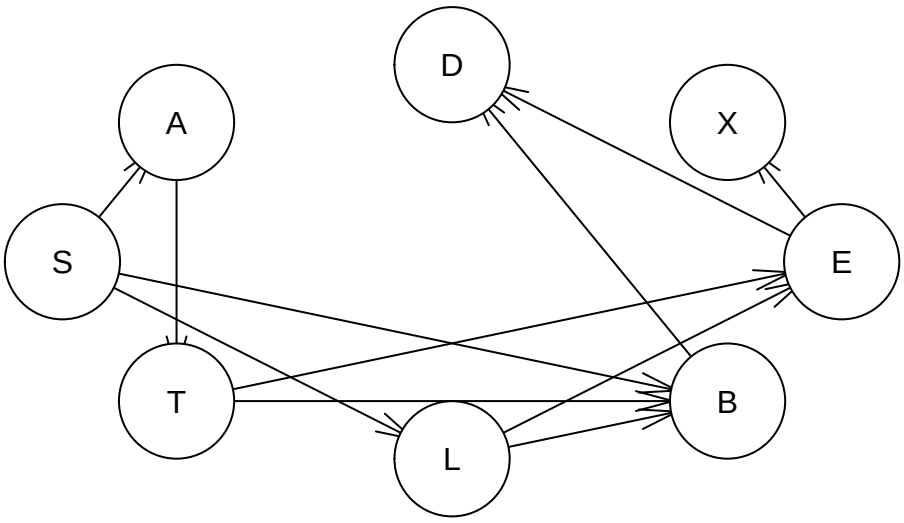
- Parameter: *score*

Creat Bayesian Network object

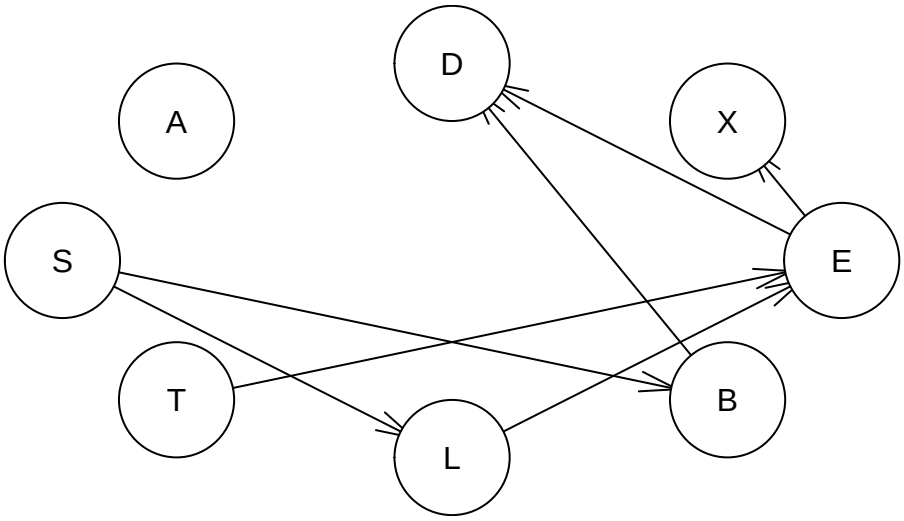
```
# check sore parameter

# create object 3
hc_o3 = hc(x = data,
           score = "aic",
           restart = 100)

# create object 4
hc_o4 = hc(x = data,
           score = "bic",
           restart = 100)
```

- Plot

```
## BN 3 - score: aic - restart: 100
```




```
## BN 2 - score: bic - restart: 100
```



In this plot we can also see that a different model has been created. The number and directions of the arrows now differ.

- all.equal

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

- score

```
## BN 3 - score: aic - restart: 100 - score value:
```

```
## [1] -11129.57
```

```
## BN 4 - score: bic - restart: 100 - score value:
```

```
## [1] -11107.29
```

The third Bayesian Network has a lower score worth.

- Object information

```
## The BN 3:
```

```
##
##   Bayesian network learned via Score-based methods
##
##   model:
##    [S][A|S][L|S][T|A][B|S:T:L][E|T:L][X|E][D|B:E]
##   nodes:                                 8
##   arcs:                                  11
##     undirected arcs:                     0
##     directed arcs:                       11
##   average markov blanket size:           3.50
##   average neighbourhood size:            2.75
##   average branching factor:              1.38
##
##   learning algorithm:                    Hill-Climbing
##   score:                                 AIC (disc.)
##   penalization coefficient:              1
##   tests used in the learning procedure:  1920
##   optimized:                             TRUE
```

```
## The BN 4:
```

```
##
##   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:              4.258597
##   tests used in the learning procedure:  1867
##   optimized:                             TRUE
```

Interpretation parameter score: We can also see here that the parameter score has a big influence on the model creation. Not surprisingly, completely new structures emerge.

Question 2

Learn a BN from 80 % of the Asia dataset. The dataset is included in the *bnlearn* package. To load the data, run *data("asia")*. Learn both the structure and the parameters. Use any learning algorithm and settings that you consider appropriate. 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 so, you **have** to use exact or approximate inference with the help of the *bnlearn* and *gRain* packages, i.e. you are not allowed to use functions such as predict. Report the confusion matrix, i.e. true/false positives/negatives. Compare your results with those of the true Asia BN, which can be obtained by running:

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

Hint: You already know the Lauritzen-Spiegelhalter algorithm for inference in BNs, which is an exact algorithm. There are also approximate algorithms for when the exact ones are too demanding computationally. 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. When you try to load the package *gRain*, you will get an error as the package *RBGL* cannot be found. You have to install this package by running the following two commands (answer no to any offer to update packages):

Results Question 2

Split the data into train and test set

```
# learn 80% of the data set
# for this we split into train and test set

# I used the code provided in the machine learning course
# create train and test set
n = dim(data)[1]
set.seed(12345)
id = sample(1:n, floor(n*0.8))
train=data[id,]
test=data[-id,]
```

Learn the BN

```
# learn the BN

### structure
bn_structure = hc(x = train,
                  score = "bic",
                  restart = 100)


### parameters
bn_parameters = bn.fit(bn_structure,
                       data = train)

### we want to do inferences
## for this we need to convert the BN object
# Convert and compile
bn_parameters_gRain = as.grain(bn_parameters)
bn_parameters_gRain_compile = compile(bn_parameters_gRain)
```

Make the prediction

```
# make prediction via pred_function


pred_e2 = pred_function(BN_model = bn_parameters_gRain_compile,
                        data = test,
                        obs_nodes = colnames(data[-2]),
                        obs_nodes_index = c(-2),
                        pred_node = "S")
```

The result of the prediction

```
## Confusion matrix of the prediction
```

	no	yes
no	322	146
yes	120	412

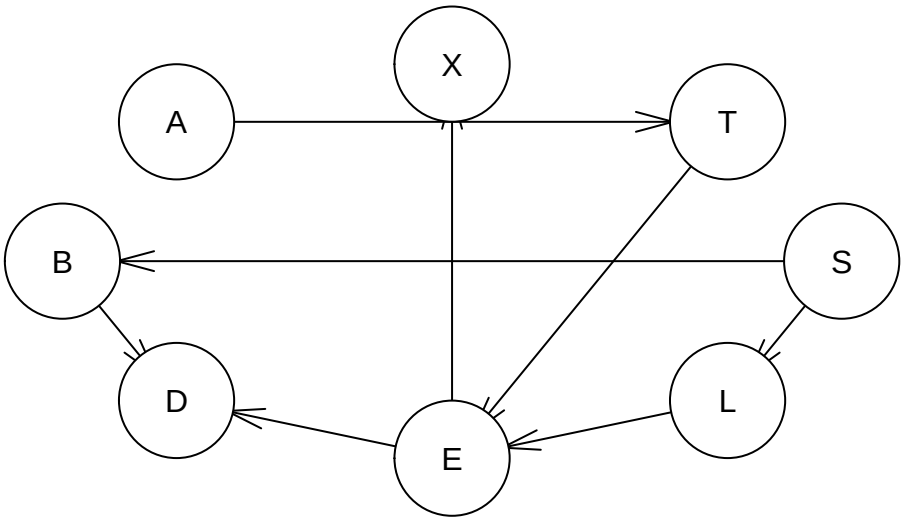
```
## Confusion matrix proportional
```

	no	yes
no	0.322	0.146
yes	0.120	0.412

```
## Classification rate

## [1] 0.734
```

Compare results to the true Aisa BN

The true Asia BN



Make the prediction

```
# make prediction via pred_function


pred_e2_true = pred_function(BN_model = bn_parameters_gRain_compile_true,
```

```
                                data = test,
                                obs_nodes = colnames(data[-2]),
                                obs_nodes_index = c(-2),
                                pred_node = "S")
```

The result of the true prediction

```
## Confusion matrix of the prediction
```

	no	yes
no	322	146
yes	120	412

```
## Confusion matrix proportional
```

	no	yes
no	0.322	0.146
yes	0.120	0.412

```
## Classification rate

## [1] 0.734
```

Question 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.

Hint: You may want to use the function *mb* from the *bnlearn* package.

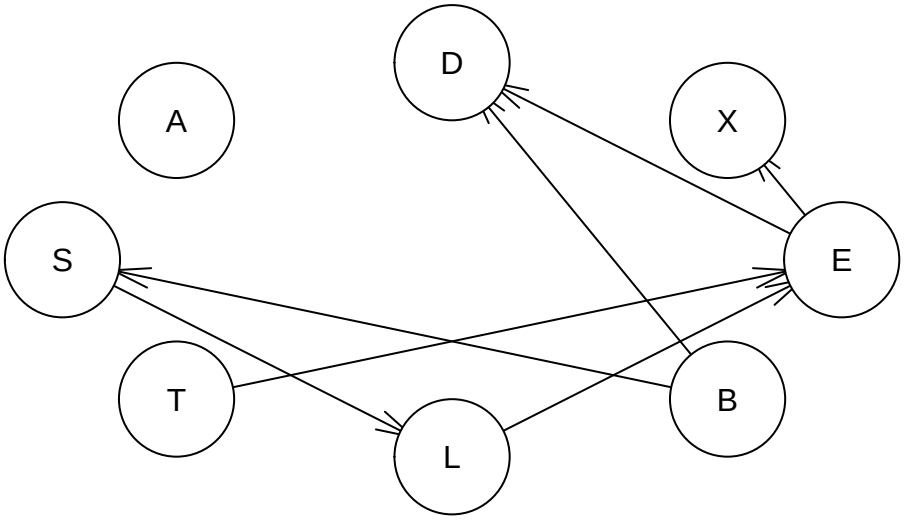
Results Question 3

Create the model

```
############################## - Question 3
mb_object = mb(bn_parameters,"S")
mb_object
```

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

As we can see in the visualization below is S parent of L and B.



Make the prediction

```
# we know that S has two childs,
# this we can observe in the visualization above,
# for the predict function we need to figure out which index the child have in the data
index1 = which(colnames(data) == mb_object[1])
index2 = which(colnames(data) == mb_object[2])

pred_e3 = pred_function(BN_model = bn_parameters_gRain_compile,
                        data = test,
                        obs_nodes = mb_object,
                        obs_nodes_index = c(index1:index2),
                        pred_node = "S")
```

The result of the prediction

```
## Confusion matrix of the prediction
```

	no	yes
no	322	146

	no	yes
yes	120	412

```
## Confusion matrix proportional
```

	no	yes
no	0.322	0.146
yes	0.120	0.412

```
## Classification rate
## [1] 0.734
```

Question 4

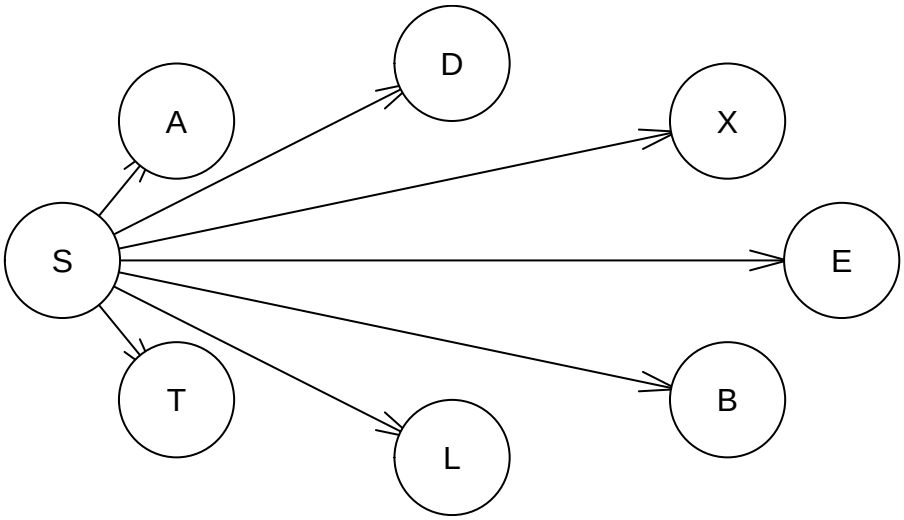
Repeat the exercise (2) using a naive Bayes classifier, i.e. the predictive variables are independent given the class variable. See p. 380 in Bishop's book or Wikipedia for more information on the naive Bayes classifier. Model the naive Bayes classifier as a BN. You **have** to create the BN by hand, i.e. you are not allowed to use the function *naive.bayes* from the *bnlearn* package.

Hint: Check http://www.bnlearn.com/examples/dag/ to see how to create a BN by hand.

Results Question 4

Create a BN, the predictive varaibles should be independent

```
# Creating an empty network
# 1. create a empty BN object
BN_hand = empty.graph(colnames(data))

# Creating a network structure
# With a specific arc set
# 2. creat a matrix which discribes the structure of the BN
arc.set = matrix(c("S", "A",
                   "S", "T",
                   "S", "L",
                   "S", "B",
                   "S", "E",
                   "S", "X",
                   "S", "D"),
                 ncol = 2, byrow = TRUE,
                 dimnames = list(NULL, c("from", "to")))

# 3. asign the object
arcs(BN_hand) = arc.set
```



Fit the BN

```
# learn

### parameters
```

```
bn_parameters_e4 = bn.fit(BN_hand,
                          data = train)

# we want to do inferences

# Convert and compile
bn_parameters_gRain_e4 = as.grain(bn_parameters_e4)
bn_parameters_gRain_compile_e4 = compile(bn_parameters_gRain_e4)
```

Make the prediction

```
pred_e4 = pred_function(BN_model = bn_parameters_gRain_compile_e4,
                        data = test,
                        obs_nodes = colnames(data[-2]),
                        obs_nodes_index = c(-2),
                        pred_node = "S")
```

The result of the prediction

```
## Confusion matrix of the prediction
```

	no	yes
no	349	119
yes	188	344

```
## Confusion matrix proportional
```

	no	yes
no	0.349	0.119
yes	0.188	0.344

```
## Classification rate

## [1] 0.693
```

Question 5

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

Results Question 5

Exercise2_pred	Exercise2_true	Exercise3	Exercise4
0.734	0.734	0.734	0.693

**Interpretation of the results. Here you can see the table with the results of the respective task. It can be seen that the prediction quality for task 2 and 3 are identical. In the first part of the second task we used the hill climbing algorithm to create the structure of the Bayesian network and then determined the parameters using the maximum likelihood method. This approach was so good that this classification has the same quality as the prediction with the True Structure. As already mentioned, we also found that the same prediction quality is present in the third part of the task. This can be explained by the fact that we forecast the S factor. Using Markov blanket, we determine which broad factors have an association with S and use them to create a forecast. Since the quality of the forecast is the same as in task two, we can deduce that we do not have to use all factors of the whole network to make forecasts. Only the factors that are directly related to the factor to be forecast need to be used. The prediction for the 4 task is worse than the others, this is because it is now assumed that all factors are independent of each other. In this example this does not correspond to reality and information from the factors L and B is lost. Therefore, the quality of the forecast is somewhat worse.

Appendix

```
knitr::opts_chunk$set(echo = F)
########################################


# if can´t run the markdown, it could be because of the sourse and package loading
# check if eval=T
# question 2 - below the hint


########################################
# libraries used in the lab
#install.packages("bnlearn")
library(bnlearn)
library(caret)
#install.packages("gRain")
library(gRain)
rm(list = ls())
set.seed(1234)
# the data for question 1
data("asia")
data = asia
#data = as.data.frame(data, stringsAsFactors = T)
# test function hc()
hc_asia = hc(data) # returns an bn object

print("Objects of the data frame: ")
colnames(data)


# plot
plot(hc_asia)

# Miscellaneous utilities
arcs(hc_asia)
#hc_asia$arcs


# Equivalence classes, moral graphs and consistent extensions
vstructs(hc_asia)


# Equivalence classes, moral graphs and consistent extensions
cpdag(hc_asia)

# Test if Two Objects are (Nearly) Equal
all.equal(hc_asia, hc_asia)


############################################
###### In this code I palyed around with some parameters
###### I wanted to find the best parameter combination for question 2
```

```
###### but since it is not part of the exercise I redo this part in a esier way



### init values
# specify the initial structure,
# the number of random restarts,
# the score,
# and the equivalentsample size = imaginary sample size,  in the BDeu score

####### Idea
### create to objects with different values and check if the objects are similar


# which parameter should we change:

####### score:
# - multinomial log-likelihood: loglik
# - Akaike Information Criterion score: aic
# - logarithm of the Bayesian Dirichlet equivalent: bde
# - the Bayesian Information Criterion score: bic
# - logarithm of the Bayesian Dirichlet sparse score: bds
# - ogarithm of the modified Bayesian Dirichlet equivalent score: mbde
# + some "not score equivalent" options

score_options = c("loglik", "aic", "bde", "bic", "bds", "mbde")

####### restart:
# create a vector with different options
restart_options = seq(from = 0, to = 100, by = 10)


### idea
# loop over all the parameter options and save the results in a list
# create another loop to check the different objects to each other

# create list to store the results
hc_objects = list()
counter = 0
for (i in score_options) {
  for (j in restart_options) {
    counter = counter +1
    hc_objects[[counter]] = hc(x = data,
                               restart = j,
                               score = i)
  }
}

# check the objects to each other
# this should be in a lopp
all.equal(hc_objects[[1]], hc_objects[[2]])
```

```r
### check the score
# in the next exercise do we have to choose a setting and algo
# I want to take the best setting
# there for I compate the core of all the models
score_restuls = vector("numeric", length = length(hc_objects))
for (i in 1:length(score_restuls)) {
  score_restuls[i] = score(hc_objects[[i]], data)
}
# find best cose
score_restuls_min = min(score_restuls)
score_restuls_min_indx = which.min(score_restuls)
# check if the min result apears more often
sum(score_restuls == score_restuls_min)



# create object 1
hc_o1 = hc(x = data,
           score = "bde",
           restart = 50)

# create object 2
hc_o2 = hc(x = data,
           score = "bde",
           restart = 100)


# check on similarity
all.equal(hc_o1, hc_o2)




# hc(x = data,
#    start = #something , #an object of class bn, the preseeded directed acyclic graph used to initiali
#    # the algorithm. If none is specified, an empty one (i.e. without any arc) is used.
#    restart = # something , # an integer, the number of random restarts.
#    score = NULL #something # a character string, the label of the network score to be used in the alg
#    # ?network-scores {bnlearn}
#
#    )


# BDeu score
# the logarithm of the Bayesian Dirichlet equivalent (uniform) score (bde)
# (also denoted BDeu), a score equivalent Dirichlet posterior dens


plot(hc_o1)

plot(hc_o2)

############################### - Question 1
```

```
###### Thinking
# Check the parameter restart - same score & compare
# Check different socres with one restart - score & compare



# create object 1
hc_o1 = hc(x = data,
           score = "bde",
           restart = 1)

# create object 2
hc_o2 = hc(x = data,
           score = "bde",
           restart = 100)
cat("BN 1 - score: bde - restart: 100")
plot(hc_o1)


cat("BN 2 - score: bde - restart: 1")
plot(hc_o2)

# vstructs(hc_o1)
# vstructs(hc_o2)
all.equal(hc_o1,hc_o2)
# check score
cat("BN 1 - score: bde - restart: 1 - score value: ")
score(x = hc_o1,
      data = data)

cat("BN 2 - score: bde - restart: 100 - score value: ")
score(x = hc_o2,
      data = data)
# check the overall model
# cpdag(hc_o1)
# cpdag(hc_o2)
cat("BN 1 - score: bde - restart: 1")
hc_o1
cat("BN 2 - score: bde - restart: 100")
hc_o2

# check sore parameter

# create object 3
hc_o3 = hc(x = data,
           score = "aic",
           restart = 100)

# create object 4
hc_o4 = hc(x = data,
           score = "bic",
           restart = 100)
cat("BN 3 - score: aic - restart: 100")
```

```r
plot(hc_o3)


cat("BN 2 - score: bic - restart: 100")
plot(hc_o4)
all.equal(hc_o3,hc_o4)
# check score
cat("BN 3 - score: aic - restart: 100 - score value: ")
score(x = hc_o3,
      data = data)

cat("BN 4 - score: bic - restart: 100 - score value: ")
score(x = hc_o4,
      data = data)
# check the overall model
# cpdag(hc_o1)
# cpdag(hc_o2)
cat("The BN 3:")
hc_o3
cat("The BN 4:")
hc_o4

dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
source("https://bioconductor.org/biocLite.R")

biocLite("RBGL")
# this code is for mac version
source("http://bioconductor.org/biocLite.R")
biocLite(c("graph", "Rgraphviz", "RBGL"))
install.packages("gRain")
################################ - Question 2
set.seed(1234)
#rm(list = ls())
# learn 80% of the data set
# for this we split into train and test set

# I used the code provided in the machine learning course
# create train and test set
n = dim(data)[1]
set.seed(12345)
id = sample(1:n, floor(n*0.8))
train=data[id,]
test=data[-id,]
###################### Prediction - Function
# function for the upcomming excersises 2-4

### function for the prediction


pred_function = function(BN_model, data, obs_nodes, obs_nodes_index, pred_node){
  ### Input:
  # BN_model: has to be a learn model
  # data: the data we want to you to make the prediction (test data)
```

```r
  # obs_nodes: the nodes we want to investigate - same length as obs_nodes_index
  # obs_nodes_index: the index of the observed - same length as obs_nodes
  # pred_node: the node we want to predict

  pred_s = vector("numeric", length = nrow(data))

  for (i in 1:nrow(data)) {
  state_perd = c()
  for (j in 1:ncol(data)) {
    if(test[i,j] == "no"){
      state_perd = c(state_perd, "no")
    } else{
      state_perd = c(state_perd, "yes")
    }
  }
  findings = setEvidence(object = BN_model,
                         nodes = obs_nodes,
                         states = state_perd[obs_nodes_index])
  # is factor - function need different input
  # need to transform from factor to list
  query = querygrain(object = findings,
                     nodes = pred_node,
                     type = "marginal")

  query_s = query$S

  # make prediction
  pred_s[i] = ifelse(query_s[1] < query_s[2],"yes","no")

  }

  cm = table(test$S, pred_s)
  cm_pro = prop.table(table(test$S, pred_s))

  res_preb = prop.table(table(test$S, pred_s))[1,1] + prop.table(table(test$S, pred_s))[2,2]

  result = list("cm" = cm,
                "cm_prob" = cm_pro,
                "res_preb" = res_preb)
  return(result)

}

##### useful functions for this exercise

### bnlearn package:
# bn.fit
# as.grain
## for approximate inference:
# prop.table
# table
# cpdist
```

```
### gRain package:
# compile
# setFinding
# querygrain

### Thinking
# we have to predict the object S in the training data, S = yes OR S = no
## Therefore I we should remove existing

# learn the BN

### structure
bn_structure = hc(x = train,
                  score = "bic",
                  restart = 100)


### parameters
bn_parameters = bn.fit(bn_structure,
                       data = train)

### we want to do inferences
## for this we need to convert the BN object
# Convert and compile
bn_parameters_gRain = as.grain(bn_parameters)
bn_parameters_gRain_compile = compile(bn_parameters_gRain)
# make prediction via pred_function


pred_e2 = pred_function(BN_model = bn_parameters_gRain_compile,
                        data = test,
                        obs_nodes = colnames(data[-2]),
                        obs_nodes_index = c(-2),
                        pred_node = "S")
cat("Confusion matrix of the prediction")
knitr::kable(pred_e2$cm)
cat("Confusion matrix proportional")
knitr::kable(pred_e2$cm_prob)
cat("Classification rate")
pred_e2$res_preb
plot(dag)
# given is a learn BN
# now we need to fit parameters, compile and make predictions
# at the end compare to the prev predictions

### parameters
bn_parameters_true = bn.fit(dag,
                       data = train)

### we want to do inferences
## for this we need to convert the BN object
# Convert and compile
bn_parameters_gRain_true = as.grain(bn_parameters_true)
```

```r
bn_parameters_gRain_compile_true = compile(bn_parameters_gRain)

# make prediction via pred_function


pred_e2_true = pred_function(BN_model = bn_parameters_gRain_compile_true,
                             data = test,
                             obs_nodes = colnames(data[-2]),
                             obs_nodes_index = c(-2),
                             pred_node = "S")
cat("Confusion matrix of the prediction")
knitr::kable(pred_e2_true$cm)
cat("Confusion matrix proportional")
knitr::kable(pred_e2_true$cm_prob)
cat("Classification rate")
pred_e2_true$res_preb
############################### - Question 3
mb_object = mb(bn_parameters,"S")
mb_object
plot(bn_structure)
# we know that S has two childs,
# this we can observe in the visualization above,
# for the predict function we need to figure out which index the child have in the data
index1 = which(colnames(data) == mb_object[1])
index2 = which(colnames(data) == mb_object[2])

pred_e3 = pred_function(BN_model = bn_parameters_gRain_compile,
                        data = test,
                        obs_nodes = mb_object,
                        obs_nodes_index = c(index1:index2),
                        pred_node = "S")
cat("Confusion matrix of the prediction")
knitr::kable(pred_e3$cm)
cat("Confusion matrix proportional")
knitr::kable(pred_e3$cm_prob)
cat("Classification rate")
pred_e3$res_preb
############################### - Question 4
BN_hand = empty.graph(colnames(data))
class(BN_hand)
BN_hand

arc.set = matrix(c("S", "A",
                   "S", "T",
                   "S", "L",
                   "S", "B",
                   "S", "E",
                   "S", "X",
                   "S", "D"),
                   ncol = 2, byrow = TRUE,
                   dimnames = list(NULL, c("from", "to")))
arc.set
```

```r
arcs(BN_hand) = arc.set
BN_hand
plot(BN_hand)
# Creating an empty network
# 1. create a empty BN object
BN_hand = empty.graph(colnames(data))

# Creating a network structure
# With a specific arc set
# 2. creat a matrix which discribes the structure of the BN
arc.set = matrix(c("S", "A",
                   "S", "T",
                   "S", "L",
                   "S", "B",
                   "S", "E",
                   "S", "X",
                   "S", "D"),
                   ncol = 2, byrow = TRUE,
                   dimnames = list(NULL, c("from", "to")))

# 3. asign the object
arcs(BN_hand) = arc.set
plot(BN_hand)
# learn

### parameters
bn_parameters_e4 = bn.fit(BN_hand,
                          data = train)

# we want to do inferences

# Convert and compile
bn_parameters_gRain_e4 = as.grain(bn_parameters_e4)
bn_parameters_gRain_compile_e4 = compile(bn_parameters_gRain_e4)

pred_e4 = pred_function(BN_model = bn_parameters_gRain_compile_e4,
                        data = test,
                        obs_nodes = colnames(data[-2]),
                        obs_nodes_index = c(-2),
                        pred_node = "S")
cat("Confusion matrix of the prediction")
knitr::kable(pred_e4$cm)
cat("Confusion matrix proportional")
knitr::kable(pred_e4$cm_prob)
cat("Classification rate")
pred_e4$res_preb
result_df = data.frame("Exercise2_pred" = pred_e2$res_preb,
                       "Exercise2_true" = pred_e2_true$res_preb,
                       "Exercise3" = pred_e3$res_preb,
                       "Exercise4" = pred_e4$res_preb)

knitr::kable(result_df)
```