Advanced Machine Learning - Lab 01

G. Hari Prasath (hargo729), Julius Kittler (julki029), Hector Plata (hecpl268) and Maximilian Pfundstein (maxpf364)

2019-09-17

Contents

1 Hill Climbing

Task: 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`.

Answer: First we will load the data set and have a look at it.

```
head(asia)
```

```
##    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
## 6 no yes  no no  no  no  no yes
```

To show that different runs of the hill climbing algorithm yield in different results, we will use two different approaches.

1. We will create one random graph, and then run the `hc()` function multiple times, also utilising the different options offered.
2. We will use different random graphs which should give us different results.

The hill climbing algorithm is basically finding the next global maximum (or minimum), which is a really easy approach but will also not give us good results. As different measurement metrics will obviously give us different results, we will not further investigate this option.

1.1 Same Graph

So we will first create a random graph using the dataset.

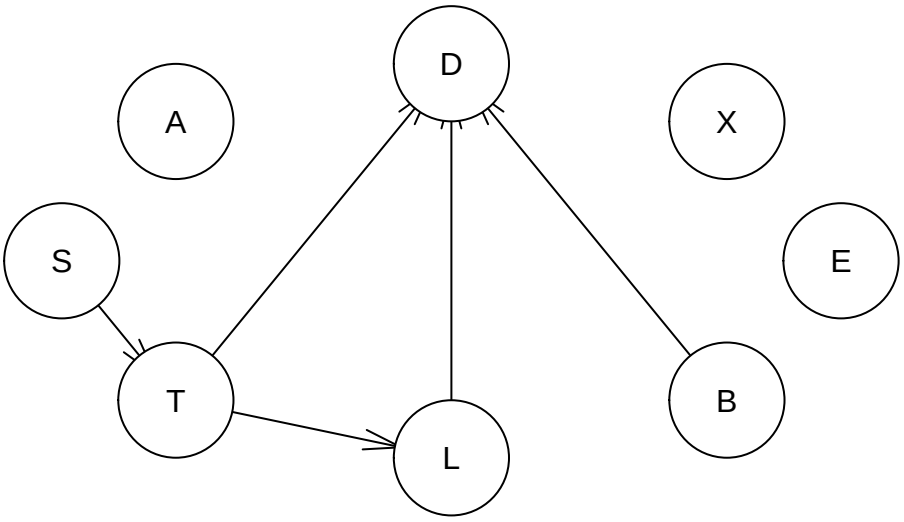
```
bayes_net1 = random.graph(colnames(asia))
bayes_net1
```

```
##
##   Random/Generated Bayesian network
##
##   model:
##    [A][S][B][E][X][T|S][L|T][D|T:L:B]
##   nodes:                                 8
##   arcs:                                  5
##     undirected arcs:                     0
##     directed arcs:                       5
##   average markov blanket size:           1.75
##   average neighbourhood size:            1.25
##   average branching factor:              0.62
##
##   generation algorithm:                  Full Ordering
##   arc sampling probability:              0.2857143
```

```
score(bayes_net1, asia)
```

```
## [1] -13931.42
```

```
plot(bayes_net1)
```



Now we will use the hill climbing algorithm to optimise our random graph

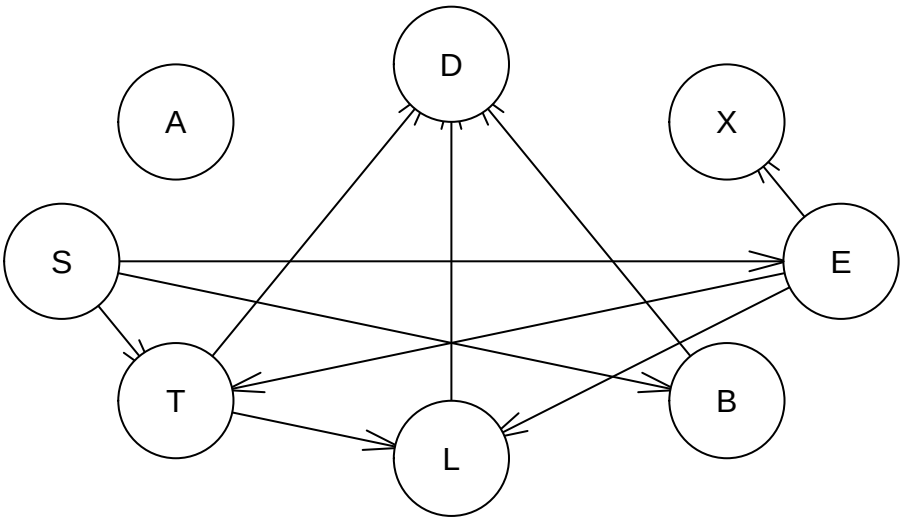
```
bayes_net1_hc = hc(asia, start = bayes_net1)
bayes_net1_hc
```

```
##
##   Bayesian network learned via Score-based methods
##
##   model:
##    [A][S][B|S][E|S][T|S:E][X|E][L|T:E][D|T:L:B]
##   nodes:                                 8
##   arcs:                                  10
##     undirected arcs:                     0
##     directed arcs:                       10
##   average markov blanket size:           3.00
##   average neighbourhood size:            2.50
##   average branching factor:              1.25
##
##   learning algorithm:                    Hill-Climbing
##   score:                                 BIC (disc.)
##   penalization coefficient:              4.258597
##   tests used in the learning procedure:  81
##   optimized:                             TRUE
```

```
score(bayes_net1_hc, asia)
```

```
## [1] -11133.45
```

```
plot(bayes_net1_hc)
```



We observe that the network has changed and has a better score. Another call.

```
bayes_net1_hc = hc(asia, start = bayes_net1_hc)

score(bayes_net1_hc, asia)
```

```
## [1] -11133.45
```

This time the score has not been updated, as we're *stuck* in the the same local optima.

What we can do is use the parameters `restart` and and `perturb`. According to the documentation and the source code the parameters `perturb` specifies how many edges or nodes (in this terminology *arcs*) to change after each restart, the default is one. The parameter `restart` specifies how often to do that. This usually results in slightly better graphs. If `perturb = 1` all combinations are quickly investigated and a different for `restart` does not really matter that much any more:

```
bayes_net1_hc_restart_low = hc(asia, start = bayes_net1_hc, restart = 10)
bayes_net1_hc_restart_high = hc(asia, start = bayes_net1_hc, restart = 100)
bayes_net1_hc_restart_vhigh = hc(asia, start = bayes_net1_hc, restart = 1000)

score(bayes_net1_hc_restart_low, asia)
```

```
## [1] -11129.19
```

```
score(bayes_net1_hc_restart_high, asia)
```

```
## [1] -11124.71
```

```
score(bayes_net1_hc_restart_vhigh, asia)
```

```
## [1] -11119.26
```

1.2 Different Graphs

If we use different random graphs in the beginning, the hill climbing algorithm will find different local optima.

```
bayes_net2_1_hc = hc(asia, start = random.graph(colnames(asia)))
bayes_net2_2_hc = hc(asia, start = random.graph(colnames(asia)))

score(bayes_net2_1_hc, asia)
```

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

```
score(bayes_net2_2_hc, asia)
```

```
## [1] -11127.42
```

As we can see, they are different. If we specify the `restart` parameter again, it will eventually find the same optima, at least for this case, as this is a very small problem.

```
bayes_net2_1_hc = hc(asia, start = random.graph(colnames(asia)), restart = 100)
bayes_net2_2_hc = hc(asia, start = random.graph(colnames(asia)), restart = 100)

score(bayes_net2_1_hc, asia)
```

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

```
score(bayes_net2_2_hc, asia)
```

```
## [1] -11115.56
```

2 Inference

Task: 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):

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

Answer: First we split into training and test, by sampling randomly.

```
################################################################################
# Exercise 2)
################################################################################

asia = asia %>% mutate(id = row_number())
train = asia %>% sample_frac(.8)
test = anti_join(asia, train, by = 'id')

train = select(train, -id)
test = select(test, -id)
testX = select(test, -S)
testY = test %>% select(S)

# Helper Functions

## Training the network
train_bayesian_network = function(structure = NULL,
                                  data = train,
                                  learning_algorithm = iamb,
                                  ...) {

  # Network
  if (is.null(structure)) {
    bayesian_network = learning_algorithm(data, ...)
  }
  else {
    bayesian_network = structure
  }

  # Parameters
  bayesian_network_fit = bn.fit(bayesian_network, data)
  bayesian_network_grain = compile(as.grain(bayesian_network_fit))

  return(list(bn_grain=bayesian_network_grain,
              bn_fit=bayesian_network_fit,
              bn_structure=bayesian_network))
}

## Predicting with the network
predict_bayesian_network = function(bayesian_network,
                                    testX_ = testX,
                                    testY_ = testY,
                                    markov_blanket = NULL) {

  # Parallel setup
  no_cores = detectCores()
  cl = makeCluster(no_cores)
  clusterExport(cl, list("querygrain", "setEvidence", "bayesian_network"),
                envir=environment())

  # predict for each data point of the test data
  if (!is.null(markov_blanket)) {
    # When predicting, only use the nodes from the Markov Blanket
```

```
    res = t(parApply(cl, testX, 1, FUN = function(x) {
      return(querygrain(setEvidence(bayesian_network, markov_blanket,
                                    x[markov_blanket]))$S)
    }))
  }
  else {
    # Predict using all nodes
    res = t(parApply(cl, testX_, 1, FUN = function(x) {
      return(querygrain(setEvidence(bayesian_network, names(x), x))$S)
    }))
  }

  # Classify
  pred = parApply(cl, res, 1, FUN = function(x) {
    if (x[2] > 0.5) return("yes")
    return("no")
  })

  # Factorise
  testY_factor = testY_[,1]
  pred_factor = factor(pred)

  # Call to a library to calculate interesting metrics
  confusion_matrix = confusionMatrix(pred_factor,
                                     testY_factor, mode="everything")

  stopCluster(cl)

  return(list(res=res, pred=pred, cf=confusion_matrix))
}
```

First we learn the network structure and then we fit the parameters. Afterwards we transform the object into a `grain` object. We do this by using the custom built function.

```
bn_2 = train_bayesian_network(structure = NULL,
                        data = train)
```

Using the given evidence we calculate the posterior probability for each case. We then take the posterior probability to classify the test data. As a comparison we also take the built in `predict()` function.

```
predicate_2 = predict_bayesian_network(bn_2$bn_grain)

# Metrics
predicate_2$cf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  353 148
##        yes 145 354
##
##                Accuracy : 0.707
##                  95% CI : (0.6777, 0.7351)
##     No Information Rate : 0.502
```

```
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.414
##
##  Mcnemar's Test P-Value : 0.907
##
##             Sensitivity : 0.7088
##             Specificity : 0.7052
##          Pos Pred Value : 0.7046
##          Neg Pred Value : 0.7094
##               Precision : 0.7046
##                  Recall : 0.7088
##                      F1 : 0.7067
##              Prevalence : 0.4980
##          Detection Rate : 0.3530
##    Detection Prevalence : 0.5010
##       Balanced Accuracy : 0.7070
##
##        'Positive' Class : no
##
```

```
# We will use predict to compare our own inference with the build in version
predicate_2_sol = predict(bn_2$bn_fit, "S", data = test, method="bayes-lw")

# Factorize
testY_factor = testY[,1]

confusionMatrix(predicate_2_sol, testY_factor, mode="everything")
```

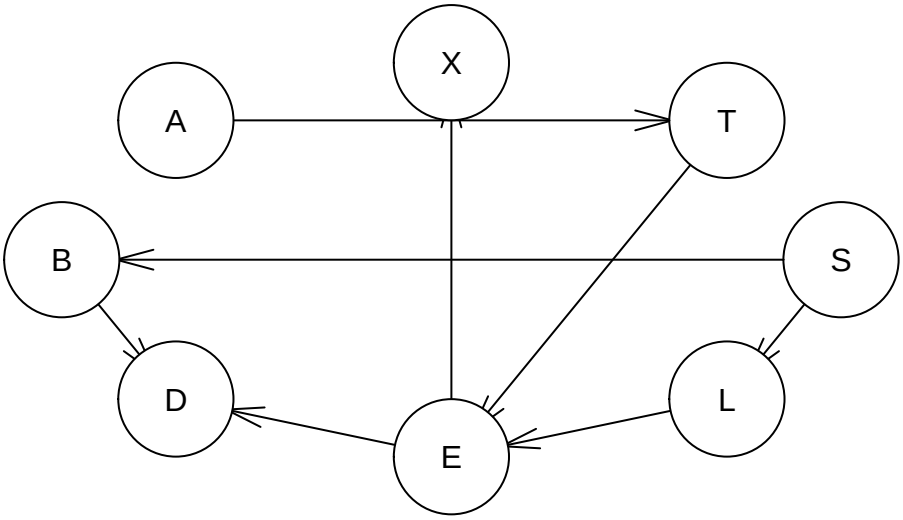
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  353 148
##        yes 145 354
##
##                Accuracy : 0.707
##                  95% CI : (0.6777, 0.7351)
##     No Information Rate : 0.502
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.414
##
##  Mcnemar's Test P-Value : 0.907
##
##             Sensitivity : 0.7088
##             Specificity : 0.7052
##          Pos Pred Value : 0.7046
##          Neg Pred Value : 0.7094
##               Precision : 0.7046
##                  Recall : 0.7088
##                      F1 : 0.7067
##              Prevalence : 0.4980
##          Detection Rate : 0.3530
##    Detection Prevalence : 0.5010
```

```
##       Balanced Accuracy : 0.7070
##
##        'Positive' Class : no
##
```

As we can see, we observe very similar results.

The true Bayesian network is given by the following:

```
# Define true network, train and use for prediction
dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
plot(dag)
```




```
bn_2_true = train_bayesian_network(dag)
predicate_2_true = predict_bayesian_network(bn_2_true$bn_grain)

predicate_2_true$cf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  347 129
##        yes 151 373
##
##                Accuracy : 0.72
##                  95% CI : (0.6911, 0.7476)
##     No Information Rate : 0.502
```

```
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.4399
##
##  Mcnemar's Test P-Value : 0.2095
##
##             Sensitivity : 0.6968
##             Specificity : 0.7430
##          Pos Pred Value : 0.7290
##          Neg Pred Value : 0.7118
##               Precision : 0.7290
##                  Recall : 0.6968
##                      F1 : 0.7125
##              Prevalence : 0.4980
##          Detection Rate : 0.3470
##    Detection Prevalence : 0.4760
##       Balanced Accuracy : 0.7199
##
##        'Positive' Class : no
##
```

We see that the true network just has a slightly better F1 score compared to the trained versions from the data.

3 Markov Blanket

Task: 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.

Answer: First we will extract the *Markov Blanket* using the function `mb()`. The *Markov Blanket* are all nodes that shield the node in question from the rest of the network. Then we apply the same procedure as before and look at the results.

```
################################################################################
# Exercise 3)
################################################################################

predicate_3_mb =
  predict_bayesian_network(bn_2$bn_grain,
                           markov_blanket = mb(bn_2$bn_structure, "S"))

predicate_3_mb$cf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  353 148
##        yes 145 354
##
##                Accuracy : 0.707
##                  95% CI : (0.6777, 0.7351)
```

```
##     No Information Rate : 0.502
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.414
##
##  Mcnemar's Test P-Value : 0.907
##
##             Sensitivity : 0.7088
##             Specificity : 0.7052
##          Pos Pred Value : 0.7046
##          Neg Pred Value : 0.7094
##               Precision : 0.7046
##                  Recall : 0.7088
##                      F1 : 0.7067
##              Prevalence : 0.4980
##          Detection Rate : 0.3530
##    Detection Prevalence : 0.5010
##       Balanced Accuracy : 0.7070
##
##        'Positive' Class : no
##
```

As we can see the F1 score (and so the results) remain unchanged when only considering the markov blanket.

4 Naive Bayes

Task: 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.

Answer: The Naive Bayes classifier assumes independence between all the predictive variables. Therefore the classifier is given by

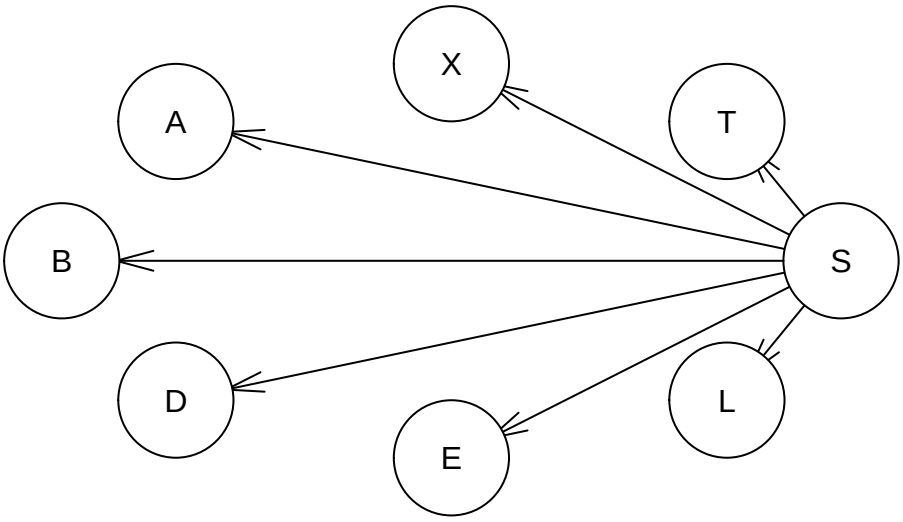
$$p(S|x_i) \propto p(S) \prod_{i=1}^{N} p(x_i|S)$$

x_i are the features, therefore our Bayesian network structure is based on the following formula:

$$p(S) \propto p(S)p(A|S)p(T|S)p(L|S)p(B|S)p(E|S)p(X|S)p(D|S)$$

```
################################################################################
# Exercise 4)
################################################################################

# Structure
bn_naive_bayes = model2network("[S][A|S][T|S][L|S][B|S][E|S][X|S][D|S]")
plot(bn_naive_bayes)
```



As we can see, according to our model there exists no dependence between the features. Let's train the parameters and look at the results.

```
bn_4_naive_bayes = train_bayesian_network(structure = bn_naive_bayes,
                      data = train,
                      learning_algorithm = iamb)

predicate_4_naive_bayes = predict_bayesian_network(bn_4_naive_bayes$bn_grain)

predicate_4_naive_bayes$cf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  375 180
##        yes 123 322
##
##                Accuracy : 0.697
##                  95% CI : (0.6675, 0.7254)
##     No Information Rate : 0.502
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.3943
##
##  Mcnemar's Test P-Value : 0.001295
##
```

```
##             Sensitivity : 0.7530
##             Specificity : 0.6414
##          Pos Pred Value : 0.6757
##          Neg Pred Value : 0.7236
##               Precision : 0.6757
##                  Recall : 0.7530
##                      F1 : 0.7123
##              Prevalence : 0.4980
##          Detection Rate : 0.3750
##    Detection Prevalence : 0.5550
##       Balanced Accuracy : 0.6972
##
##        'Positive' Class : no
##
```

5 Explanation of Different Results

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

Answer: As the observations of the Markov Blanket make S independent from all other variables, we should and we do observe the exact same result in 2) and 3) (for the same trained network). For the Naive Bayes classifier we see, that we get a slightly worse F1 score, which is to be expected, as our assumption does not hold, when we compare with the true network given. Despite this fact, the Naive Base classifier does quite well.

6 Source Code

```
knitr::opts_chunk$set(echo = TRUE)
library(bnlearn)
library(tibble)
library(dplyr)
library(gRain)
library(caret)
library(parallel)
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install("RBGL")
set.seed(42)

################################################################################
# Exercise 1)
################################################################################

as_tibble("asia")


head(asia)


bayes_net1 = random.graph(colnames(asia))
bayes_net1
```

```
score(bayes_net1, asia)
plot(bayes_net1)


bayes_net1_hc = hc(asia, start = bayes_net1)
bayes_net1_hc
score(bayes_net1_hc, asia)
plot(bayes_net1_hc)

bayes_net1_hc = hc(asia, start = bayes_net1_hc)

score(bayes_net1_hc, asia)


bayes_net1_hc_restart_low = hc(asia, start = bayes_net1_hc, restart = 10)
bayes_net1_hc_restart_high = hc(asia, start = bayes_net1_hc, restart = 100)
bayes_net1_hc_restart_vhigh = hc(asia, start = bayes_net1_hc, restart = 1000)

score(bayes_net1_hc_restart_low, asia)
score(bayes_net1_hc_restart_high, asia)
score(bayes_net1_hc_restart_vhigh, asia)


bayes_net2_1_hc = hc(asia, start = random.graph(colnames(asia)))
bayes_net2_2_hc = hc(asia, start = random.graph(colnames(asia)))

score(bayes_net2_1_hc, asia)
score(bayes_net2_2_hc, asia)


bayes_net2_1_hc = hc(asia, start = random.graph(colnames(asia)), restart = 100)
bayes_net2_2_hc = hc(asia, start = random.graph(colnames(asia)), restart = 100)

score(bayes_net2_1_hc, asia)
score(bayes_net2_2_hc, asia)


################################################################################
# Exercise 2)
################################################################################

asia = asia %>% mutate(id = row_number())
train = asia %>% sample_frac(.8)
test = anti_join(asia, train, by = 'id')

train = select(train, -id)
test = select(test, -id)
testX = select(test, -S)
testY = test %>% select(S)

# Helper Functions

## Training the network
```

```
train_bayesian_network = function(structure = NULL,
                                  data = train,
                                  learning_algorithm = iamb,
                                  ...) {

  # Network
  if (is.null(structure)) {
    bayesian_network = learning_algorithm(data, ...)
  }
  else {
    bayesian_network = structure
  }

  # Parameters
  bayesian_network_fit = bn.fit(bayesian_network, data)
  bayesian_network_grain = compile(as.grain(bayesian_network_fit))

  return(list(bn_grain=bayesian_network_grain,
              bn_fit=bayesian_network_fit,
              bn_structure=bayesian_network))
}

## Predicting with the network
predict_bayesian_network = function(bayesian_network,
                                    testX_ = testX,
                                    testY_ = testY,
                                    markov_blanket = NULL) {

  # Parallel setup
  no_cores = detectCores()
  cl = makeCluster(no_cores)
  clusterExport(cl, list("querygrain", "setEvidence", "bayesian_network"),
                envir=environment())

  # predict for each data point of the test data
  if (!is.null(markov_blanket)) {
    # When predicting, only use the nodes from the Markov Blanket
    res = t(parApply(cl, testX, 1, FUN = function(x) {
      return(querygrain(setEvidence(bayesian_network, markov_blanket,
                                    x[markov_blanket]))$S)
    }))
  }
  else {
    # Predict using all nodes
    res = t(parApply(cl, testX_, 1, FUN = function(x) {
      return(querygrain(setEvidence(bayesian_network, names(x), x))$S)
    }))
  }

  # Classify
  pred = parApply(cl, res, 1, FUN = function(x) {
    if (x[2] > 0.5) return("yes")
    return("no")
```

```
  })

  # Factorise
  testY_factor = testY_[,1]
  pred_factor = factor(pred)

  # Call to a library to calculate interesting metrics
  confusion_matrix = confusionMatrix(pred_factor,
                                     testY_factor, mode="everything")

  stopCluster(cl)

  return(list(res=res, pred=pred, cf=confusion_matrix))
}


bn_2 = train_bayesian_network(structure = NULL,
                        data = train)


predicate_2 = predict_bayesian_network(bn_2$bn_grain)

# Metrics
predicate_2$cf


# We will use predict to compare our own inference with the build in version
predicate_2_sol = predict(bn_2$bn_fit, "S", data = test, method="bayes-lw")

# Factorize
testY_factor = testY[,1]

confusionMatrix(predicate_2_sol, testY_factor, mode="everything")


# Define true network, train and use for prediction
dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
plot(dag)


bn_2_true = train_bayesian_network(dag)
predicate_2_true = predict_bayesian_network(bn_2_true$bn_grain)

predicate_2_true$cf


##############################################################################
# Exercise 3)
##############################################################################

predicate_3_mb =
  predict_bayesian_network(bn_2$bn_grain,
                           markov_blanket = mb(bn_2$bn_structure, "S"))
```

```
predicate_3_mb$cf


##############################################################################################
# Exercise 4)
##############################################################################################

# Structure
bn_naive_bayes = model2network("[S][A|S][T|S][L|S][B|S][E|S][X|S][D|S]")
plot(bn_naive_bayes)


bn_4_naive_bayes = train_bayesian_network(structure = bn_naive_bayes,
                        data = train,
                        learning_algorithm = iamb)

predicate_4_naive_bayes = predict_bayesian_network(bn_4_naive_bayes$bn_grain)

predicate_4_naive_bayes$cf
```