Adv ML Lab1

Andreas Stasinakis(andst745)

September 11, 2019

Contents

```r
#Importing the libraries
library("bnlearn")
library("Rgraphviz")
library("gRain")
library("gridExtra")
```

Question 1.1 Hill - Climbing for different parameters

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 included 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.

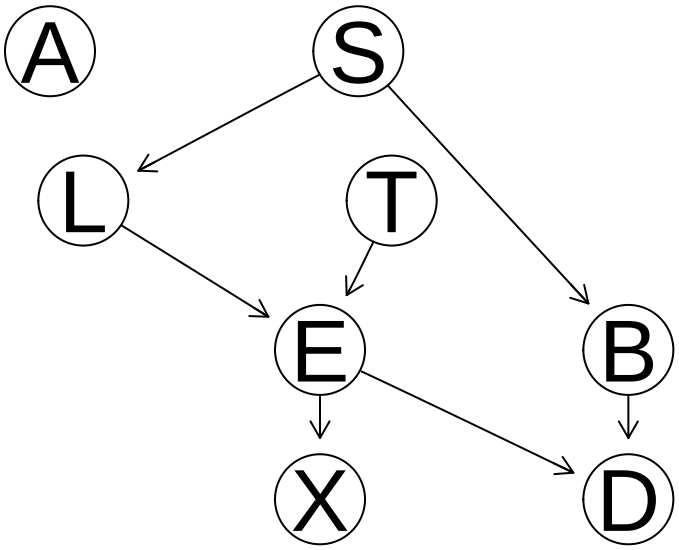
```r
set.seed(1234567890)
#load the data
data("asia")

#We will create several  objects of BN in order to compare them
#first approach: default hc
BN_bic = hc(asia, score = "bic") #default score BIC

#Truy with different score function
BN_aic =  hc(asia,score = "aic")

#Same score function but different restart number
BN_10 = hc(asia,restart = 10)
BN_50 = hc(asia,restart = 50)
BN_100 = hc(asia,restart = 100)
BN_500 = hc(asia,restart = 500)
BN_1000 = hc(asia,restart = 1000)


#Instead of starting from an empty graph, we will start from a random graph
rdm_graph = random.graph(colnames(asia))
BN_rdm500 = hc(asia,restart = 500,start = rdm_graph)
```

In this task we compare different graphs, which all created using the hill-climbing algorithm. As we can see below though, different runs of the algorithm, will give us different graphs. Starting from an initial state, the algorithm will visit the neighbors and it will try to find the optimal solution. The problem is that HC does not guarantee *global* optima, but only *local*. Therefore, different runs of HC may give different graphs. The reason why HC may stuck in a local optima is the following. Assume that the logarithm have to choose between two graphs. If the graphs have the same structure(equivelant graphs), but different directions between the nodes, means that they have also the same score. So the algorithm is not able to choose and it will choose randomly. As a result, there is a chance that it will choose the wrong path, which will drive to a wrong final result(local optima). As a conclusion, the HC algorithm will give us equivelant graphs(which means same statistical models), but not *equal* graphs.

In order to confirm the above statement, we present some different runs of HC algorithm, using different parameters. We first start with different score functions(AIC vs BIC). The easiest way to understand that there is a difference between the two graphs is by plotting them.

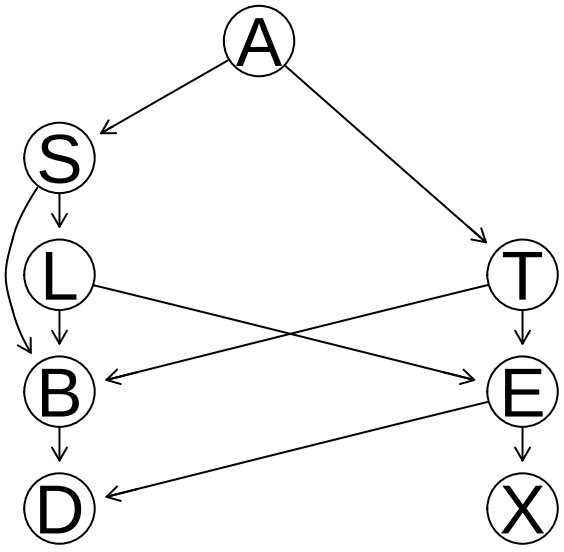
Using BIC as a score function:

```
graphviz.plot(BN_bic)
```



Using AIC as a score function:

```
graphviz.plot(BN_aic)
```



we also print their arcs in order to compare. As we can observe, the *AIC* score has a more complicated graph(4 more arcs).

```
bic_arcs = arcs(BN_bic)
aic_arcs = arcs(BN_aic)
print(list("Arcs using BIC:" = t(bic_arcs)))
```

```
## $`Arcs using BIC:`
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## from "B"  "L"  "E"  "S"  "T"  "E"  "S"
## to   "D"  "E"  "X"  "B"  "E"  "D"  "L"
```

```
print(list("Arcs using AIC:" = t(aic_arcs)))
```

```
## $`Arcs using AIC:`
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
## from "B"  "L"  "E"  "S"  "T"  "E"  "S"  "T"  "A"  "L"   "A"
## to   "D"  "E"  "X"  "B"  "E"  "D"  "L"  "B"  "T"  "B"   "S"
```

We also create different graphs, using AIC as a score function, but changing the number of restart. That means that the algorithm will start many times the procedure in order to find the optimal solution. It is clear again that we will have different graphs, but as the number of restarts increases, the graphs have more chances to look similar to the true value.

First we compare the graphs for Res = 10 and Res = 50. As we can see they are not alike because of different arc sets.

```
print(list("Number of restarts : 10 vs 50 ",all.equal(BN_10,BN_50)))
```

```
## [[1]]
## [1] "Number of restarts : 10 vs 50 "
##
## [[2]]
## [1] "Different arc sets"
```

```
print(list("Number of restarts : 100 vs 500 ",all.equal(BN_100,BN_500)))
```

```
## [[1]]
## [1] "Number of restarts : 100 vs 500 "
##
## [[2]]
## [1] "Different arc sets"
```

```
print(list("Number of restarts : 100 vs 1000 ",all.equal(BN_100,BN_1000)))
```

```
## [[1]]
## [1] "Number of restarts : 100 vs 1000 "
##
## [[2]]
## [1] "Different arc sets"
```

```
vstructs(BN_10)
```

```
##      X   Z   Y
## [1,] "T" "E" "L"
## [2,] "B" "D" "E"
```

```
vstructs(BN_100)
```

```
##      X   Z   Y
## [1,] "T" "E" "L"
## [2,] "B" "D" "E"
```

```
vstructs(BN_1000)
```

```
##      X   Z   Y
## [1,] "T" "E" "L"
## [2,] "B" "D" "E"
```

For all the situations above, using multiply restarts, the graphs are not equal but equivelant. The best way to understand that is to compare their v structure. As we can see above, the algorithm has the same v structure

for different number of restarts. That means that those networks are equivelant(same statistical model, which leads to same predictions), but different(not equal) graphs.

Their major difference is the direction of some nodes. Of course some times we may take the same graphs, because there is some randomness when the algorithm runs.

Finally, we run the algorithm using different initial graphs. One empty graph and one random graph. Again it is clear that the two graphs are similar *but* not equal.

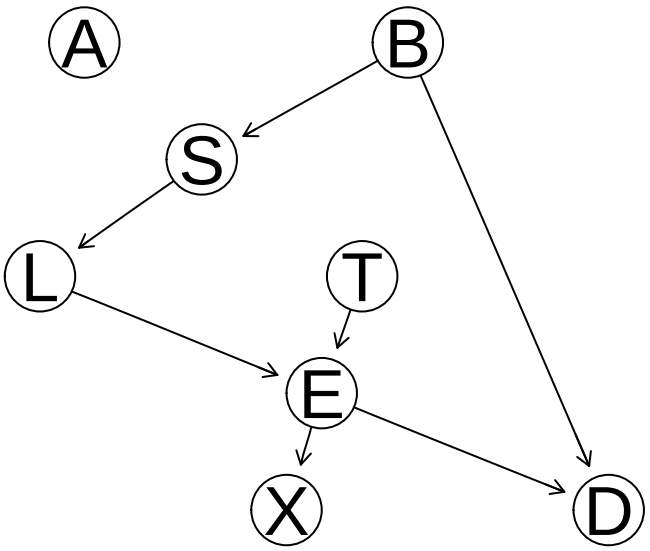
```
#print the arcs for each graph
print(list("Arcs for empty graph:" = t(arcs(BN_500))))
```

```
## $`Arcs for empty graph:`
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## from "B"  "S"  "L"  "T"  "E"  "B"  "E"
## to   "S"  "L"  "E"  "E"  "D"  "D"  "X"
```

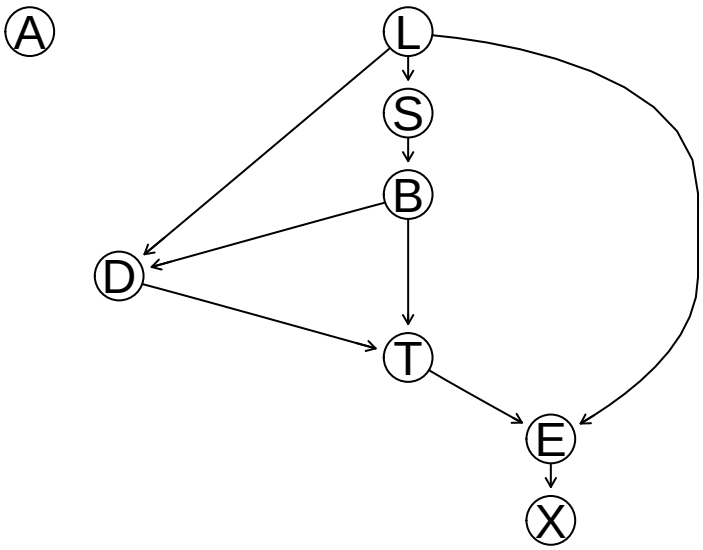
```
print(list("Arcs for random graph:" = t(arcs(BN_rdm500))))
```

```
## $`Arcs for random graph:`
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## from "L"  "S"  "D"  "T"  "B"  "E"  "L"  "L"  "B"
## to   "S"  "B"  "T"  "E"  "D"  "X"  "D"  "E"  "T"
```

```
graphviz.plot(BN_500)
```




```
graphviz.plot(BN_rdm500)
```



Question 1.2 Train a BN and compare it with the true one.

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]").

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): source("https://bioconductor.org/biocLite.R") biocLite("RBGL")

```
#Function for the predictions

Pred_function = function(test1,BN,node,col){

  #Input: test1 = Test data to classify(should be character)
  # BN = The bayessian network which is already compliled
  # node = The node i want to obtain the posterior probs
  #col = The column i do not want as a predictor

  #a for loop to find all the posterior probabilities for each row in the test set
  col_names = colnames(test1)
  predictions = rep(0, nrow(test1))
  #for each row of the test data set i calculate the exact pos probabilities
  #For the state S
```

```
  for (r in 1:nrow(test1)) {
    temp = setEvidence(BN, nodes = col_names[-col],states = test1[r,-col])
    temp_prob = as.numeric(unlist(querygrain(temp,nodes = node)))

    #here we predict w.r.t the probabilities of yes or no
    if(temp_prob[1]> temp_prob[2]){
      predictions[r] = "no"
    }else{
      predictions[r] = "yes"
    }

  }
  return(predictions)
}


#Split the data into training 80% and test 20%
set.seed(12345)
data("asia")
n = dim(asia)[1]
id = sample(1:n, floor(n*0.8))
train = asia[id,]
test = asia[-id,]

IAMB = iamb(x = train)

#fit the model for the training data
model = bn.fit(IAMB,train)

#tranform it to grain
bn_model = as.grain(model)

#compile the BN
#create a junction tree and establishing clique potentials.
BN = compile(bn_model)

#as character for the dataset
# NOTE : setEvidence NEEDS characters not factors for the states
test1 = apply(test, 2, as.character)


#Confusion matrix
predictions_BN = Pred_function(test1 = test1,BN,"S",2)
conf_matrix = table(test$S,predictions_BN)

#misclassification rate
accur_aic = (sum(diag(conf_matrix))/sum(conf_matrix))


################## TRUE BN NETWORK #####################################
dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")

true_model = bn.fit(dag,train)
```

```
#tranform it to grain
true_model = as.grain(true_model)

#compile the BN
#create a junction tree and establishing clique potentials.
BN_true = compile(true_model)

#make the predictions using the function
predictions_true = Pred_function(test1 = test1,BN_true,"S",2)
#Confusion matrix
conf_matrix_true = table(test$S,predictions_true)

#misclassification rate
accur_true = sum(diag(conf_matrix_true))/sum(conf_matrix_true)
```

Question 1.3 Train a BN using the Markov Blanket method(mb).

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.

```
#Here we do not use all the nodes for predicting
mb_S = mb(model, "S") #ask for a bn.fit object

#a for loop to find all the posterior probabilities for each test set
predictions_mb = rep(0, nrow(test))
for (r in 1:nrow(test)) {
  temp = setEvidence(BN, nodes = mb_S,states = test1[r,mb_S])
  temp_prob = as.numeric(unlist(querygrain(temp,nodes = "S")))

  if(temp_prob[1]> temp_prob[2]){
    predictions_mb[r] = "no"
  }else{
    predictions_mb[r] = "yes"
  }

}

#Confusion matrix
conf_matrix_mb = table(test$S,predictions_mb)

#misclassification rate
accur_mb = sum(diag(conf_matrix_mb))/sum(conf_matrix_mb)
```

Question 1.4 Train a BN model using the naive Bayes classifier.

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.

```
#fit a naive bayes
bayes = naive.bayes(train,training = "S")
pred = predict(bayes,data = test)
mis_rate_bayes1 = length(which(as.vector(pred) == test$S))/nrow(test)


#Now we have to create our BN
col_names = colnames(test)
e = empty.graph(col_names)


#fit
dag_bayes = model2network("[S][A|S][T|S][L|S][B|S][E|S][X|S][D|S]")


bayes_model = bn.fit(dag_bayes,train)

#tranform it to grain
bayes_model = as.grain(bayes_model)

#compile the BN
#create a junction tree and establishing clique potentials.
BN_bayes = compile(bayes_model)

#Make the predictions
predictions_bayes = Pred_function(test1 = test1,BN_bayes,"S",2)




#Confusion matrix
conf_matrix_bayes = table(test$S,predictions_bayes)

#misclassification rate
accur_bayes = sum(diag(conf_matrix_bayes))/sum(conf_matrix_bayes)
```

Question 1.5 Comparison and results.

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

As we can seen from the table below, the lower accuracy is given by the naive Bayes classifier. The reason why that happens is the strong Bayes assumption, which is that all the events(nodes) are conditionally independent to each other. Unfortunately, as we can see from the true graph, there are many dependences between the nodes. Therefore, the bayes classifier *does not* capture all the true relatioships and that is the reason why it performs less efficient than the other learning structures. Moreover, the classifier, because of the assumption, creates *wrong* relationships bettwen node S and other nodes, which do not exist in the real graph. In this way, the classifier is "forced" to learn the structure from the noise. The joint posterior distribution, would be like

$$P(S, A, T, L, B, E, X, D) = P(S)P(A|S)P(T|S)P(L|S)P(B|S)P(E|S)P(X|S)P(D|S)$$

We now compare the IAMB BN we obtain using all the variables, with the one for which we used the Markov blanket approach. The two models have the same accuracy score and confusion matrix. This is because in

this case, we only care about the node "S". In both cases, "S" depends only on nodes "B" and "D". Therefore, in Markov blanket, we want to estimate the distribution $P(S/B, D)$. In order to do that, we marginalize with respect to all the other nodes. In the graph though, the node S does only depend on those two nodes, which means that the marginalization will give us the same result as the joint distribution.

Finally, it is obvious that we have a slightly lower accuracy between the model we train and the real graph. The reason why that happens is because the IAMB algorithm was not able to "catch" the dependency between the node "S" and "L". So for our model "S" and "L", are independent while they should not. Therefore, the joint distribution in our model is

$$P(S, A, T, L, B, E, X, D) = P(A)P(S)P(T)P(L)P(X)P(D)P(B|S : D)P(E|T : L)$$

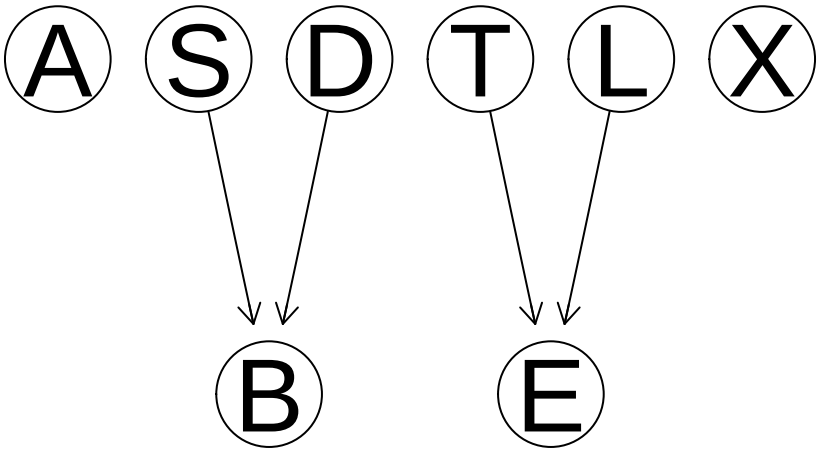
. On the other hand, the joint distribution of the true graph is

$$P(S, A, T, L, B, E, X, D) = P(A)(S)(T|A)(L|S)(B|S)(D|B : E)(E|T : L)(X|E)$$

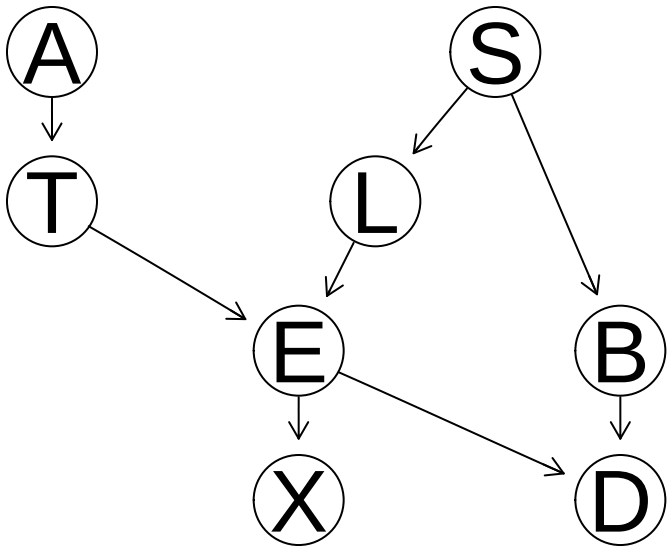
.

As we mentioned before, our accuracy is lower for the IAMB method we learn the structure of the graph. If we want a better accuracy(equal to the true graph), we could learn the structure using Hill climbing algorithm. In that case, we may obtain a more complicated graph, which we trying to avoid, but we increase the accuracy of the model.

```
graphviz.plot(IAMB)
```




```
graphviz.plot(dag)
```




```
accuracies = data.frame(accur_aic,accur_true,accur_mb,accur_bayes)
colnames(accuracies) = c("IAMB", "TRUE", "MB", "BAYES")
knitr::kable(x = accuracies, caption = "Accuracy for all methods ")
```

Table 1: Accuracy for all methods

IAMB	TRUE	MB	BAYES
0.722	0.734	0.722	0.693

```
#Print the results
knitr::kable(x = conf_matrix, caption = "confusion matrix for IAMB model")
```

Table 2: confusion matrix for IAMB model

	no	yes
no	330	138
yes	140	392

```
knitr::kable(x = conf_matrix_true, caption = "confusion matrix for the true Graph")
```

Table 3: confusion matrix for the true Graph

	no	yes
no	322	146
yes	120	412

```
knitr::kable(x = conf_matrix_mb, caption = "confusion matrix for Markov Blanket")
```

Table 4: confusion matrix for Markov Blanket

	no	yes
no	330	138
yes	140	392

```
knitr::kable(x = conf_matrix_bayes, caption = "confusion matrix for bayes BN")
```

Table 5: confusion matrix for bayes BN

	no	yes
no	349	119
yes	188	344