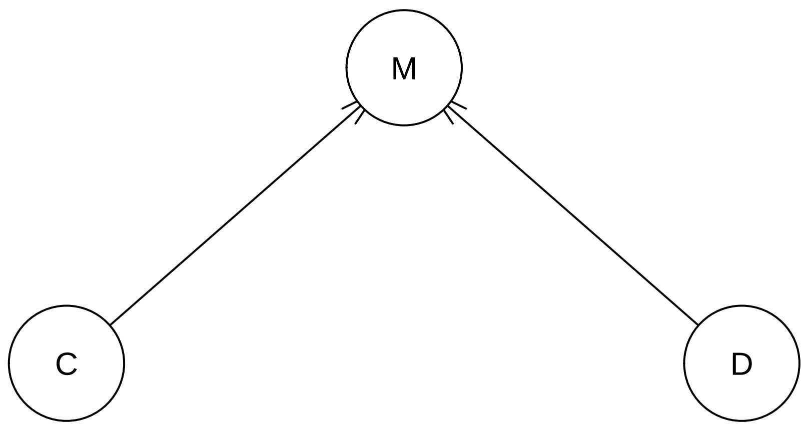
Tenta Oct 2019

Oskar Hidén - oskhi827

10/21/2020

Graphical models

```r
library(bnlearn)
?model2network
?custom.fit

bn = model2network(string = "[C][D][M|C:D]")
plot(bn)

cptC = matrix(c(1/3, 1/3, 1/3), ncol = 3, dimnames = list(NULL, c("D1", "D2", "D3")))
cptD = matrix(c(1/3, 1/3, 1/3), ncol = 3, dimnames = list(NULL, c("D1", "D2", "D3")))
cptM = c(0, 0.5, 0.5, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0.5, 0, 0.5, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0.5, 0.5)
dim(cptM) = c(3, 3, 3)
dimnames(cptM) = list("M" = c("D1", "D2", "D3"), "D" =  c("D1", "D2", "D3"),
                      "C" = c("D1", "D2", "D3"))
monty_hall = custom.fit(bn, dist =list(C = cptC, D = cptD, M = cptM) )

library(gRbase)
```




```r
library(gRain)

# Create Graphical independance network ( grain object )
fit_grain = as.grain(monty_hall)

first_pick = which.max(querygrain(fit_grain, nodes = "C")$C) #due to equal prop which.max will take the

monty_pick = "D2"
actions = c(names(first_pick), monty_pick)

# Finding/evidance or potentials
nodes_evid = c("D", "M")
evid = setEvidence(fit_grain, nodes_evid, actions )
# Querygrain to get conditional distributon
post_cond = querygrain(evid, nodes="C")
post_cond
```

```
## $C
## C
##        D1        D2        D3
## 0.3333333 0.0000000 0.6666667
```

```r
cat("second_pic = ", names(which.max(post_cond$C)))
```

```
## second_pic =  D3
```

```r
# Monty does another pic
monty_pick = "D3"
actions = c(names(first_pick), monty_pick)

# Finding/evidance or potentials
nodes_evid = c("D", "M")
evid = setEvidence(fit_grain, nodes_evid, actions )
# Querygrain to get conditional distributon
post_cond = querygrain(evid, nodes="C")
post_cond
```

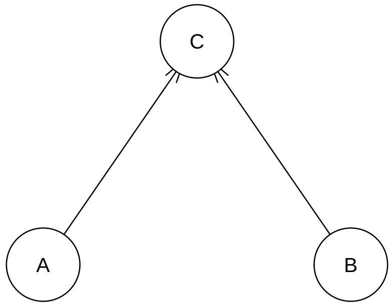
```
## $C
## C
##        D1        D2        D3
## 0.3333333 0.6666667 0.0000000
```

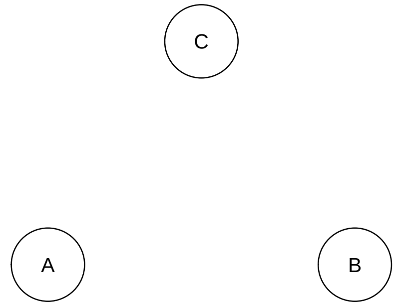
```r
cat("second_pic = ", names(which.max(post_cond$C)))
```

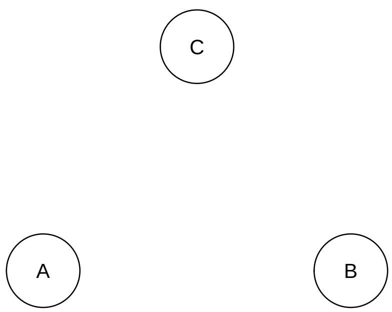
```
## second_pic =  D2
```

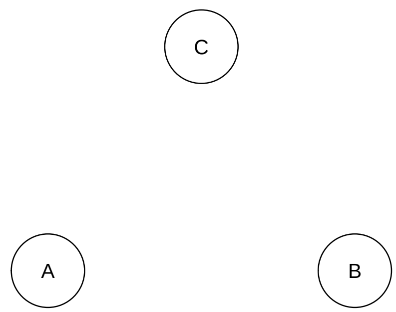
Switch door in both cases.

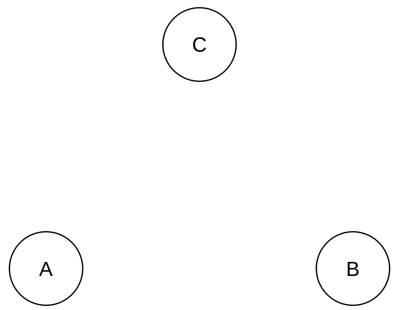
```r
xor_bn = model2network(string = "[A][B][C|A:B]")
#plot(xor_bn)

# discrete Bayesian network from expert knowledge.
net = model2network("[A][B][C|A:B]")
cptA = matrix(c(0.5, 0.5), ncol = 2, dimnames = list(NULL, c("TRUE", "FALSE")))
cptB = matrix(c(0.5, 0.5), ncol = 2, dimnames = list(NULL, c("TRUE", "FALSE")))
cptC = c(0, 1, 1, 0, 1, 0, 0, 1)
dim(cptC) = c(2, 2, 2)
dimnames(cptC) = list("C" = c("TRUE", "FALSE"), "A" =  c("TRUE", "FALSE"),
                   "B" = c("TRUE", "FALSE"))
cfit = custom.fit(xor_bn, dist = list(A = cptA, B = cptB, C = cptC))

sim = rbn(cfit, n=1000)


# learn bn from simulations above using hill climbing
n = dim(sim)[1]

for (i in 1:10) {
  sim_samp = sim[sample(1:n, size=floor(n/4), replace=FALSE), ]
  learned_bn = hc(sim_samp, restart = 0, score = "bic")
  plot.new()
  plot(learned_bn)
}
```

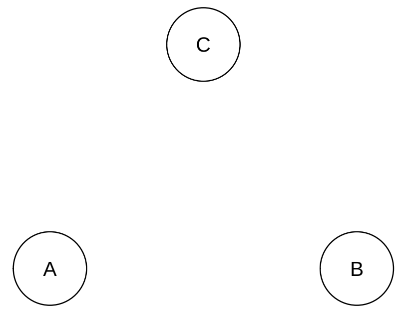

C
A
B

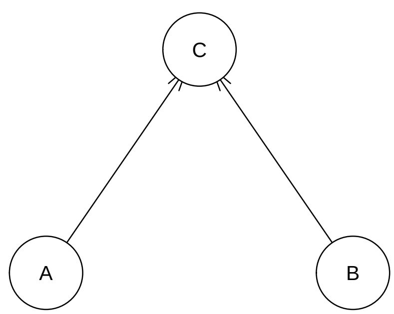

C
A
B

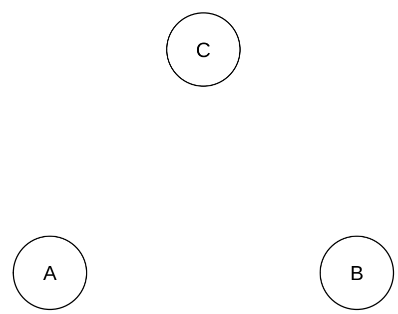

C
A
B

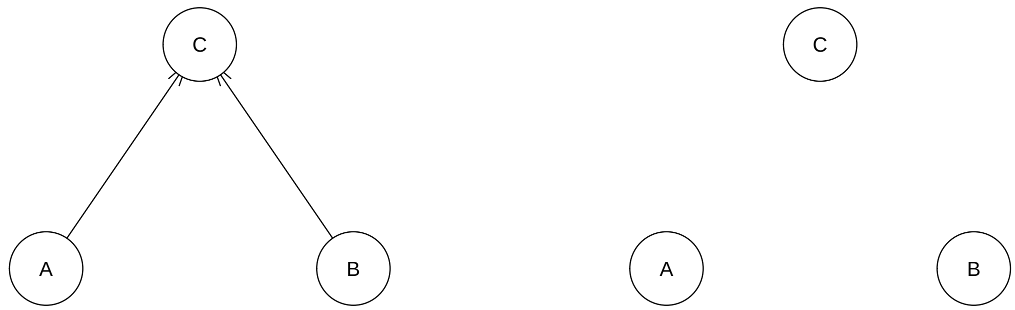

C
A
B


C
A
B


C
A
B


C
A
B


C
A
B



Given that the problem at hand is rather easy (i.e., many observations and few variables), why does the hill-climbing algorithm fail to recover the true BN structure in most runs ?

HC composes the network by changing (adding, removing och revering) an edge at a time. Then HC evaluates the change by a scoring method, if the score increases HC keep the change. If worse or equal HC sticks with previous BN. HC has a risk of getting trapped in local omptimum and therfore HC fail to recover the true BN. In this case, since HC is evaluateing one edge (i.e. if there is a dependence between the nodes) at a time. HC will not add any edge becase in this data there is no dependence between e.g. A and C (marginal independence) if we do not know B.

2 - Hidden Markow Models

```
library(HMM)

states = c("1", "2", "3", "4", "5", "6", "7", "8", "9", "10")
symbols = c("1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11")
start_prob = rep(0, 10)
start_prob[1] = 1
#start_prob = NULL
sur_state = function(x){
  state = x%%10
  if (state ==0) {
    state=10
  }
  return(state)
}
trans_prob = matrix(data=0, nrow = 10, ncol=10)
for (i in 1:10) {
  trans_prob[i,i] = 0.5
  trans_prob[i,sur_state(i+1)] = 0.5
}
emmis_prob = matrix(data=0, nrow = 10, ncol=11)
```

```r
for (i in 1:10) {
  for (j in -2:2) {
    emmis_prob[i,sur_state(i+j)] = 0.1
  }
  emmis_prob[i,11] = 0.5
}
HMM = initHMM(states, symbols, start_prob, trans_prob, emmis_prob)
HMM
```

```
## $States
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
##
## $Symbols
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11"
##
## $startProbs
##  1  2  3  4  5  6  7  8  9 10
##  1  0  0  0  0  0  0  0  0  0
##
## $transProbs
##     to
## from   1   2   3   4   5   6   7   8   9  10
##   1  0.5 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
##   2  0.0 0.5 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0
##   3  0.0 0.0 0.5 0.5 0.0 0.0 0.0 0.0 0.0 0.0
##   4  0.0 0.0 0.0 0.5 0.5 0.0 0.0 0.0 0.0 0.0
##   5  0.0 0.0 0.0 0.0 0.5 0.5 0.0 0.0 0.0 0.0
##   6  0.0 0.0 0.0 0.0 0.0 0.5 0.5 0.0 0.0 0.0
##   7  0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.5 0.0 0.0
##   8  0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.5 0.0
##   9  0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.5
##   10 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5
##
## $emissionProbs
##       symbols
## states   1   2   3   4   5   6   7   8   9  10  11
##     1  0.1 0.1 0.1 0.0 0.0 0.0 0.0 0.0 0.1 0.1 0.5
##     2  0.1 0.1 0.1 0.1 0.0 0.0 0.0 0.0 0.0 0.1 0.5
##     3  0.1 0.1 0.1 0.1 0.1 0.0 0.0 0.0 0.0 0.0 0.5
##     4  0.0 0.1 0.1 0.1 0.1 0.1 0.0 0.0 0.0 0.0 0.5
##     5  0.0 0.0 0.1 0.1 0.1 0.1 0.1 0.0 0.0 0.0 0.5
##     6  0.0 0.0 0.0 0.1 0.1 0.1 0.1 0.1 0.0 0.0 0.5
##     7  0.0 0.0 0.0 0.0 0.1 0.1 0.1 0.1 0.1 0.0 0.5
##     8  0.0 0.0 0.0 0.0 0.0 0.1 0.1 0.1 0.1 0.1 0.5
##     9  0.1 0.0 0.0 0.0 0.0 0.0 0.1 0.1 0.1 0.1 0.5
##     10 0.1 0.1 0.0 0.0 0.0 0.0 0.0 0.1 0.1 0.1 0.5
```

```r
obs = c(1,11,11,11)

viterbi(HMM, obs)
```

```
## [1] "1" "1" "1" "1"
```

2 - Alternative implemenation of HMM

```
# ----- Alternative implementation of HMM: -----
States=1:10 # Sectors
Symbols=1:11 # Sectors + malfunctioning
transProbs=matrix(c(.5,.5,0,0,0,0,0,0,0,0,
                    0,.5,.5,0,0,0,0,0,0,0,
                    0,0,.5,.5,0,0,0,0,0,0,
                    0,0,0,.5,.5,0,0,0,0,0,
                    0,0,0,0,.5,.5,0,0,0,0,
                    0,0,0,0,0,.5,.5,0,0,0,
                    0,0,0,0,0,0,.5,.5,0,0,
                    0,0,0,0,0,0,0,.5,.5,0,
                    0,0,0,0,0,0,0,0,.5,.5,
                    .5,0,0,0,0,0,0,0,0,.5), nrow=length(States), ncol=length(States), byrow = TRUE)
emissionProbs=matrix(c(.1,.1,.1,0,0,0,0,0,.1,.1,.5,
                       .1,.1,.1,.1,0,0,0,0,0,.1,.5,
                       .1,.1,.1,.1,.1,0,0,0,0,0,.5,
                       0,.1,.1,.1,.1,.1,0,0,0,0,.5,
                       0,0,.1,.1,.1,.1,.1,0,0,0,.5,
                       0,0,0,.1,.1,.1,.1,.1,0,0,.5,
                       0,0,0,0,.1,.1,.1,.1,.1,0,.5,
                       0,0,0,0,0,.1,.1,.1,.1,.1,.5,
                       .1,0,0,0,0,0,.1,.1,.1,.1,.5,
                       .1,.1,0,0,0,0,0,.1,.1,.1,.5), nrow=length(States), ncol=length(Symbols), byrow =
startProbs=c(.1,.1,.1,.1,.1,.1,.1,.1,.1,.1)
hmm=initHMM(States,Symbols,startProbs,transProbs,emissionProbs)

viterbi(hmm, obs)
```

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

3 - (No SPM) - Extra - Rund Reinfore w gussian q funciton.

```
policy = function(state, mu, sigma){
  sig = exp(state%*%sigma)
  probs = c()
  for (a in 1:4) {
    probs = c(probs, (1/(sig*sqrt(pi*2)))*exp(-((a-mu%*%state)^2)/(2*sig^2)))
  }
  return(probs)
}

policy_action = function(state,mu,sigma){
  probs = policy(state, mu, sigma)

  return(sample(1:4,size = 1, prob = probs))
}

state = c(1,4)
mu = c(1,1)
```

```r
sigma = c(0.5, 0.5)

action = policy_action(state, mu, sigma)
action
```

```
## [1] 3
```

```r
update_policy = function(state, action, step, alpha = 0.2, gamma = 0.95, reward = 5){
  G = gamma^(n-step)*reward
  sig = exp(state%*%sigma)

  delta_mu = (action-mu%*%state)/sig^2 * state
  delta_sigma = ((action-mu%*%state)^2/sig^2 - 1) * state
  mu <<- mu + alpha*G*gamma^i * delta_mu
  sigma <<- sigma + alpha*G*gamma^i * delta_sigma
}

update_policy(state, action, 1)
mu
```

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

```r
sigma
```

```
## [1] 0.5 0.5
```

4 - Gaussian process

```r
#install.packages('kernlab')
library(kernlab)



# Algorithm 2.1 sunns in n^3/6 instead of O(n^3). GP can be estimated faster, O(n) with eg. KISS-GP
posteriorGP = function(X_input, y_targets, k_cov_function, sigmaNoise=1, XStar){
  A = k_cov_function(X_input, X_input)
  A = A + diag(length(X_input))*sigmaNoise^2
  L = t(chol(A)) # chol Returns t(L)
  L_y = solve(L, y_targets)
  alpha = solve(t(L), L_y)

  #k_star = k_cov_function(X_input, XStar)
  #f_star = t(k_star)%*%alpha

  #v = solve(L,k_star)
  #V_f_star = k_cov_function(XStar, XStar) - t(v)%*%v
  n = length(y_targets)
  log_mar = -0.5 * (t(y_targets)%*%alpha) - sum(diag(L))-(n/2)*log(2*pi)
  return(list("log" = log_mar) ) #list("mean"=f_star, "cov" = V_f_star, "log" = log_mar))
}
```

```
temprature = read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/TempTull
time = (1:2190)
nr_year = 2190/365
day = rep((1:365), nr_year)

reduce_data = function(array , nth){
  array[seq(1, length(array), nth)]
}

temp = reduce_data(temprature$temp,5)
time = reduce_data(time, 5)
day = reduce_data(day, 5)

set_se_kernel = function(ell, sigma_f){
  se_kernel = function(x , y){
    r_square = sum((x-y)*(x-y)) # Euclidian distance^2
    return(sigma_f^2*exp(-r_square/(2*ell^2)))
  }
  class(se_kernel) <- "kernel"
  return(se_kernel)
}

LM = function(sigmaf_ell, X, y, k, sigmaNoise){
  #set kernel
  kernel = k(sigmaf_ell[2], sigmaf_ell[1])

  k_cov = function(x, x_star){
    return(kernelMatrix(kernel = kernel, x=x, y=x_star))
  }

  post = posteriorGP(X, y, k_cov, sigmaNoise, X)
  return(post$log)
}

regr = lm(scale(temp) ~ scale(time) + I(scale(time))^2)
sigmaNoiseFit = sd(regr$residuals)

# using Optim
res = optim(par = c(1,0.1), fn = LM, X=scale(time),y=scale(temp),k=set_se_kernel,sigmaNoise=sigmaNoiseF

res$par
```

```
## [1] 1.114025 0.153773
```

```
res$value
```

```
## [1] -918.7371
```

```
# Grid search
best_s = 20
best_l = 0.2
```

```r
best_log = LM(c(best_s, best_l), scale(time), scale(temp), set_se_kernel, sigmaNoiseFit)
best_log
```

```
##           [,1]
## [1,] -988.9449
```

```r
for (i in seq(1,3,1)) { # sigma_F
  for (j in seq(0.1,0.5,0.1)) {  # ell
    foo = LM(c(i, j), scale(time), scale(temp), set_se_kernel, sigmaNoiseFit)
    if (foo > best_log) {
      best_log = foo
      best_s = i
      best_l = j
    }
  }
}
best_log
```

```
##           [,1]
## [1,] -921.2252
```

```r
best_s
```

```
## [1] 2
```

```r
best_l
```

```
## [1] 0.2
```

```r
# 2 - Fraud
library(kernlab)
data <- read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/banknoteFraud
names(data) <- c("varWave","skewWave","kurtWave","entropyWave","fraud")
data[,5] <- as.factor(data[,5])

set.seed(111)
SelectTraining <- sample(1:dim(data)[1], size = 1000,replace = FALSE)
train = data[SelectTraining,]
test = data[-SelectTraining,]

select_valid = sample(1:dim(train)[1], size = 200, replace = FALSE)
valid = train[select_valid,]
train = train[-select_valid,]

LM_class = function(sigma){
  gp_fraud = gausspr(fraud ~ varWave + skewWave + kurtWave + entropyWave, data=train, type="classificat
                     kpar=list("sigma" = sigma))
  predict = predict(gp_fraud, valid[,1:4])
  table = table(predict, valid[,5])
  missclassrate = 1- sum(diag(table))/sum(table)
```

```r
  return(missclassrate)
}

opti = optim(par = 0.1, fn = LM_class, lower = c(.Machine$double.eps), method="L-BFGS-B")
opti$par
```

```
## [1] 0.1
```

```r
opti$value
```

```
## [1] 0.01
```

```r
# run on test data
  gp_fraud = gausspr(fraud ~ varWave + skewWave + kurtWave + entropyWave, data=train, type="classificati
                     kpar=list("sigma" = opti$par))
  predict = predict(gp_fraud, test[,1:4])
  table = table(predict, test[,5])
  missclassrate = 1- sum(diag(table))/sum(table)
  table
```

```
##
## predict   0   1
##       0 208   0
##       1  10 154
```

```r
  missclassrate
```

```
## [1] 0.02688172
```