exam_2019_mysol

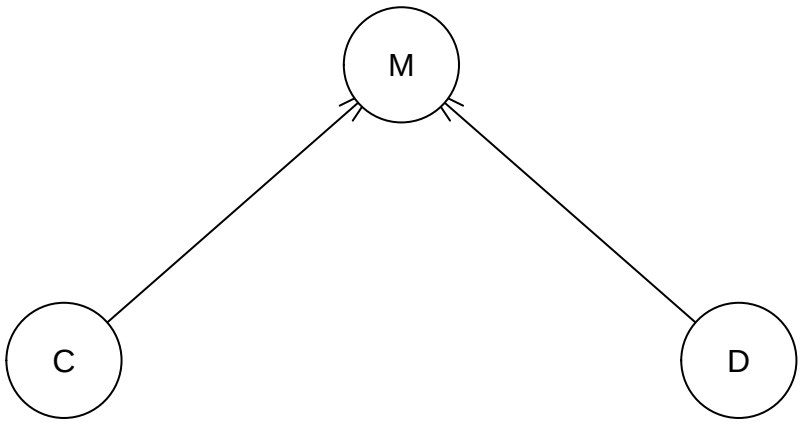
David Björelind

10/22/2020

1. Graphical Models

a)

```
#C: Car
#D: Door of choice
#M: Monty's choice

# Making the network model
graph = model2network("[D][C][M|C:D]")
plot(graph)
```



```r
# Making the Conditional Probability Tables
cptC = matrix(c(1/3, 1/3, 1/3), ncol = 3, dimnames = list(NULL, c("Car1", "Car2", "Car3")))

cptD = matrix(c(1/3, 1/3, 1/3), ncol = 3, dimnames = list(NULL, c("Choice1", "Choice2", "Choice3")))

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,
         0.5, 0.5, 0)
dim(cptM) = c(3,3,3)
dimnames(cptM) = list("M" = c("Door1", "Door2", "Door3"), "D" =  c("Choice1", "Choice2", "Choice3"), "C

dist = list("D" = cptD, "C"= cptC, "M" = cptM) # Largest one needs to be last and names needs to be the
parameters = custom.fit(graph, dist = list("D" = cptD, "C"= cptC, "M" = cptM))

### EXAKT INFERENCE ###
grain = as.grain(parameters)
structure = compile(grain) # creating junction tree, separators & residuals. Potentials
goal = c("C")
evi = setEvidence(structure, nodes = c(""), states = c(""))
dist = querygrain(evi, nodes = goal)
# Picking door 1, monty 2
evi = setEvidence(structure, nodes = c("D", "M"), states = c("Choice1", "Door2"))
querygrain(evi, nodes = goal)
```

```
## $C
## C
##      Car1      Car2      Car3
## 0.3333333 0.0000000 0.6666667
```

```r
# Picking door 3, monty 2
evi = setEvidence(structure, nodes = c("D", "M"), states = c("Choice3", "Door2"))
querygrain(evi, nodes = goal)
```

```
## $C
## C
##      Car1      Car2      Car3
## 0.6666667 0.0000000 0.3333333
```

```r
evi = setEvidence(structure, nodes = c("D", "M"), states = c("Choice1", "Door3"))
querygrain(evi, nodes = goal)
```

```
## $C
## C
##      Car1      Car2      Car3
## 0.3333333 0.6666667 0.0000000
```

```
### APPROXIMATE INFERENCE ###

a = cpdist(fitted = parameters, nodes = "C", evidence = TRUE)
table(a)/sum(table(a))
```

```
## a
##   Car1   Car2   Car3
## 0.3268 0.3272 0.3460
```

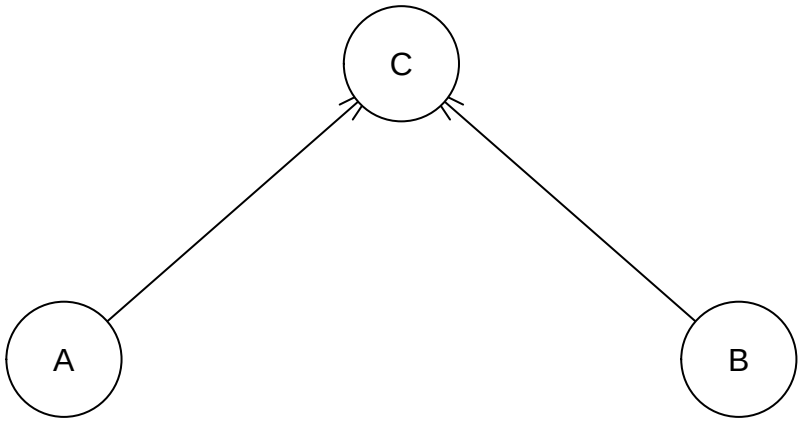
```
b =  cpdist(fitted = parameters,nodes = "C", evidence = (D=="Choice1" & M=="Door2"))
table(b)/sum(table(b))
```

```
## b
##      Car1      Car2      Car3
## 0.3482032 0.0000000 0.6517968
```

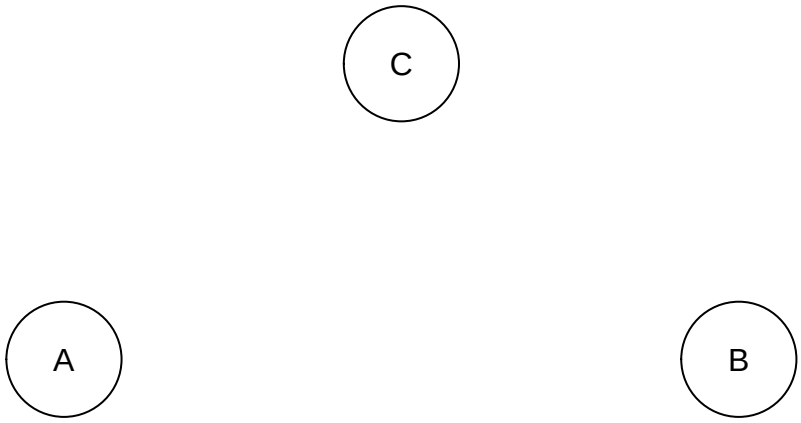
Conclusion: Always switch doors!!

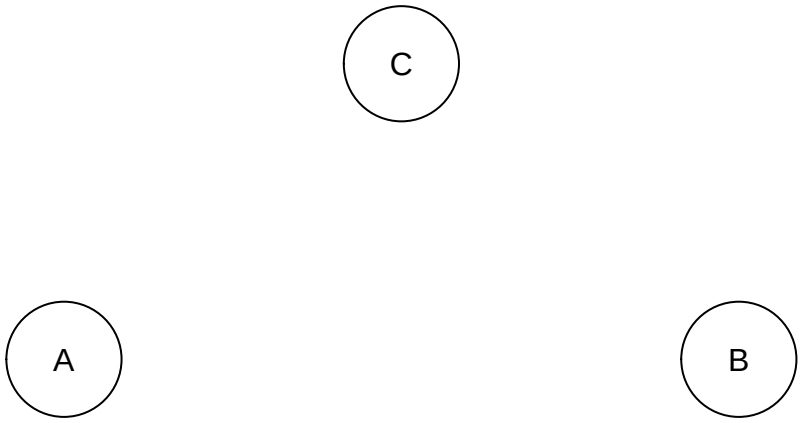
b)

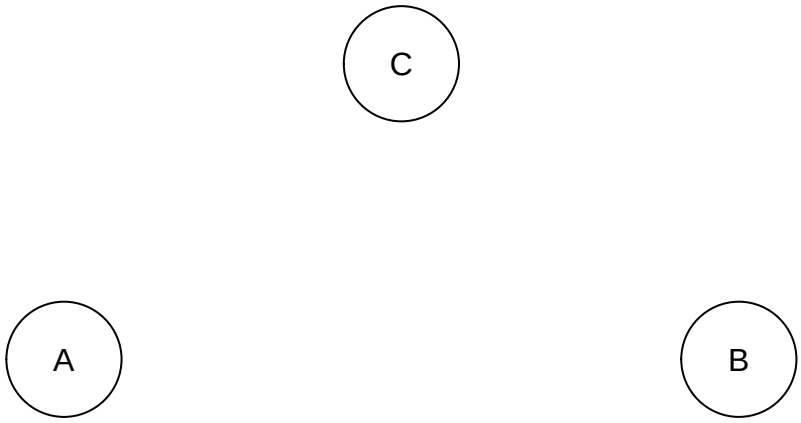
```
# Making the network model
graph = model2network("[B][A][C|A:B]")
plot(graph)
```

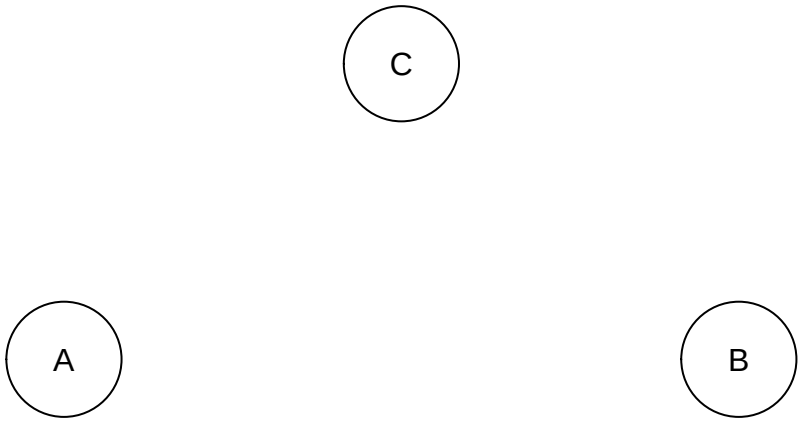


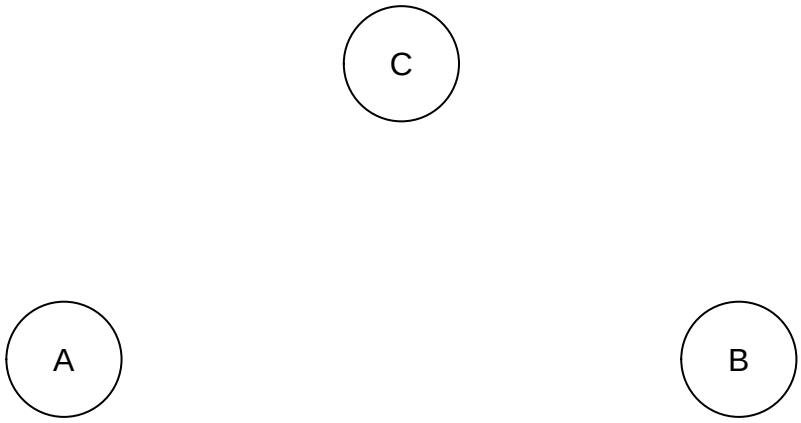
```
# Making the Conditional Probability Tables
cptA = matrix(c(1/2, 1/2), ncol = 2, dimnames = list(NULL, c("A0", "A1")))

cptB = matrix(c(1/2, 1/2), ncol = 2, dimnames = list(NULL, c("B0", "B1")))

cptC = c(1, 0,
         0, 1,
         0, 1,
         1, 0)
dim(cptC) = c(2,2,2)
dimnames(cptC) = list("C" = c("C0", "C1"), "A" =  c("A0", "A1"), "B" = c("B0", "B1"))

dist = list("A" = cptA, "B"= cptB, "C" = cptC) # Largest one needs to be last and names needs to be the
parameters = custom.fit(graph, dist = list("A" = cptA, "B"= cptB, "C" = cptC))
niter = 1000
# Drawing samples
sample = rbn(parameters, n=niter)

# Learning HC from samples
learn_graph = hc(sample)

# Repeating 10 times
for (i in 1:10){
  sample = rbn(parameters, n=niter)
  learn_graph = hc(sample)
  plot.new()
  plot(learn_graph)
}
```

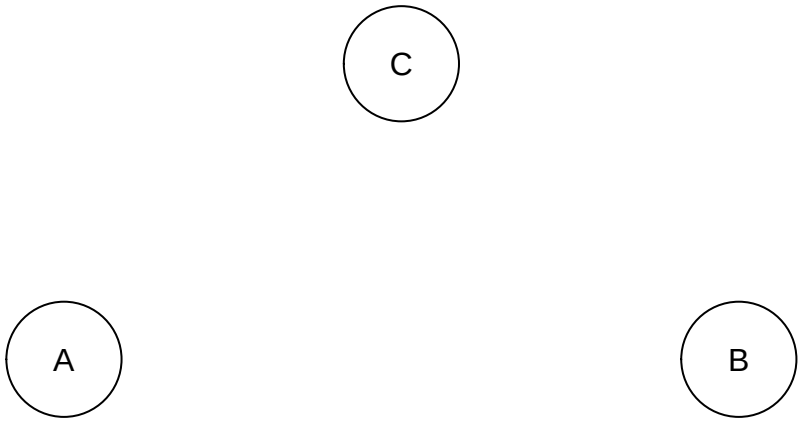

C
A
B

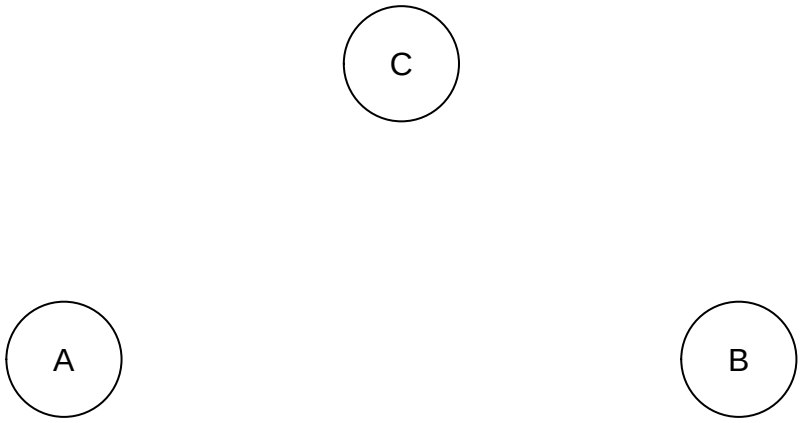

C
A
B

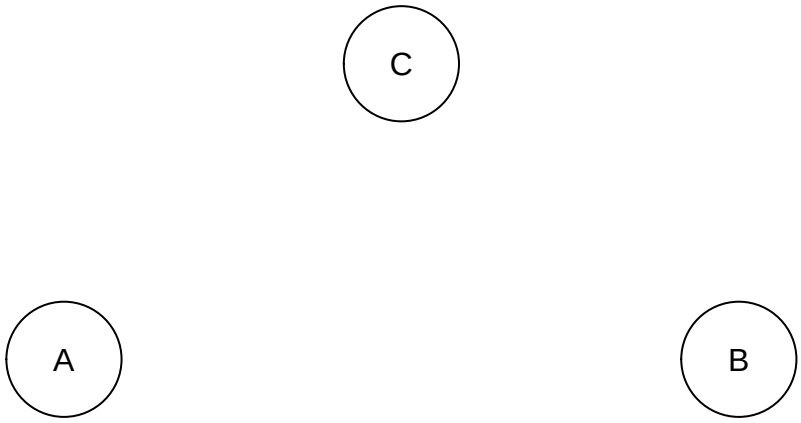

C
A
B

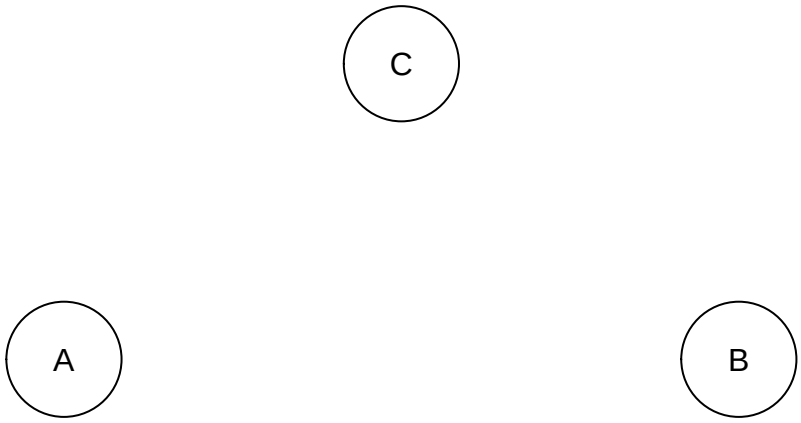

C
A
B

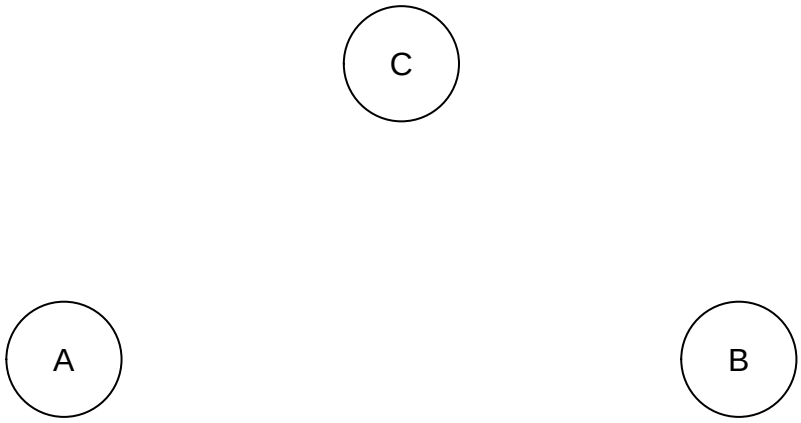

C
A
B


C
A
B


C
A
B


C
A
B


C
A
B



Why does HC fail to recover the true BN structure in most runs?

The algorithm gets stuck on a local optimum? Since HC starts in a random spot (and we're not allowed to use random restarts) it can get stuck. HC is also Score-based.

Edge from A -> C but A is also marginally dependant of C. That is why the algorithm can't find two dependant variables!

2. Hidden Markov Models

```
# Building a Hidden Markov Model
state = rep(1:10) # Actual number of states that the robot can be in (Hidden)
probs = rep(1/10, 10)
symbols = 1:11 # States that we can observe (Not hidden)

# If robot is in sector i:
emissionP = matrix(c(
  0.1, 0.1, 0.1, 0, 0, 0, 0, 0, 0.1, 0.1, 0.5,
  0.1, 0.1, 0.1, 0.1, 0, 0, 0, 0, 0, 0.1, 0.5,
  0.1, 0.1, 0.1, 0.1, 0.1, 0, 0, 0, 0, 0, 0.5,
  0, 0.1, 0.1, 0.1, 0.1, 0.1, 0, 0, 0, 0, 0.5,
  0, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0, 0, 0, 0.5,
  0, 0, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0, 0, 0.5,
  0, 0, 0, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0, 0.5,
  0, 0, 0, 0, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5,
  0.1, 0, 0, 0, 0, 0, 0.1, 0.1, 0.1, 0.1, 0.5,
```

```
  0.1, 0.1, 0, 0, 0, 0, 0, 0.1, 0.1, 0.1, 0.5),
  ncol = 11, byrow = TRUE
)

#transP = matrix(c(
#  0.1, 0.1, 0.1, 0, 0, 0, 0, 0, 0.1, 0.1, 0,
#  0.1, 0.1, 0.1, 0.1, 0, 0, 0, 0, 0, 0.1, 0,
#  0.1, 0.1, 0.1, 0.1, 0.1, 0, 0, 0, 0, 0, 0,
#  0, 0.1, 0.1, 0.1, 0.1, 0.1, 0, 0, 0, 0, 0,
#  0, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0, 0, 0, 0,
#  0, 0, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0, 0, 0,
#  0, 0, 0, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0, 0,
#  0, 0, 0, 0, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0,
#  0.1, 0, 0, 0, 0, 0, 0.1, 0.1, 0.1, 0.1, 0,
#  0.1, 0.1, 0, 0, 0, 0, 0, 0.1, 0.1, 0.1, 0,
#  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),
#  ncol = 11, byrow = TRUE
#)
transP = matrix(c(
  0.5, 0.5, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0.5, 0.5, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0.5, 0.5, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0.5, 0.5, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0.5, 0.5, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0.5, 0.5, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0.5, 0.5, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0.5, 0.5, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0.5, 0.5,
  0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0.5),
  ncol = 10, byrow = TRUE
)
# Initializing hidden markov model
robot = initHMM(States = state, Symbols = symbols, startProbs = probs, transProbs = transP, emissionProb

# Defining path
obs = c(1, 11, 11, 11)

# Most probable path, using VITERBI algorithm
posterior(robot, obs)
```

```
##       index
## states   1   2    3     4
##      1  0.2 0.2 0.20 0.175
##      2  0.2 0.2 0.20 0.200
##      3  0.2 0.2 0.20 0.200
##      4  0.0 0.1 0.15 0.175
##      5  0.0 0.0 0.05 0.100
##      6  0.0 0.0 0.00 0.025
##      7  0.0 0.0 0.00 0.000
##      8  0.0 0.0 0.00 0.000
##      9  0.2 0.1 0.05 0.025
##      10 0.2 0.2 0.15 0.100
```

```
viterbi(robot, obs)
```

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

```
# Calculating Smoothing (distribution)

###true = sim_data$states # The true states, used for comparisson later

alpha = exp(forward(robot, obs))
beta = exp(backward(robot, obs))
sum_alpha = apply(alpha, 2, sum)
sum_alphabeta = apply(alpha*beta, 2, sum) # Only needed for filtering

# Smoothed probability distribution (using all available obs)
smoothing = t(apply(alpha*beta, 1, "/", sum_alphabeta))
```

According to Viterbi, most probable path for the scenario is for the robot to stay in sector 1 all the time. Using the smoothing distribution, we are not so sure.

3. Reinforcement Learning

```
# No assignment :(((
```

4. Gaussian Processes

a) Extension of lab

```
# 2.1
posteriorGP = function(X, y, XStar, sigmaNoise, k, sigmaF, l){ # signaF NOT squared
  #par = c(sigmaF, l)
  K = k(X,X, sigmaF^2, l)
  #K = k(par,X,X)
  n = length(XStar)
  L = t(chol(K + sigmaNoise^2*diag(dim(K)[1])))
  kStar = k(X,XStar, sigmaF^2, l)
  #kStar = k(par,X,XStar)
  alpha = solve(t(L), solve(L,y))

  FStar = t(kStar) %*% alpha
  v = solve(L, kStar)
  vf = k(XStar, XStar, sigmaF^2, l) - t(v)%*%v #+ sigmaNoise^2*diag(n) #Adding sigma for noise
  #vf = k(par,XStar, XStar) - t(v)%*%v #+ sigmaNoise^2*diag(n) #Adding sigma for noise
  logmarglike = -t(y)%*%alpha/2 - sum(diag(L)) - n/2*log(2*pi)

  return(list("mean" = FStar,"variance" = vf,"logmarglike" = logmarglike))
  #return(logmarglike)
}

SquaredExpKernel <- function(x1,x2,sigmaF=1,l=3){
```

```
  n1 <- length(x1)
  n2 <- length(x2)
  K <- matrix(NA,n1,n2)
  for (i in 1:n2){
    K[,i] <- sigmaF^2*exp(-0.5*( (x1-x2[i])/l)^2 )
  }
  return(K)
}

# Reading the data
temps = read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/TempTullinge.
time = seq(from=1, to=2190, by=5)
temps = temps[time]
day = rep(seq(from=1, to=361, by=5), times=6)
# Data scaling
daymean = mean(day)
daysd = sd(day)
timemean = mean(time)
timesd = sd(time)
tempsmean = mean(temps)
tempssd = sd(temps)

day_s = scale(day)
time_s = scale(time)
temps_s = scale(temps)

# Search for the best hyperparameters

# Approach 1 - Search Grid
polyFit <- lm(temps_s ~ time_s + I(time_s^2))
sigmaN = sd(polyFit$residuals)

top_sigmaF = 0
top_ell = 0
top_lik = -100000
for(sigmaF in seq(0.1, 3, by=0.1)){
  for(ell in seq(0.01, 0.2, by=0.05)){

    res = posteriorGP(X = time_s, y = temps_s, XStar = time_s, sigmaNoise = sigmaN, k = SquaredExpKernel
    if (res$logmarglike > top_lik){
      print("updating!")
      top_sigmaF <<- sigmaF
      top_ell <<- ell
      top_lik <<- res$logmarglike
    }
  }
}
top_ell
top_sigmaF
top_lik

# Approach 2 - Optim
SEKernel2 <- function(par=c(20,0.2),x1,x2){
  n1 <- length(x1)
```

```
  n2 <- length(x2)
  K <- matrix(NA,n1,n2)
  for (i in 1:n2){
    K[,i] <- (par[1]^2)*exp(-0.5*( (x1-x2[i])/par[2])^2 )
  }
  return(K)
}

# Får det inte att fungera...
#optim(par = c(1,0.1),
#fn = posteriorGP, X=time_s, y=temp_s, k=SEKernel2, sigmaNoise=sigmaN, method="L-BFGS-B",
#lower = c(.Machine$double.eps, .Machine$double.eps), control=list(fnscale=-1))
```

b)

```
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 = data[SelectTraining,]
test_data = data[-SelectTraining,]
  #GPfitfraud <- gausspr(fraud ~  varWave + skewWave + kurtWave + entropyWave, data=train_data, kernel=


acVal = function(par=c(0.1)){# 'par' måste finnas med här!
  GPfitfraud =  gausspr(x = train_data[,1:4], y = train_data[,5], kernel = "rbfdot", kpar = list(sigma=p
  #GPfitfraud <- gausspr(fraud ~  varWave + skewWave + kurtWave + entropyWave, data=train_data, kernel=
  pred = predict(GPfitfraud,test_data[,1:4])
  conf = table(pred, test_data[,5])
  acc = sum(diag(conf))/sum(conf)
  return(acc)
}

optim(par = c(0.05),
fn = acVal, method="L-BFGS-B",
lower = c(.Machine$double.eps, .Machine$double.eps), control=list(fnscale=-1))

### SEARCH GRID

best_acc = 0
best_sigma = 0.05
for(sigma in seq(0.05, 1, by=0.05)){
    acc = acVal(sigma)
    if (acc > best_acc){
      print("updating!")
      best_acc <<- acc
      best_sigma <<- sigma
    }
}
best_acc
best_sigma
```

Another chunk