Advanced Machine Learning - lab01

Lennart Schilling (lensc874)

2019-09-19

Contents

Lab01

$$\left.\begin{array}{l}A \to B\\ B \to A\end{array}\right\}$$ not equal, but equivalent $\Rightarrow$ check for
· skeleton
· v-structures
} cpdag()

only retains the direction of arrow if there is no other equivalent BN with other arrow
e.g. $A \to B$
cpdag: $A - B$

$\boxed{\begin{array}{l}A \to B \to C\\ A \leftarrow B \leftarrow C\\ A \leftarrow B \to C\end{array}}$
all of same equivalent class.
cpdag(): $A - B - C$

v-structure
$\boxed{A \to B \leftarrow C}$
no other equivalent class.
cpdag(): $A \to B \leftarrow C$

Assignment 1. Hill-climbing algorithm.

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").

· look up parameter IS

Importing library and data.

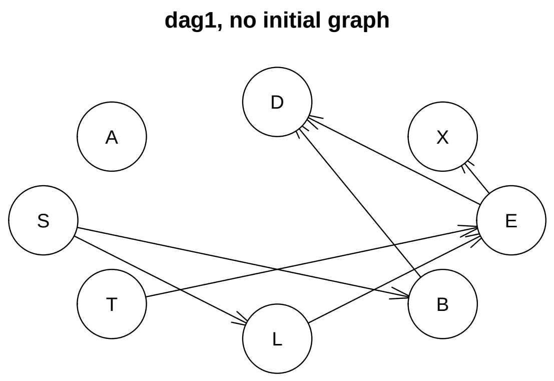
```r
library(bnlearn)
data("asia")
```

Comparing multiple runs of hill-climbing algorithm.

Within this exercise, we will perform the hill-climbing algorithm. In general, this algorithm will add, remove or reverse any edge in an existing graph that improves the Bayesian score (posterior probability) the most. Therefore, it follows the Greedy approach which means that within each iteration, it will decide to move in the direction which immediately optimizes the graph the most. The algorithm stops iterating if a further change of the graph do not directly lead to an improvemenet.

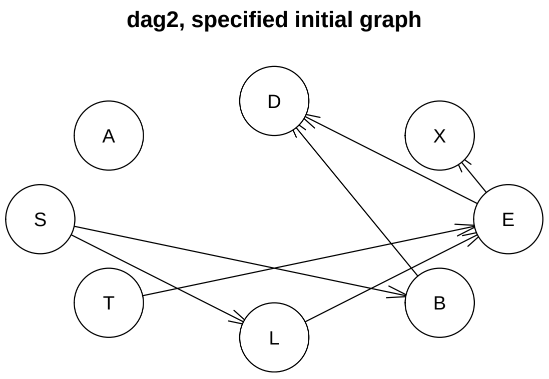
To learn the structure of a Bayesian network, the function `hc()` is used. In the first run, we start the algorithm without specifying any initialized structure. Thus, the algorithm assumes an empty graph as the starting graph.

```r
# Running hc algorithm without specified initial graph.
dag1 = hc(x = asia)
plot(dag1, main = "dag1, no initial graph")
```



In another run, we assume now the derived Bayesian network (*dag1*) as the new initial graph and run the algorithm again.

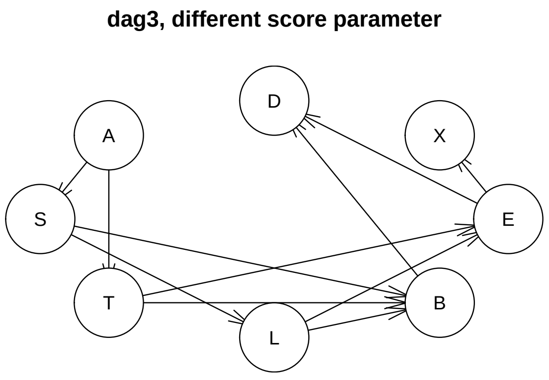
```r
# Running hc algorithm with specified initial graph.
dag2 = hc(x = asia, start = dag1)
plot(dag2, main = "dag2, specified initial graph")
```



No change is visible. This makes sense because as already said the algorithm stops iterating if a further change of the graph do not directly lead to an improvemenet. Since it then stopped at the beginning when *dag1* has been created, *dag2* will not differ from *dag1*, because any change of *dag1* will not increase the Bayesian score.

An adjustment of the `restart`-parameter does also not lead to any changes, because this only results in exactly the same iterations for each restart, because for each iteration, there is only the one best solution again and again. However, a change might be obtained by adjusting the `score`-parameter. By default, the *Bayesian Information Criterion (BIC)* is used. Instead, we will use the *Akaike information criterion (AIC)*.

```
dag3 = hc(x = asia, score = "aic")
plot(dag3, main = "dag3, different score parameter")
```



As it can be seen, the graph has changed and it is shown that the hill-climbing algorithm can return non-equivalent Bayesian network structures. Using a different score leads to different optimal steps within the Greedy-approached algorithm and therefore finally to a different optimal graph.

A comparison between *dag1* and *dag3* confirms that:

```
# Checking equality of dag1 and dag3.
print(all.equal(dag1, dag3))
```

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

Assignment 2. Learning a BN. Performing classification using BN.

Learn a BN from 80 percent 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 perent 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): source("https://bioconductor.org/biocLite.R") biocLite("RBGL")

Importing libraries and data.

First, the required data and libraries will be imported.

```r
# library(bnlearn) # already imported in assignment 1.
library(gRain)
# data("asia") # already imported in assignment 1.
```

Dividing data.

Since the task is to use 80% of the data for learning to classify the remaining 20% of the dataset, the *Asia* dataset will be split.

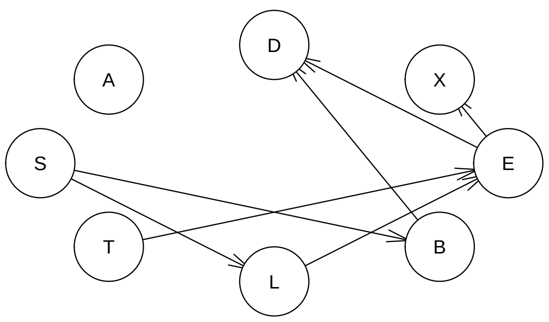
```r
# train (80%) and test set (20%)
n = nrow(asia)
set.seed(12345)
id = sample(1:n, floor(n*0.8))
train = asia[id,]
test = asia[-id,]
```

Learning structure of the BN using training set.

In the following, the learning of the BN starts. Again, like in *assignment 1*, we use the hill-climbing algorithm to learn the structure of the Bayesian Network.

```r
# Learning structure using hill-climbing algorithm.
bn_learned_structure = hc(x = train)
plot(bn_learned_structure, main = "Learned structure of BN")
```

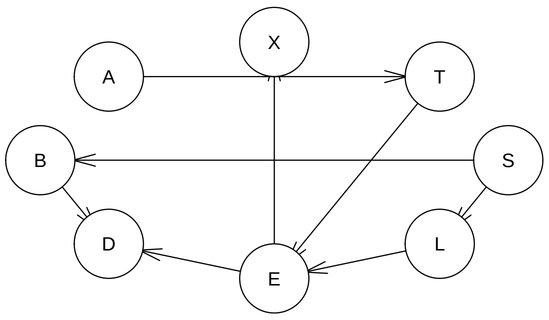
Learned structure of BN



Since the goal is also to compare the results with those of the true Asia BN, we will simultaneously perform the classification using both the learned BN and also the true BN. The structure of the true BN is given.

```
# Implementing structure of true BN.
bn_true_structure = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
plot(bn_true_structure, main = "True structure of BN")
```

True structure of BN



Learning parameters of BN using training set.

In the following, by usage of the learned/implemented BN structure, the parameters are learned. As method, *maximum likelihood* is chosen.

```
# Learning parameters using maximum likelihood estimation.
bn_learned_param = bn.fit(x = bn_learned_structure,
                          data = train,
                          method = "mle")

bn_true_param = bn.fit(x = bn_true_structure,
                       data = train,
                       method = "mle")
```

Classifying test set using learned BN.

With both obtained BN, it possible to classify the test set.

To set the evidence, I decided to use the function `setEvidence()`, since it is documented that it is "an improvement of setFinding() (and as such setFinding is obsolete). Users are recommended to use setEvidence() in the future.".

```r
# Converting object with fit parameters to gRain-object
bn_learned_param_gRain = as.grain(bn_learned_param)
bn_true_param_gRain = as.grain(bn_true_param)

# Creating a junction tree by performing Lauritzen-Spiegelhalter algorithm.
# Establishing clique potentials.
bn_learned_junc_tree = compile(bn_learned_param_gRain)
bn_true_junc_tree = compile(bn_true_param_gRain)

# Classifying test set.
  # Implementing function for classification.
  classify_test_set = function(bn_junc_tree, given_variables) {
    # Initializing vector to store predictions.
    predictions = c()
    # Looping over each row / each observation of data.
    for (row in 1:nrow(test)) {
      row_states = c()
      # Looping over all variables but "S".
      for (x in which(colnames(test) %in% given_variables)) {
        # Collecting states.
        if (test[row, x] == "yes") {
          row_states = c(row_states, "yes")
        } else {
          row_states = c(row_states, "no")
        }
      }
      # Setting evidence.
      evidence = setEvidence(object = bn_junc_tree,
                             # Setting nodes to all variables except "S".
                             nodes = given_variables,
                             states = row_states)
      # Predicting S from junction tree using evidence for current row.
      row_prediction = querygrain(object = evidence,
                                  nodes = c("S"))$S

      # Adding row prediction to collection.
      if (row_prediction["yes"] >= 0.5) {
        predictions[row] = "yes"
      } else {
        predictions[row] = "no"
      }
    }
    # Returning collection of all predictions.
    return (predictions)
  }
  # Classifying using implemented function.
  bn_learned_classification =
    classify_test_set(bn_junc_tree = bn_learned_junc_tree,
                      given_variables = c("A", "T", "L", "B", "E", "X", "D"))

  bn_true_classification =
    classify_test_set(bn_junc_tree = bn_true_junc_tree,
                      given_variables = c("A", "T", "L", "B", "E", "X", "D"))
```

Reporting confusion matrices.

For both classifications, the confusion matrix can be obtained.

```
knitr::kable(table(bn_learned_classification, test$S),
             caption = "Confusion matrix using learned structure")
```

Table 1: Confusion matrix using learned structure

	no	yes
no	322	120
yes	146	412

```
knitr::kable(table(bn_true_classification, test$S),
             caption = "Confusion matrix using true structure")
```

Table 2: Confusion matrix using true structure

	no	yes
no	322	120
yes	146	412

Assignment 3. Performing classification with BN given only observations for Markov blanket of S.

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.

Getting Markov blanket of S.

```
# Getting Markov blanket from fitted BN structures.
mb_learned = mb(x = bn_learned_param,
            node = "S")
mb_true = mb(x = bn_true_param,
             node = "S")
```

Classifying test set using learned BN given only observations only for Markov blanket of S.

```
# Classifying using implemented function.
bn_learned_classification_mb =
  classify_test_set(bn_junc_tree = bn_learned_junc_tree,
                    given_variables = mb_learned)

bn_true_classification_mb =
  classify_test_set(bn_junc_tree = bn_true_junc_tree,
                    given_variables = mb_true)
```

Reporting confusion matrices.

```
knitr::kable(table(bn_learned_classification, test$S),
             caption = "Confusion matrix using learned structure")
```

Table 3: Confusion matrix using learned structure

	no	yes
no	322	120
yes	146	412

```
knitr::kable(table(bn_true_classification, test$S),
             caption = "Confusion matrix using true structure")
```

Table 4: Confusion matrix using true structure

	no	yes
no	322	120
yes	146	412

Conclusions.

The result has not changed compared to the previous classification from *assignment 2*.

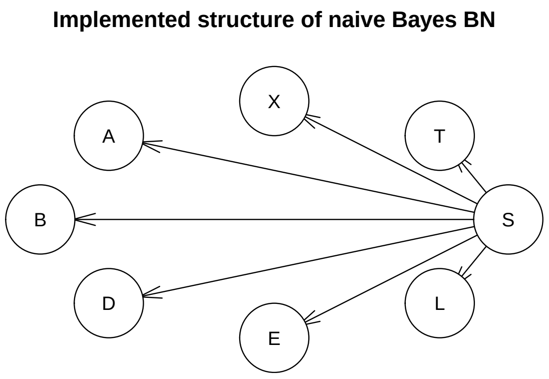
Assignment 4. Repeating assignment 2 using a naive Bayes classifier.

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.

Implementing structure for naive Bayes BN.

As the given in *assignment 2*, the function `model2network()` can be used to implement a BN strucutre. Dependencies can be implemented by usage of the "|" sign. Since a naive Bayes classifier assumes that the predictive variables are independent given the class variable (S), the BN for the naive Bayes classifier is represented by S being the parent to all other nodes.

```
bn_nBayes_structure = model2network("[S][A|S][T|S][L|S][B|S][E|S][X|S][D|S]")
plot(bn_nBayes_structure, main = "Implemented structure of naive Bayes BN")
```



Learning parameters of BN using training set. Classifying test set using naive Bayes BN.

Using the implemented naive Bayes BN structure, the fitting and classification process from *assignment 2* can be repeated.

```
# Learning parameters using maximum likelihood estimation.
bn_nBayes_param = bn.fit(x = bn_nBayes_structure,
                         data = train,
                         method = "mle")

# Converting object with fit parameters to gRain-object
bn_nBayes_param_gRain = as.grain(bn_nBayes_param)

# Creating a junction tree by performing Lauritzen-Spiegelhalter algorithm.
# Establishing clique potentials.
bn_nBayes_junc_tree = compile(bn_nBayes_param_gRain)

# Classifying using implemented function.
bn_nBayes_classification =
  classify_test_set(bn_junc_tree = bn_nBayes_junc_tree,
                    given_variables = c("A", "T", "L", "B", "E", "X", "D"))
```

Reporting confusion matrix.

The confusion matrix can be obtained.

```
knitr::kable(table(bn_nBayes_classification, test$S),
             caption = "Confusion matrix using learned structure")
```

Table 5: Confusion matrix using learned structure

	no	yes
no	349	188
yes	119	344

Assignment 5. Analysis of the obtained results.

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

After running a trained model with parameters of own choice, we obtain the confusion matrix as reported inquestion 2. The obtained confusion matrix from the trained model in question 2 is exactly the same as theactual model provided in the lab. In plain english this means, our model performs equally well as the truemodel. Thus we could say our model is trained well.Subsequently, we run a model only on the Markov Blanket of node "S". This model again returns exactly thesame confusion matrix. We believe that the reason that models from question 2 and 3 return the same resultsbecause according to the Markov Blanket, node "S" is determined by the markov blanket nodes. Therefore amore complicated model adds no explanatory power. For question 4, the naive Bayes model, we get a higher error term. Thus the model is worse in predicting.This comes from the fact that the model assumes independence amongst the all other nodes. In fact this is ahighly naive assumption, that's why it's called Naive Bayes. The Naive Bayes is thus different from the truemodel.