Adv ML Lab1

Andreas Stasinakis(andst745) & Mim Kemal Tekin(mimte666) & Stefano Toffol(steto820) & Bruno Barakat(bruba569)

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.

When running two times the hill-climbing algorithm on the same data, the bayesian structures obtained can be different. We can use two different things to prove that two networks are not equivalent :

-If they have a different BDE score, then the networks are not equivalent because the BDE score is network equivalent (it gives the same score to equivalent graphs)

-If they have different v-structures

Learning two structure from the same score can yield graphs that differ only by their edge directions. It is because it is impossible for the algorithm to know the actual causality between variables, since it's only learning from observations. So even if edges directions are different, the BDE score cannot distinguish between them, we can say nothing about the equivalence of the graphs just by looking at the score. If we compare the two network's V-structures, we can see they are the same, therefore, in this case, the two graphs are equivalent.

```r
set.seed(1256)
ntwk1=hc(asia, restart = 100, score="bic")
ntwk2=hc(asia, restart = 100, score="bic")

all.equal(ntwk1, ntwk2)
```

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

```r
cat('Ntwk1 Score', bnlearn::score(ntwk1, asia, type="bde"), '\nNtwk2 Score',
    bnlearn::score(ntwk2, asia, type="bde"), '\n')
```

```
## Ntwk1 Score -11095.79
## Ntwk2 Score -11095.79
```

```r
eqntwk1=cpdag(ntwk1, moral = FALSE)
eqntwk2=cpdag(ntwk2, moral = FALSE)

all.equal(eqntwk1, eqntwk2)
```

```
## [1] TRUE
```

When learning with different imaginary sample sizes (ISS), the obtained networks can differ more. In this case, learning from the BDE score and with two different ISS, the obtained networks differ by their connections. The BDE score for the graphs are different, the *netwk*2 having a better one because of the added connection between A and T. Since the BDE score is network-equivalent, the resulting bayesian networks are definitely not equivalent in this case. Comparing V-structures confirms this non-equivalency.

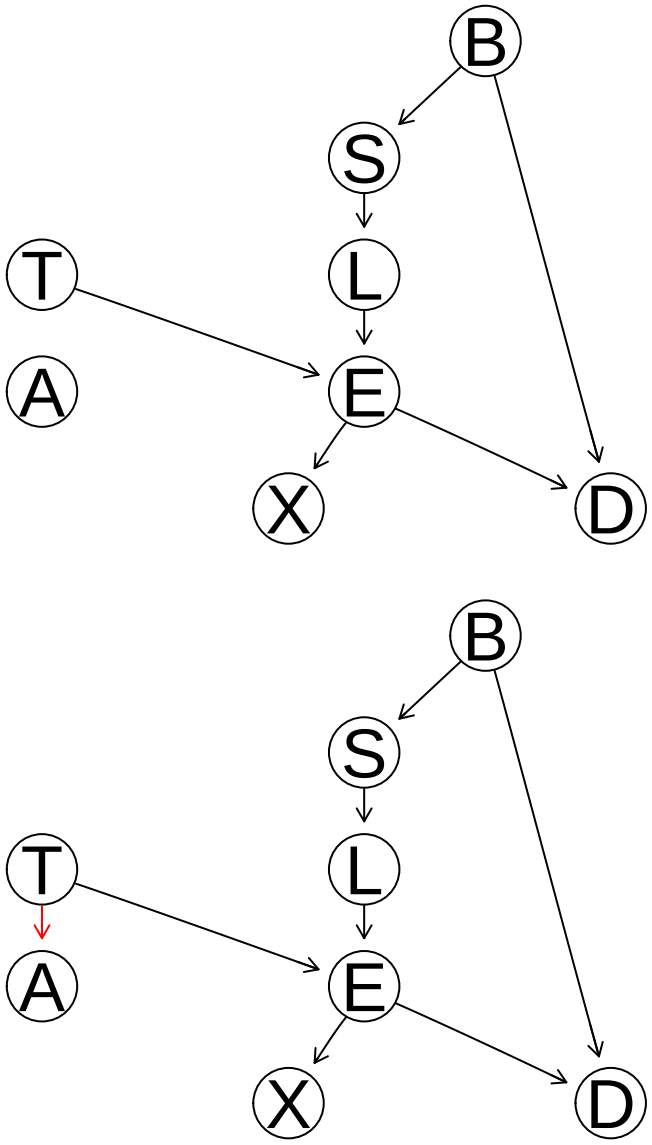
```
ntwk1=hc(asia, restart = 100, iss=1, score="bde")
ntwk2=hc(asia, restart = 100, iss=2, score="bde")

all.equal(ntwk1, ntwk2)
```

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

```
cat('Ntwk1 Score', bnlearn::score(ntwk1, asia, type="bde"), '\nNtwk2 Score',
    bnlearn::score(ntwk2, asia, type="bde"), '\n')
```

```
## Ntwk1 Score -11095.79
## Ntwk2 Score -11096.64
```

```
graphviz.compare(ntwk1, ntwk2)
```




```
eqntwk1=cpdag(ntwk1, moral = FALSE)
eqntwk2=cpdag(ntwk2, moral = FALSE)
```

```
all.equal(eqntwk1, eqntwk2)
```

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

Finally, when learning from different scores, the graphs can differ completely. For example, using the log likelihood will put connections between all variables since it's not penalizing the number of parameters. In the case below, the two models are definitely not equivalent, proven by the different BDE scores or the different V structures.

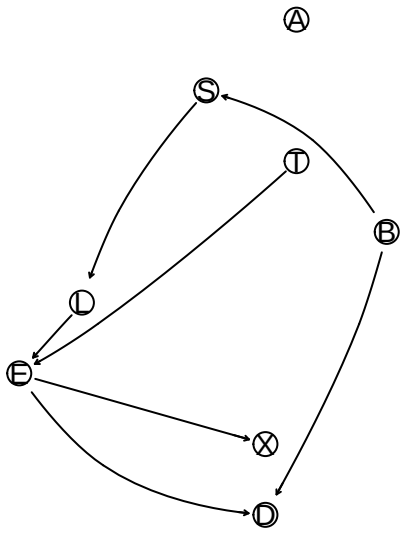
```
ntwk1=hc(asia, restart = 100, iss=1, score="bde")
ntwk2=hc(asia, restart = 100, score="loglik")

all.equal(ntwk1, ntwk2)
```

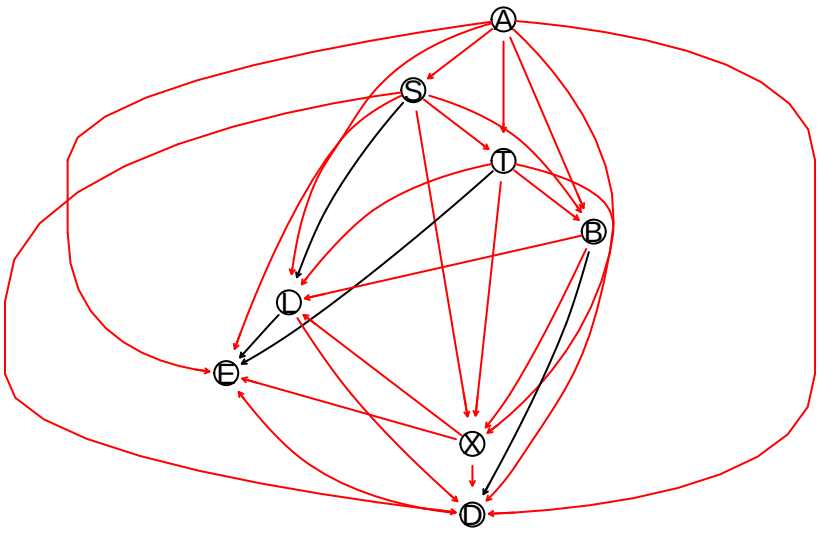
```
## [1] "Different number of directed/undirected arcs"
```

```
cat('Ntwk1 Score', bnlearn::score(ntwk1, asia, type="bde"), '\nNtwk2 Score',
    bnlearn::score(ntwk2, asia, type="bde"), '\n')
```

```
## Ntwk1 Score -11095.79
## Ntwk2 Score -11260.43
```

```
graphviz.compare(ntwk1, ntwk2)
```






```
eqntwk1=cpdag(ntwk1, moral = FALSE)
eqntwk2=cpdag(ntwk2, moral = FALSE)

all.equal(eqntwk1, eqntwk2)
```

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

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

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

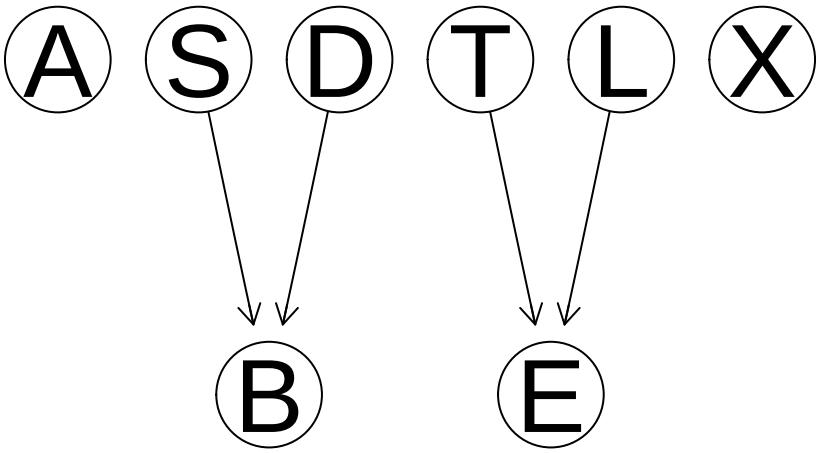
. On the other hand, the joint distribution of the true graph is

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

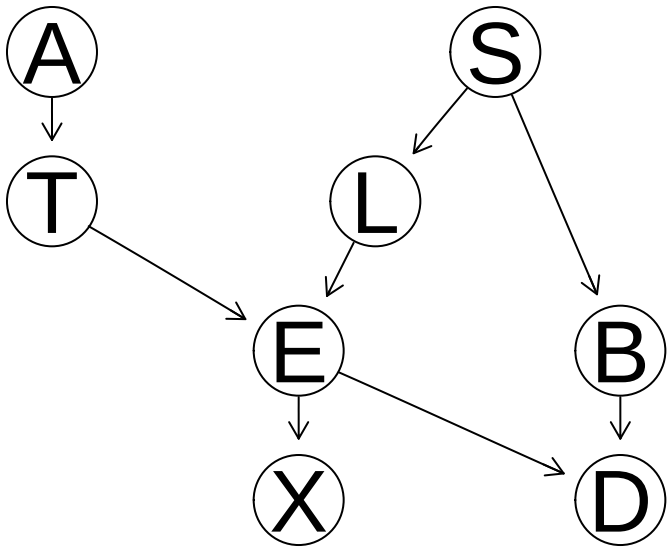
.

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

We also print the results from the other students. We can say that using different algorithm to learn the structure will give different results. As we can see, all the accuracies are equal or really close to the value of the true graph. The reason why this is happening is because, all those graphs(learning structures), spot all the important dependencies of the true graph in constrast to the IAMB method which described above. The disadvantage of those method is that they also found some unesseccery dependencies, which make the graph more complicated and computational expensive(increasing the number of parameters).

Table 6: Accuracy for other students

	andst745	mimte666	steto820	bruba569
Structure	BDE IIS=2	IAC	IAC,rest = 100	IAMB
Par. Learning	MLE	MLE	MLE	MLE
accuracy	0.733	0.734	0.734	0.722
real BN accuracy	0.734	0.734	0.734	0.734