David Bjorelind lab1

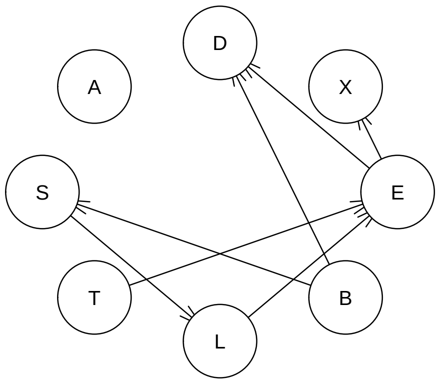
Lab 1: Graphical Models

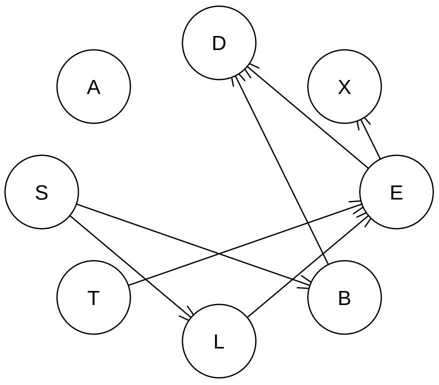
David Björelind, davbj395

This is my personal lab report for lab 1

1) Hill-climbing algorithm

```r
data('asia')
niter = 100;
networks = list()
networks[[1]] = hc(asia, restart = 5, optimized = FALSE)

for (i in 2:niter){
  hillclimb = hc(asia, restart = 5)
  networks[[i]] = hillclimb
  if (all.equal(networks[[i]], networks[[i-1]]) != TRUE){
    # Plotting graphs that differ from each other
    plot(networks[[i]])
    plot(networks[[i-1]])
    break()
  }
}
```





I run the algorithm until I generate two different networks. The differences in the graphs are different directions of some arcs.

The reason why Hill-climbing algorithm can produce different networks is because it can get stuck in a local maximum. In each step the algorithm, it proceeds if the score is improved, which can be done in many different ways. An arc can be added/removed/flipped in each iteration. When an arc is flipped, HC algorithm produces the same score,

2) Learning a Bayesian Network

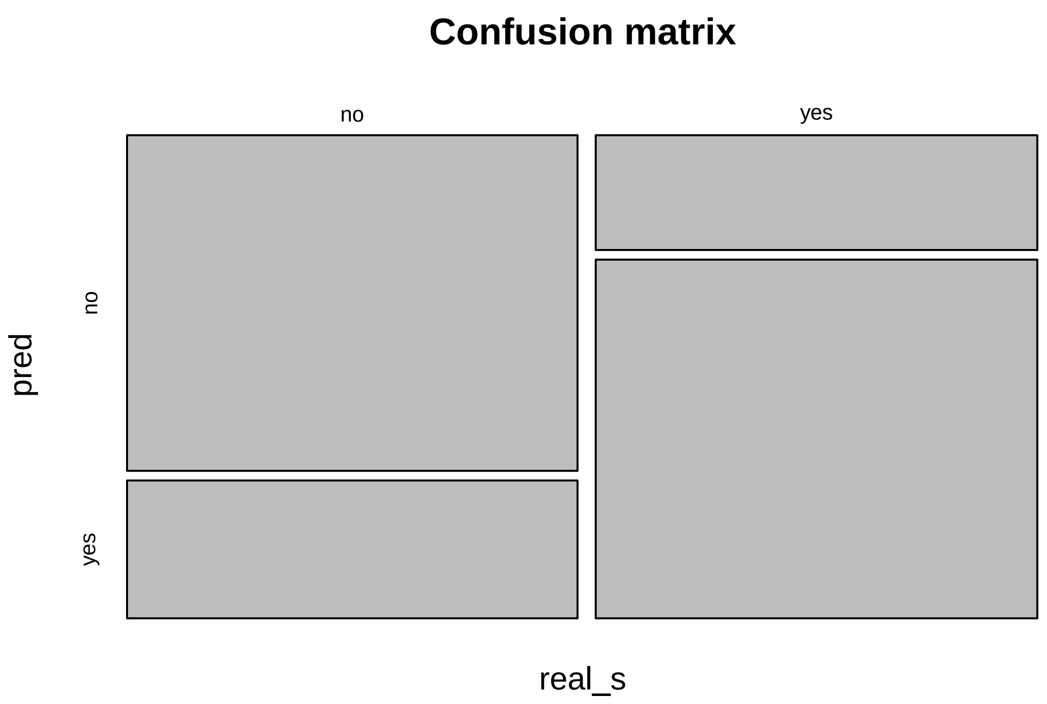
```
# Getting 80% of data set
train = asia[(1:floor(0.8*length(asia[,1]))),]
test = asia[(floor(0.8*length(asia[,1])+1):length(asia[,1])),]
real_graph = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
real_s = test$S

# Getting the network to use
network = hc(asia, restart = 10, optimized = FALSE)
parameters = bn.fit(network, train)

classify = function(parameters, classify_data, info){
  # Cleaning data
  clean_data = c()
  for (i in 1:length(classify_data)){
    clean_data = c(clean_data, as.character(classify_data[1,i]));
  }

  grain = as.grain(parameters)
  structure = compile(grain) # creating junction tree, separators & residuals. Potentials for the cliqu

  goal = c("S")
  evi = setEvidence(structure, nodes = info, states = clean_data)
  dist = querygrain(evi, nodes = goal)
  if (dist$S[1] < 0.5){
    return("yes")
  }else{
    return("no")
  }
}

# Making predictions on the test set
nodes = c("A", "T", "L", "B", "E", "X", "D")
pred = c()
for (i in 1:nrow(test)){
  pred = c(pred, classify(parameters, test[i,][-2], info = nodes))
}
```

Computing confusion matrix and plotting it

```
conf_matrix = table(real_s, pred)
print(conf_matrix)
```

```
##        pred
## real_s  no yes
##     no  358 147
##     yes 120 375
```

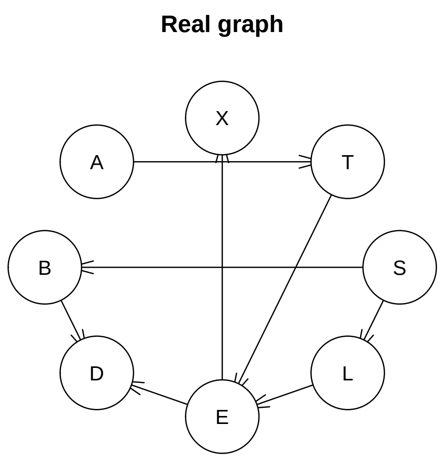
```
plot(conf_matrix, main = "Confusion matrix")
```

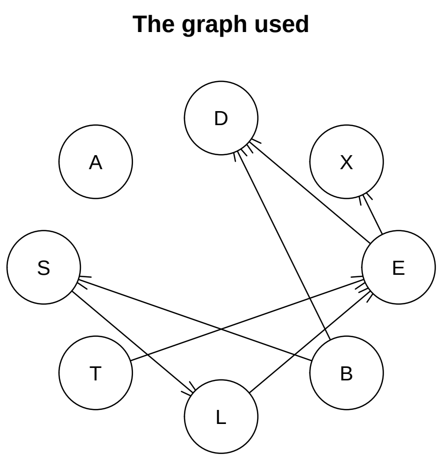


```
acc = (conf_matrix[1]+conf_matrix[4])/sum(conf_matrix)
```

Comparing the "real" network with the network used

```
plot(real_graph, main = "Real graph")
plot(network, main = "The graph used")
```



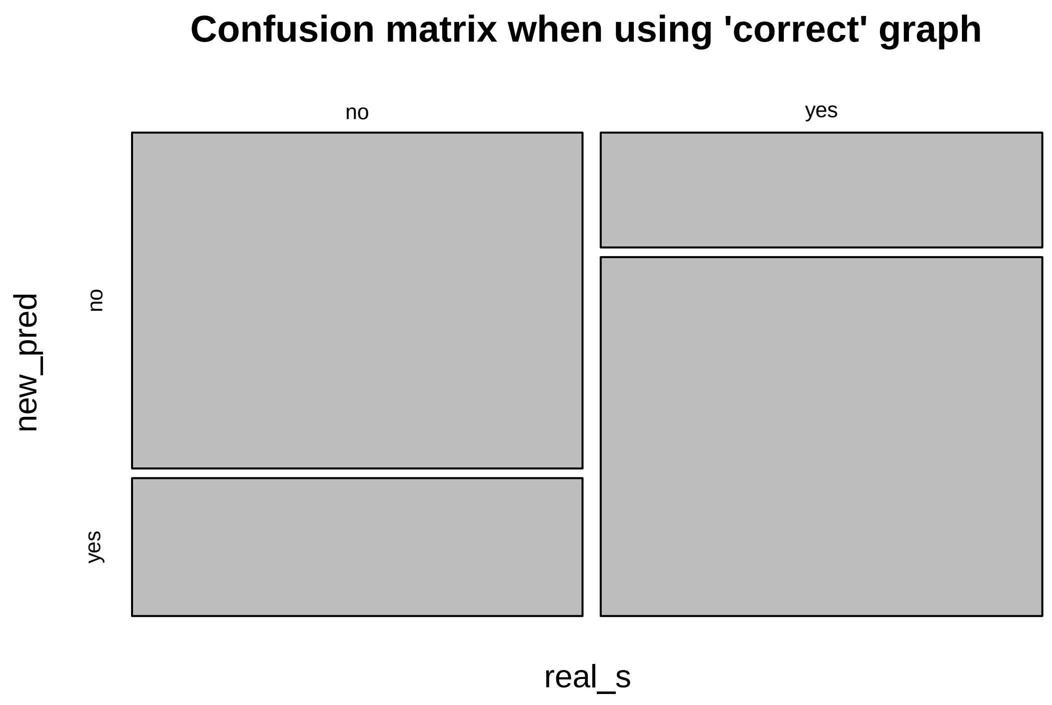


Redoing the classification when If the real network is used in calculation.

```
# Making predictions on the test set
new_parameters = bn.fit(real_graph, train)
nodes = c("A", "T", "L", "B", "E", "X", "D")
new_pred = c()
for (i in 1:nrow(test)){
  new_pred = c(new_pred, classify(new_parameters, test[i,][-2], info = nodes))
}
new_conf_matrix = table(real_s, new_pred)
print(new_conf_matrix)
```

```
##        new_pred
## real_s  no yes
##     no  358 147
##     yes 120 375
```

```
plot(new_conf_matrix, main = "Confusion matrix when using 'correct' graph")
```



```r
new_acc = (new_conf_matrix[1]+new_conf_matrix[4])/sum(new_conf_matrix)
```

We get the exact same performance out of the two graphs when doing classification, when though one of them is considered "the real one". This can be explained by that the node **S** has the same connections to other nodes (only **B** & **L**) in both networks. This means that we don't get any additional information by having other parts of the network "correct".

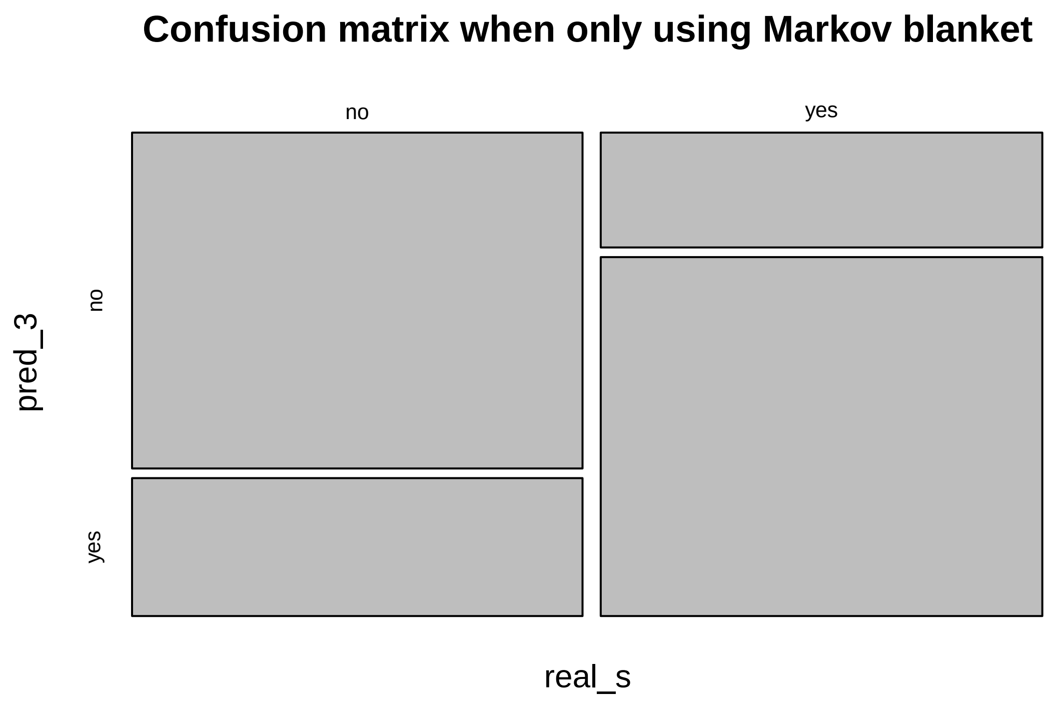
3) Markov blanket

Repeating the predictions made in 2), with the exception that the information available is restricted the Markov blanket for node **S**. This can be seen in the real graph above. The Markov blanket is defined as set of nodes that includes parents, children and other parents of children for a node. For node **S**, the Markov blanket consists of nodes **B** and **L**.

```r
classify = function(parameters, classify_data, info){
  # Cleaning data
  clean_data = c()
  for (i in 1:length(classify_data)){
    clean_data = c(clean_data, as.character(classify_data[1,i]));
  }

  grain = as.grain(parameters)
  structure = compile(grain) # creating junction tree, separators & residuals. Potentials for the cliqu

  goal = c("S")
  evi = setEvidence(structure, nodes = info, states = clean_data)
  dist = querygrain(evi, nodes = goal)
  if (dist$S[1] < 0.5){
    return("yes")
  }else{
    return("no")
  }
}

train = asia[(1:floor(0.8*length(asia[,1]))),]
test = asia[(floor(0.8*length(asia[,1])+1):length(asia[,1])),]
real_graph = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
real_s = test$S
new_parameters = bn.fit(real_graph, train)

mb = c("L", "B")
pred_3 = c()
for (i in 1:nrow(test)){
  pred_3 = c(pred_3, classify(new_parameters, test[i,][4:5], info = mb))
}
conf_matrix_3 = table(real_s, pred_3)
print(conf_matrix_3)
```

```
##       pred_3
## real_s  no yes
##    no  358 147
##    yes 120 375
```

```r
plot(conf_matrix_3, main = "Confusion matrix when only using Markov blanket")
```



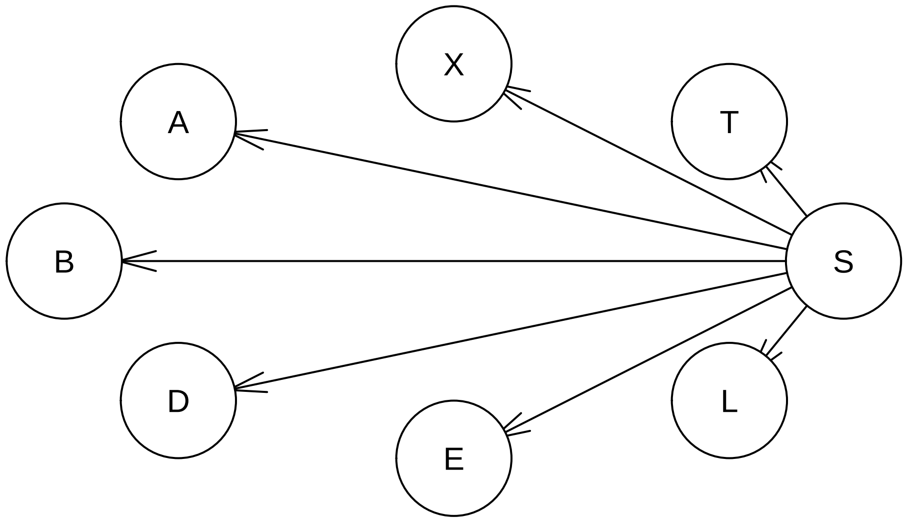
```
acc_3 = (conf_matrix_3[1]+conf_matrix_3[4])/sum(conf_matrix_3)
```

Here, information from node B and L is used. The result is just like the previous ones, meaning that we don't lose any information by only using the Markov blanket when doing inference from the graph.

4) Naive bayes classifier

To represent a Naive Bayes classifier using a Bayesian Network, node **S** needs to be connected to all other nodes. It can be said that the separating set for **S** makes up the entire network. This means that when deriving probabilistic inference from **S**, all available information in the network is used and taken into consideration.

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

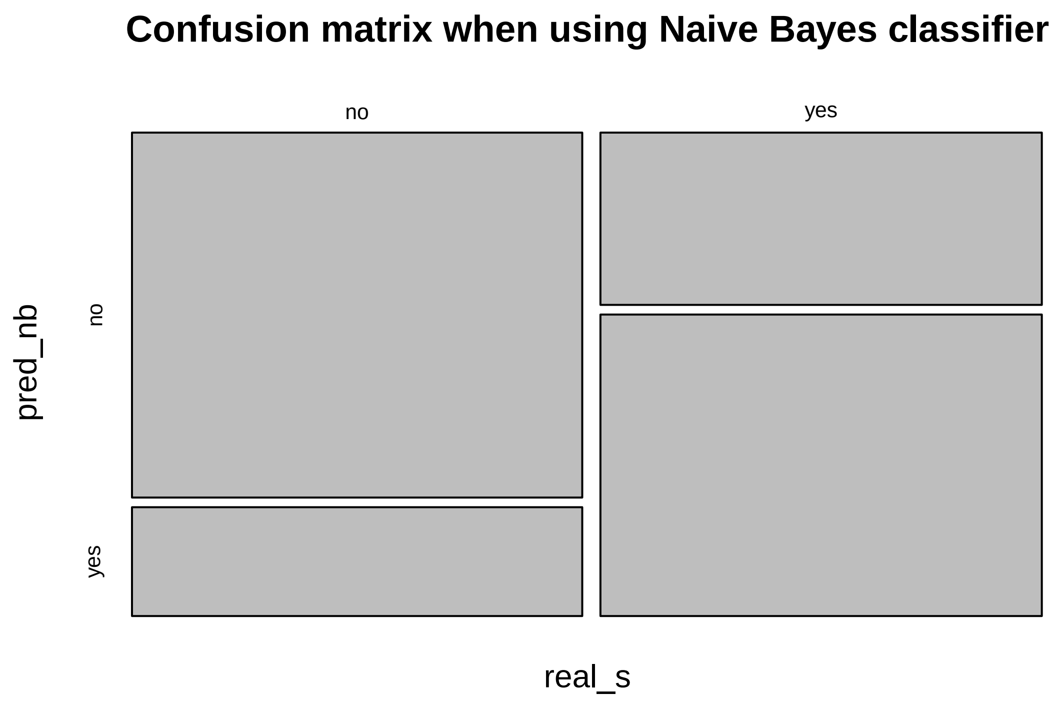


Here is the network modeled to represent a Naive Bayes classifier.

```r
# Classification using the new network
nb =  c("A", "T", "L", "B", "E", "X", "D")
nb_parameters = bn.fit(nb_network, train)
pred_nb = c()
for (i in 1:nrow(test)){
  pred_nb = c(pred_nb, classify(nb_parameters, test[i,][-2], info = nb))
}
conf_matrix_nb = table(real_s, pred_nb)
print(conf_matrix_nb)
```

```
##        pred_nb
## real_s  no yes
##     no  389 116
##     yes 180 315
```

```r
plot(conf_matrix_nb, main = "Confusion matrix when using Naive Bayes classifier")
```



```
acc_nb = (conf_matrix_nb[1]+conf_matrix_nb[4])/sum(conf_matrix_nb)
```

The performance is worse of than using previous methods.

5) Comparissons of 2-4

```
data = data.frame(x = c(acc, new_acc, acc_3, acc_nb), y = c("Own DAG", "Correct DAG", "Markov Blanket",
print(data)
```

```
##       x              y
## 1 0.733        Own DAG
## 2 0.733    Correct DAG
## 3 0.733 Markov Blanket
## 4 0.704    Naive Bayes
```

The table above shows the accuracy of the different classification done. The reason why the all results except **Naive Bayes** is the same is because the relations to node **S** is the same. Then, when using **Naive Bayes**, node **S** get related to all other nodes in the network which worsens performance. The model now assumes there is a possible dependence between S and all the other nodes in the network, which is not correct. In other words, the Markov Blanket i changed in 5). This makes it so that the statistical inference made is different from the previous exercises.