732A96: Advanced Machine Learning - Computer Lab 1

Hariprasath Govindarajan (hargo729)

September 17, 2019

Contents

Question 1: Hill-Climbing Algorithm

Question: 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:

We load the `asia` dataset which will be used for this lab. The dataset looks as follows:

```r
data("asia")
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
```

In order to check if multiple runs of the hill climbing algorithm return non-equivalent Bayesian Network structures, we will run the `hc()` function from the `bnlearn` package using different combinations of parameters - `start` and `restarts`. The `start` parameter is used to specify the initial graph for the hill climbing algorithm to start searching and by default this is an empty graph. The `restarts` parameter is used to specify the number of restarts to use in the hill climbing algorithm. Each restart involves a perturbation which leads to a new starting graph for the restarted search for the maximum.

Firstly, we generate a random graph having the same nodes as in the data. We run the `hc()` function with this random graph as the start and with 0 restarts. Then, we run the `hc()` function again with the same parameters (`start` and `restarts`) to check if there is any randomness present in the algorithm. Then, we run the `hc()` function again but this time with `restarts=100`. Then we repeat this process with a different random graph as start.

```r
# Random graph 1
rnd_graph_1 = random.graph(colnames(asia))

# restart = 0
hc_res_1 = hc(asia, start = rnd_graph_1, restart = 0)
hc_res_2 = hc(asia, start = rnd_graph_1, restart = 0)

# restart = 100
hc_res_3 = hc(asia, start = rnd_graph_1, restart = 100)

# restart = 1000
hc_res_4 = hc(asia, start = rnd_graph_1, restart = 1000)
```

Trial 1: start = random_graph_1, restarts = 0

```
## Score =  -11141.71
```

Trial 2: start = random_graph_1, restarts = 0

```
## Score =  -11141.71
```

```
## Results of trial 1 and 2 are equivalent:  TRUE
```

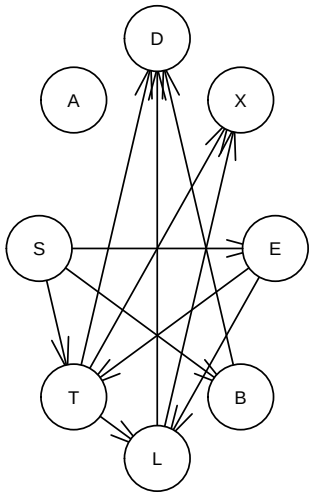
Trial 3: start = random_graph_1, restarts = 100

```
## Score =  -11127.52
```

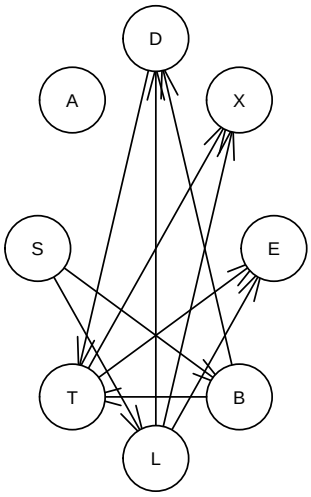
Trial 4: start = random_graph_1, restarts = 1000

```
## Score =  -11119.26
```

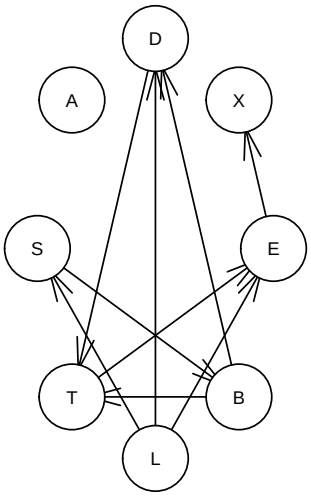
Trial 1&2 Result



Trial 3 Result



Trial 4 Result



Now, we generate a 2nd random graph for further trials.

```
# Random graph 2
rnd_graph_2 = random.graph(colnames(asia))

# restart = 0
hc_res_5 = hc(asia, start = rnd_graph_2, restart = 0)

# restart = 100
hc_res_6 = hc(asia, start = rnd_graph_2, restart = 100)

# restart = 1000
hc_res_7 = hc(asia, start = rnd_graph_2, restart = 1000)
```

Trial 5: start = random_graph_2, restarts = 0

```
## Score =  -11154.06
```

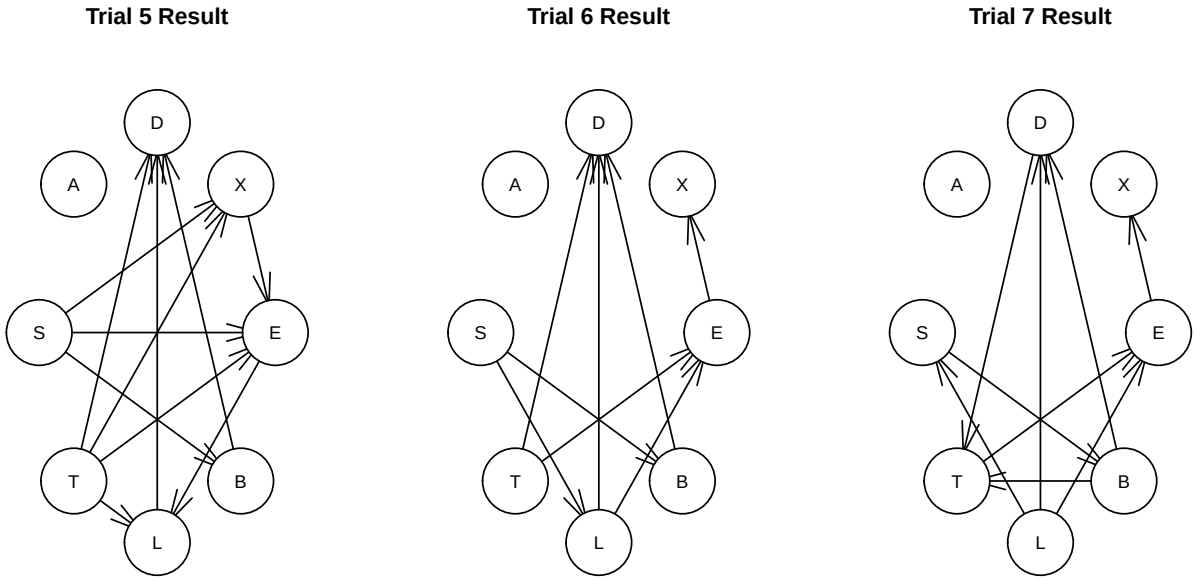
Trial 6: start = random_graph_2, restarts = 100

```
## Score =  -11120.65
```

Trial 7: start = random_graph_2, restarts = 1000

```
## Score =  -11119.26
```

```
## Results of trial 4 and 7 are equivalent:  TRUE
```



Observations:

We can see that just running the same function multiple times with zero restarts and the same starting graph produces the same resulting networks. It finds the nearest occuring maximum from the starting graph for the scoring metric. By increasing the number of restarts, we are able to get a network with a higher score. After 1000 restarts, we get the same result from using 2 different random starting graphs.

Explanations:

The simple hill climbing algorithm without any restarts finds the nearest occuring maximum which could be a local maximum. So, the bayesian network found corresponding to the maximum varies based on the start parameter where we use a random graph. When we use 0 restarts there is no randomness and the algorithm always produces the same resulting network. By specifying multiple restarts, the hill climbing restarts from different start graphs by introducing perturbations in the graph (which can be specified using `perturbs` parameter, by default `perturbs=1`). Since the algorithm searches for the maximum with multiple different starting positions, it has a higher chance to find the global maxima. So, when we specify a higher number of restarts, the algorithm finds a bayesian network which fits the data better and hence, has a higher score.

Question 2: Bayesian Network Model Learning

Question: 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:

We split the `asia` dataset into train (80%) and test (20%) sets. We will use the variable "S" as the target and all other variables as predictors.

```
# Split into train/test sets
set.seed(12345)
n = nrow(asia)
id = sample(1:n, floor(n*0.8))
train = asia[id, ]
test = asia[-id, ]

# Split into predictor X and target Y
# 2nd variable "S" is target in this case
train_x = train[ , -2]
test_x = test[ , -2]
test_y = test[ , 2]
```

```
# Variable names in X
x_names = names(test_x)
```

The function `bn_predict()` can train a bayesian network using a given training data and predict the probabilities for a target node in the network for the test data. The network structure can be explicitly specified and will be learned if not specified. The structure is learned using the Incremental Association (`iamb`) algorithm by default. It is possible to specify whether the probabilities should be predicted using only the Markov Blanket of the target node or all the nodes in the graph.

```
# Function to predict values for a node using a bayesian network
bn_predict = function(train, test, pred_node, bn_struct=NULL,
                      str_learn=iamb, mb = FALSE){
  # Remove target node from predictor variables
  x_names = names(test_x)
  x_names = x_names[x_names != pred_node]
  test_x = test[, x_names]

  # Learn structure if not specified
  if(is.null(bn_struct)){
    bn_struct = str_learn(train)
  }

  # Parameter learning
  bn_mod = bn.fit(bn_struct, train)
  bn_mod_grain = compile(as.grain(bn_mod))

  if(mb){
    # Get markov blanket of target node
    bn_mb = mb(bn_mod, pred_node)
    bn_mb = bn_mb[bn_mb != pred_node]

    # Get test data for markov blanket
    test_x = test_x[, bn_mb]
    x_names = names(test_x)
  }

  # Query probabilities from learned BN
  probs = t(sapply(1:nrow(test_x),
                   function(i) {
                     querygrain(
                       setEvidence(bn_mod_grain, nodes=x_names,
                                   states=fact2chr(test_x[i,])))[[pred_node]]
                     }))

  res = list(bn_struct = bn_struct, bn_fit = bn_mod,
```

```
              bn_fit_grain = bn_mod_grain, pred_probs = probs)

  return(res)
}
```

Inference using learned network structure

In this question, we will learn the structure and parameters of the network using the 80% training data. For the test data, we infer the probabilities of the target node "S" using all the other nodes. We classify the node "S" to the class with higher probability.

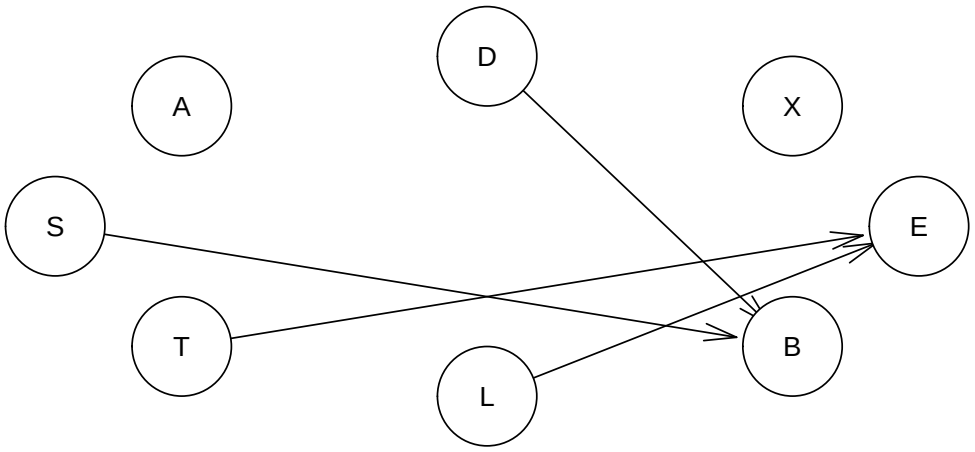
```
# Train BN and predict probabilities
bn_res = bn_predict(train, test, "S")

# Classify to most likely class
S_pred = ifelse(bn_res$pred_probs[,"yes"] > 0.5, "yes", "no")

# Performance metrics
cm_q2 = confusionMatrix(as.factor(S_pred), test_y)
f1_q2 = cm_q2$byClass["F1"]
acc_q2 = cm_q2$overall["Accuracy"]
```

The learned Bayesian network has the following structure:

```
## Bayesian Network Model:
##  [A][S][T][L][X][D][B|S:D][E|T:L]
```



The following confusion matrix shows the true classes in the rows and the predicted classes in the columns.

	no	yes
no	330	138
yes	140	392

```
##        F1
## 0.7036247
```

```
## Accuracy
##    0.722
```

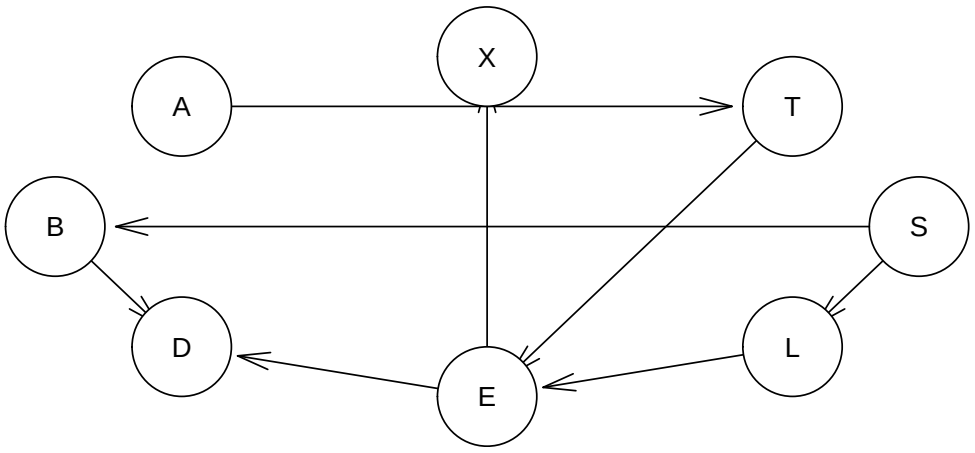
Inference using true network structure

Now, we are given the true structure of this network. So, we train a bayesian network to only learn the parameters. Based on this Bayesian network, we classify the node "S" again.

```
# True BN structure
dag_true = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
```

The true Bayesian network looks as follows:

```
## Bayesian Network Model:
##  [A][S][B|S][L|S][T|A][E|L:T][D|B:E][X|E]
```



```r
# Train BN and predict probabilities
bn_true_res = bn_predict(train, test, "S", bn_struct = dag_true)

# Classify to most likely class
S_true_pred = ifelse(bn_true_res$pred_probs[,"yes"] > 0.5, "yes", "no")
```

The following confusion matrix shows the true classes in the rows and the classes predicted by using the true network in the columns.

	no	yes
no	322	146
yes	120	412

```
##        F1
## 0.7076923
```

```
## Accuracy
##    0.734
```

Comparison of inferences using true and learned network structures

The following confusion matrix shows the classes predicted by using the true network in the rows and the classes predicted earlier by learning both the structure and parameters in the columns. We see that the predictions are mostly similar between the learned structure and the true structure. Also, we see that the F1 score and accuracy are slightly better for the Bayesian network learned using the true structure. This makes sense as the structure learned using `iamb` is different from the true structure and leads to slight differences in predictions.

	no	yes
no	442	0
yes	28	530

Question 3: Inference using Markov Blanket

Question: 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:

Now, we want to classify the node "S" given observations only for the Markov blanket of "S".

```
# Train BN and predict probabilities
bn_mb_res = bn_predict(train, test, "S", mb = T)

# Classify to most likely class
S_mb_pred = ifelse(bn_mb_res$pred_probs[,"yes"] > 0.5, "yes", "no")

# Performance metrics
cm_q3 = confusionMatrix(as.factor(S_mb_pred), test_y)
f1_q3 = cm_q3$byClass["F1"]
acc_q3 = cm_q3$overall["Accuracy"]
```

The following confusion matrix shows the true classes in the rows and the classes predicted given the observations for the Markov blanket in the columns.

	no	yes
no	330	138
yes	140	392

```
##        F1
## 0.7036247

## Accuracy
##    0.722
```

Question 4: Naive Bayes as a Bayesian Network

Question: 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.

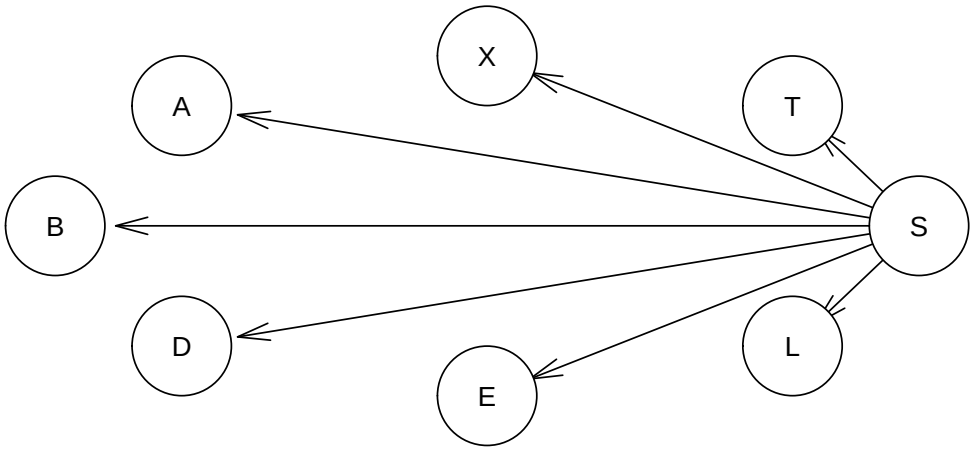
Hint: Check `http://www.bnlearn.com/examples/dag/` to see how to create a BN by hand.

Answer:

Now, we want to model and use a Naive Bayes classifier as a Bayesian Network. The Naive Bayes approach assumes that the predictor variables are conditionally independent of each other given the target variable. This can be modeled using the following network.

```
# Construct Naive Bayes network
bn_naive_net = model2network("[S][A|S][T|S][L|S][B|S][E|S][X|S][D|S]")
```

```
## Bayesian Network Model:
##  [S][A|S][B|S][D|S][E|S][L|S][T|S][X|S]
```



We train the Naive Bayes network to learn the parameters and obtain the classifications for "S" in the test data.

```
# Train BN and predict probabilities
bn_naive_res = bn_predict(train, test, "S", bn_struct = bn_naive_net)

# Classify to most likely class
S_naive_pred = ifelse(bn_naive_res$pred_probs[,"yes"] > 0.5, "yes", "no")

# Performance metrics
cm_q4 = confusionMatrix(as.factor(S_naive_pred), test_y)
f1_q4 = cm_q4$byClass["F1"]
acc_q4 = cm_q4$overall["Accuracy"]
```

The following confusion matrix shows the true classes in the rows and the classes predicted using the Naive Bayes network in the columns.

	no	yes
no	349	119
yes	188	344

```
##        F1
## 0.6945274

## Accuracy
##    0.693
```

Question 5: Analysis and Comparison of Results

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

Answer:

Here, we compare the results of questions 2-4.

```
## Results 2 and 3 are similar:  TRUE

## Results 2 and 4 are similar:  129 string mismatches
```

The following confusion matrix shows the classes predicted by learning the structure and parameters in the rows and the classes predicted using the Naive Bayes network in the columns.

	no	yes
no	439	31
yes	98	432

We see that the results from questions 2 and 3 are exactly similar but the result of question 4 is a bit different. The following table shows the F1 scores obtained using each approach.

Question	F1 Score	Accuracy
Question 2	0.7036247	0.722
Question 3	0.7036247	0.722
Question 4	0.6945274	0.693

The target node is independent of the nodes outside the Markov Blanket conditional on the variables in the Markov Blanket. So, predicting using only the variable values in the Markov Blanket gives the same result as using all the variable values. So, we get the same result in questions 2 and 3. In question 4, we make a strong assumption about conditional independence of predictor variables which leads to a slight reduction in the accuracy and F1 score of the model. We also assume that the target variable depends on all the predictor variables. So, we are ignoring some dependences and independences in the graph and this leads to the slight decrease in the model performance for Naive Bayes compared to the Bayesian networks.

Appendix

```r
# Set up general options

knitr::opts_chunk$set(echo = FALSE, warning = FALSE, message = FALSE, fig.width=8)

set.seed(123456)

library(ggplot2)
options(kableExtra.latex.load_packages = FALSE)
library(kableExtra)
library(caret)
library(gRain)
library(bnlearn)

options(scipen=999)


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

data("asia")
head(asia)

# Random graph 1
rnd_graph_1 = random.graph(colnames(asia))

# restart = 0
hc_res_1 = hc(asia, start = rnd_graph_1, restart = 0)
hc_res_2 = hc(asia, start = rnd_graph_1, restart = 0)

# restart = 100
hc_res_3 = hc(asia, start = rnd_graph_1, restart = 100)

# restart = 1000
hc_res_4 = hc(asia, start = rnd_graph_1, restart = 1000)

cat("Score = ", score(hc_res_1, asia))

cat("Score = ", score(hc_res_2, asia))
cat("Results of trial 1 and 2 are equivalent: ", all.equal(hc_res_1, hc_res_2))

cat("Score = ", score(hc_res_3, asia))

cat("Score = ", score(hc_res_4, asia))
```

```r
par(mfrow = c(1, 3))
plot(hc_res_1, main = "Trial 1&2 Result")
plot(hc_res_3, main = "Trial 3 Result")
plot(hc_res_4, main = "Trial 4 Result")


# Random graph 2
rnd_graph_2 = random.graph(colnames(asia))

# restart = 0
hc_res_5 = hc(asia, start = rnd_graph_2, restart = 0)

# restart = 100
hc_res_6 = hc(asia, start = rnd_graph_2, restart = 100)

# restart = 1000
hc_res_7 = hc(asia, start = rnd_graph_2, restart = 1000)

cat("Score = ", score(hc_res_5, asia))

cat("Score = ", score(hc_res_6, asia))

cat("Score = ", score(hc_res_7, asia))
cat("Results of trial 4 and 7 are equivalent: ", all.equal(hc_res_4, hc_res_7))

par(mfrow=c(1,3))
plot(hc_res_5, main = "Trial 5 Result")
plot(hc_res_6, main = "Trial 6 Result")
plot(hc_res_7, main = "Trial 7 Result")


# ------------------------------------------------------------------------
# Question 2
# ------------------------------------------------------------------------

# Split into train/test sets
set.seed(12345)
n = nrow(asia)
id = sample(1:n, floor(n*0.8))
train = asia[id, ]
test = asia[-id, ]

# Split into predictor X and target Y
# 2nd variable "S" is target in this case
train_x = train[ , -2]
test_x = test[ , -2]
```

```
test_y = test[ , 2]

# Variable names in X
x_names = names(test_x)

# Function to convert a row of factor values to a character vector
fact2chr = function(x, levels = c("no", "yes")){
  x_chr = sapply(x, as.character)
  names(x_chr) = c()

  return(x_chr)
}

# Function to predict values for a node using a bayesian network
bn_predict = function(train, test, pred_node, bn_struct=NULL,
                      str_learn=iamb, mb = FALSE){
  # Remove target node from predictor variables
  x_names = names(test_x)
  x_names = x_names[x_names != pred_node]
  test_x = test[, x_names]

  # Learn structure if not specified
  if(is.null(bn_struct)){
    bn_struct = str_learn(train)
  }

  # Parameter learning
  bn_mod = bn.fit(bn_struct, train)
  bn_mod_grain = compile(as.grain(bn_mod))

  if(mb){
    # Get markov blanket of target node
    bn_mb = mb(bn_mod, pred_node)
    bn_mb = bn_mb[bn_mb != pred_node]

    # Get test data for markov blanket
    test_x = test_x[, bn_mb]
    x_names = names(test_x)
  }

  # Query probabilities from learned BN
  probs = t(sapply(1:nrow(test_x),
                   function(i) {
                     querygrain(
                       setEvidence(bn_mod_grain, nodes=x_names,
                                   states=fact2chr(test_x[i,])))[[pred_node]]
```

```
                    }))

  res = list(bn_struct = bn_struct, bn_fit = bn_mod,
             bn_fit_grain = bn_mod_grain, pred_probs = probs)

  return(res)
}

# Train BN and predict probabilities
bn_res = bn_predict(train, test, "S")

# Classify to most likely class
S_pred = ifelse(bn_res$pred_probs[,"yes"] > 0.5, "yes", "no")

# Performance metrics
cm_q2 = confusionMatrix(as.factor(S_pred), test_y)
f1_q2 = cm_q2$byClass["F1"]
acc_q2 = cm_q2$overall["Accuracy"]

cat("Bayesian Network Model: \n", as.character(bn_res$bn_struct))
plot(bn_res$bn_struct)

kable(table(as.character(test_y), S_pred))
print(f1_q2)
print(acc_q2)

# True BN structure
dag_true = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")

cat("Bayesian Network Model: \n", as.character(dag_true))
plot(dag_true)

# Train BN and predict probabilities
bn_true_res = bn_predict(train, test, "S", bn_struct = dag_true)

# Classify to most likely class
S_true_pred = ifelse(bn_true_res$pred_probs[,"yes"] > 0.5, "yes", "no")

kable(table(as.character(test_y), S_true_pred))
print(confusionMatrix(as.factor(S_true_pred), test_y)$byClass["F1"])
print(confusionMatrix(as.factor(S_true_pred), test_y)$overall["Accuracy"])

kable(table(S_true_pred, S_pred))


# ------------------------------------------------------------------------
```

```r
# Question 3
# ------------------------------------------------------------------------

# Train BN and predict probabilities
bn_mb_res = bn_predict(train, test, "S", mb = T)

# Classify to most likely class
S_mb_pred = ifelse(bn_mb_res$pred_probs[,"yes"] > 0.5, "yes", "no")

# Performance metrics
cm_q3 = confusionMatrix(as.factor(S_mb_pred), test_y)
f1_q3 = cm_q3$byClass["F1"]
acc_q3 = cm_q3$overall["Accuracy"]

kable(table(as.character(test_y), S_mb_pred))
print(f1_q3)
print(acc_q3)


# ------------------------------------------------------------------------
# Question 4
# ------------------------------------------------------------------------

# Construct Naive Bayes network
bn_naive_net = model2network("[S][A|S][T|S][L|S][B|S][E|S][X|S][D|S]")

cat("Bayesian Network Model: \n", as.character(bn_naive_net))
plot(bn_naive_net)


# Train BN and predict probabilities
bn_naive_res = bn_predict(train, test, "S", bn_struct = bn_naive_net)

# Classify to most likely class
S_naive_pred = ifelse(bn_naive_res$pred_probs[,"yes"] > 0.5, "yes", "no")

# Performance metrics
cm_q4 = confusionMatrix(as.factor(S_naive_pred), test_y)
f1_q4 = cm_q4$byClass["F1"]
acc_q4 = cm_q4$overall["Accuracy"]

kable(table(as.character(test_y), S_naive_pred))
print(f1_q4)
print(acc_q4)

cat("Results 2 and 3 are similar: ", all.equal(S_pred, S_mb_pred))
```

```r
cat("Results 2 and 4 are similar: ", all.equal(S_pred, S_naive_pred))

kable(table(S_pred, S_naive_pred))
f1_scores = data.frame(question = c("Question 2", "Question 3", "Question 4"),
           f1 = c(f1_q2, f1_q3, f1_q4),
           accuracy = c(acc_q2, acc_q3, acc_q4))

kable(f1_scores, col.names = c("Question", "F1 Score", "Accuracy")) %>%
  kable_styling(latex_option = "striped") %>% row_spec(0, bold = TRUE)
```