Advanced Machine Learning - Exam 2017-10-19

Maximilian Pfundstein (maxpf364)

2019-10-22

Contents

1 Graphical Models **1**
1.1 D-Seperation 1
1.2 Percentage of Essentials DAGs 4

2 Hidden Markov Models **4**
2.1 Modeling 4
2.2 Likelihood of Segments 5

3 Gaussian Processes **10**
3.1 A Priori 10
3.2 Posterior 14
3.3 Bayesian View 17

4 Source Code **18**

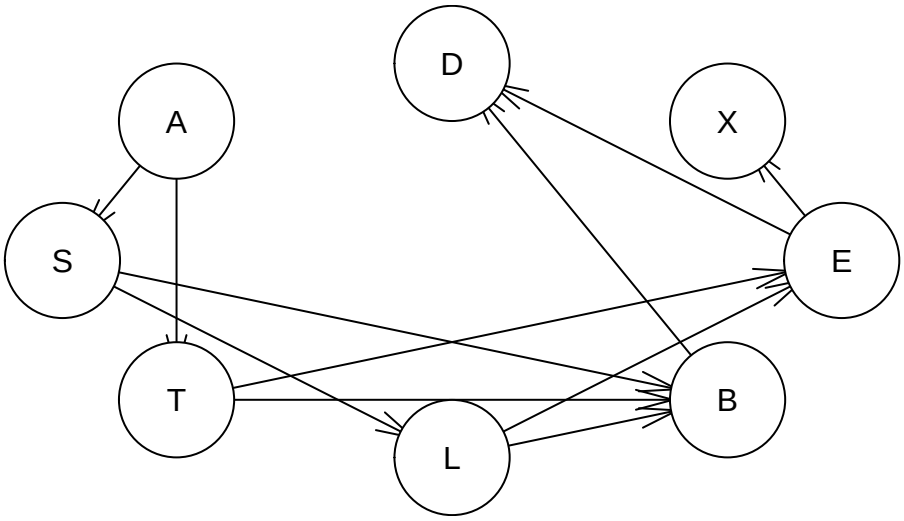
1 Graphical Models

1.1 D-Seperation

```
# Learning the Network
set.seed(123)
data("asia")
bayesian_network = hc(asia,restart=10,score="aic",iss=10)
```

```
## Warning in check.unused.args(extra, c(method.extra.args[[heuristic]],
## score.extra.args[[score]])): unused argument(s): 'iss'.
```

```
# Learning the Parameters
bayesian_network_fit = bn.fit(bayesian_network, asia)
# Compile for faster computations
bayesian_network_grain = compile(as.grain(bayesian_network_fit))

plot(bayesian_network)
```




```
# Case: E is independent from L | S, B

case1 = setFinding(bayesian_network_grain, nodes=c("S", "B", "E"),states=c("yes","yes", "no"))
querygrain(case1, c("D"))
```

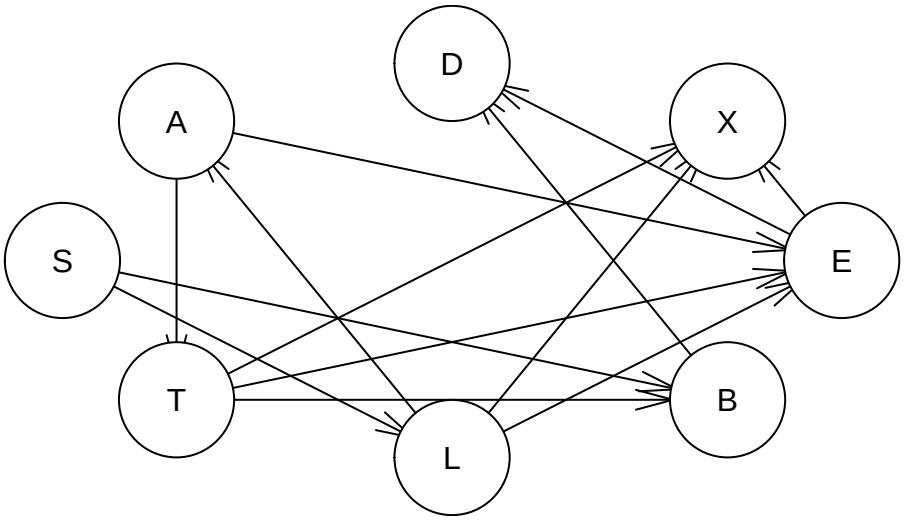
```
## $D
## D
##        no       yes
## 0.2137306 0.7862694
```

```
case2 = setFinding(bayesian_network_grain, nodes=c("S", "B", "E"),states=c("no","yes", "no"))
querygrain(case2, c("D"))
```

```
## $D
## D
##        no       yes
## 0.2137306 0.7862694
```

Solution:

```
# Learn a BN from the Asia dataset, find a separation statement (e.g. B _|_ E | S, T) and, then, check
# it corresponds to a statistical independence.

# SOLUTION
set.seed(123)
data("asia")
hc3<-hc(asia,restart=10,score="bde",iss=10)
plot(hc3)
```




```
hc4<-bn.fit(hc3,asia,method="bayes")
hc5<-as.grain(hc4)
hc6<-compile(hc5)
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("yes","yes","yes"))
querygrain(hc7,c("B"))
```

```
## $B
## B
##        no       yes
## 0.3367347 0.6632653
```

```
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("yes","yes","no"))
querygrain(hc7,c("B"))
```

```
## $B
## B
##        no       yes
## 0.3367347 0.6632653
```

```
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("yes","no","yes"))
querygrain(hc7,c("B"))
```

```
## $B
## B
##       no      yes
## 0.282208 0.717792
```

```
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("yes","no","no"))
querygrain(hc7,c("B"))
```

```
## $B
## B
##       no      yes
## 0.282208 0.717792
```

```
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("no","yes","yes"))
querygrain(hc7,c("B"))
```

```
## $B
## B
```

```
##        no       yes
## 0.4183673 0.5816327
```

```
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("no","yes","no"))
querygrain(hc7,c("B"))
```

```
## $B
## B
##        no       yes
## 0.4183673 0.5816327
```

```
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("no","no","yes"))
querygrain(hc7,c("B"))
```

```
## $B
## B
##        no       yes
## 0.7030014 0.2969986
```

```
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("no","no","no"))
querygrain(hc7,c("B"))
```

```
## $B
## B
##        no       yes
## 0.7030014 0.2969986
```

```
#B _|_ E | S, T
```

1.2 Percentage of Essentials DAGs

```
f = function(graph) {
  return(!is.character(all.equal(cpdag(graph), graph)))
}

N = 50000

# Randomly sample graphs
graphs = random.graph(c("A", "B", "C", "D", "E"), num = N, method="melancon")
unique_graphs = unique(graphs)

length(unique_graphs)/sum(sapply(unique_graphs, f))
```

```
## [1] 10.93112
```

2 Hidden Markov Models

2.1 Modeling

```
set.seed(123)
N = 100

# Defining States Z1, Z2, ..., ZN
```

```r
states = paste(rep("Z", N), 1:N, sep = "")

# Defining Symbols
symbols = c("Door", "Corridor")

# Starting Probabilities
startProbs = rep(1/N, N)

# Transition Probabilities
transProbs = matrix(0, ncol = N, nrow = N)
diag(transProbs) = 0.1
diag(transProbs[,-1]) = 0.9
transProbs[N, N] = 1.0

# Emission Probabilities
emissionProbs = matrix(NA, ncol = 2, nrow = N)
emissionProbs[,1] = 0.1
emissionProbs[,2] = 0.9
emissionProbs[10,] = c(0.9, 0.1)
emissionProbs[11,] = c(0.9, 0.1)
emissionProbs[12,] = c(0.9, 0.1)
emissionProbs[20,] = c(0.9, 0.1)
emissionProbs[21,] = c(0.9, 0.1)
emissionProbs[22,] = c(0.9, 0.1)
emissionProbs[30,] = c(0.9, 0.1)
emissionProbs[31,] = c(0.9, 0.1)
emissionProbs[32,] = c(0.9, 0.1)

robot_hmm = initHMM(States = states,
                    Symbols = symbols,
                    startProbs = startProbs,
                    transProbs = transProbs,
                    emissionProbs = emissionProbs)
```

2.2 Likelihood of Segments

```r
nSim = 100
simulatedStates = simHMM(robot_hmm, nSim)
simulatedStates
```

```
## $states
##   [1] "Z30"  "Z31"  "Z32"  "Z33"  "Z33"  "Z34"  "Z35"  "Z36"  "Z37"  "Z38"
##  [11] "Z38"  "Z39"  "Z40"  "Z41"  "Z42"  "Z43"  "Z44"  "Z45"  "Z46"  "Z46"
##  [21] "Z47"  "Z48"  "Z49"  "Z49"  "Z50"  "Z51"  "Z52"  "Z53"  "Z54"  "Z55"
##  [31] "Z55"  "Z55"  "Z56"  "Z57"  "Z58"  "Z59"  "Z60"  "Z61"  "Z62"  "Z63"
##  [41] "Z64"  "Z65"  "Z66"  "Z67"  "Z68"  "Z69"  "Z70"  "Z71"  "Z72"  "Z73"
##  [51] "Z74"  "Z75"  "Z76"  "Z77"  "Z78"  "Z79"  "Z80"  "Z81"  "Z82"  "Z83"
##  [61] "Z84"  "Z85"  "Z86"  "Z87"  "Z88"  "Z89"  "Z90"  "Z91"  "Z92"  "Z93"
##  [71] "Z94"  "Z95"  "Z96"  "Z97"  "Z98"  "Z99"  "Z100" "Z100" "Z100" "Z100"
##  [81] "Z100" "Z100" "Z100" "Z100" "Z100" "Z100" "Z100" "Z100" "Z100" "Z100"
##  [91] "Z100" "Z100" "Z100" "Z100" "Z100" "Z100" "Z100" "Z100" "Z100" "Z100"
##
```

```
## $observation
##   [1] "Door"     "Door"     "Door"     "Door"     "Corridor" "Corridor"
##   [7] "Door"     "Corridor" "Corridor" "Corridor" "Door"     "Corridor"
##  [13] "Corridor" "Door"     "Corridor" "Corridor" "Corridor" "Door"
##  [19] "Corridor" "Corridor" "Corridor" "Corridor" "Corridor" "Corridor"
##  [25] "Corridor" "Door"     "Corridor" "Corridor" "Corridor" "Corridor"
##  [31] "Corridor" "Corridor" "Corridor" "Corridor" "Corridor" "Corridor"
##  [37] "Corridor" "Corridor" "Door"     "Corridor" "Corridor" "Corridor"
##  [43] "Corridor" "Corridor" "Corridor" "Corridor" "Corridor" "Corridor"
##  [49] "Corridor" "Corridor" "Corridor" "Corridor" "Corridor" "Corridor"
##  [55] "Corridor" "Corridor" "Corridor" "Corridor" "Corridor" "Corridor"
##  [61] "Corridor" "Corridor" "Corridor" "Corridor" "Corridor" "Corridor"
##  [67] "Corridor" "Corridor" "Corridor" "Corridor" "Corridor" "Corridor"
##  [73] "Corridor" "Corridor" "Corridor" "Corridor" "Corridor" "Corridor"
##  [79] "Corridor" "Corridor" "Corridor" "Corridor" "Corridor" "Corridor"
##  [85] "Corridor" "Corridor" "Corridor" "Corridor" "Door"     "Door"
##  [91] "Corridor" "Corridor" "Door"     "Corridor" "Door"     "Corridor"
##  [97] "Corridor" "Corridor" "Corridor" "Corridor"
```

```
custom_forward = function(hmm, observations) {

  Z = matrix(NA, ncol=length(hmm$States), nrow=length(observations))

  Z[1,] = hmm$emissionProbs[, observations[1]] * hmm$startProbs

  for (t in 2:length(observations)) {
    Z[t, ] = hmm$emissionProbs[, observations[t]] * (Z[t-1,] %*% hmm$transProbs)
  }

  return(t(Z))
}

start_time = Sys.time()
alpha = exp(forward(robot_hmm, simulatedStates$observation))
end_time = Sys.time()

builtInForward = end_time - start_time

start_time = Sys.time()
alpha = exp(custom_forward(robot_hmm, simulatedStates$observation))
end_time = Sys.time()

customForward = end_time - start_time

c(builtInForward, customForward)
```

```
## Time differences in secs
## [1] 2.65869164 0.01545811
```

```
filtered = prop.table(alpha, 2)

which.maxima = function(x){
  return(which(x==max(x)))
}
```

```r
# For some reason we get werid behaviour at t=95. The system should be sure it
# is in the last state, shouldn't it?
apply(filtered, 2 ,which.maxima)[1:20]
```

```
## [[1]]
## [1] 10 11 12 20 21 22 30 31 32
##
## [[2]]
## [1] 11 12 21 22 31 32
##
## [[3]]
## [1] 12 22 32
##
## [[4]]
## [1] 12 22 32
##
## [[5]]
## [1] 13 23 33
##
## [[6]]
## [1] 14 24 34
##
## [[7]]
## [1] 15 25 35
##
## [[8]]
## [1] 16 26 36
##
## [[9]]
## [1] 17 27 37
##
## [[10]]
## [1] 18 28 38
##
## [[11]]
## [1] 20 30
##
## [[12]]
## [1] 40
##
## [[13]]
## [1] 41
##
## [[14]]
## [1] 20 30
##
## [[15]]
## [1] 43
##
## [[16]]
## [1] 43
##
## [[17]]
## [1] 44
```

```
##
## [[18]]
## [1] 45
##
## [[19]]
## [1] 46
##
## [[20]]
## [1] 47
```

```
# Sweet animation. Plots a lot, so you have to call it manually!
for (state in 1:(N-8)) {
  plot(x=1:100, y=filtered[,state], type='l')

  Sys.sleep((1-state/N) * 0.7)
}
```

Solution:

```
set.seed(123)

States<-1:100
Symbols<-1:2 # 1=door

transProbs<-matrix(rep(0,length(States)*length(States)), nrow=length(States), ncol=length(States), byrow
for(i in 1:99){
  transProbs[i,i]<-.1
  transProbs[i,i+1]<-.9
}

emissionProbs<-matrix(rep(0,length(States)*length(Symbols)), nrow=length(States), ncol=length(Symbols),
for(i in States){
  if(i %in% c(10,11,12,20,21,22,30,31,32)){
    emissionProbs[i,1]<-.9
    emissionProbs[i,2]<-.1
  }
  else{
    emissionProbs[i,1]<-.1
    emissionProbs[i,2]<-.9
  }
}

startProbs<-rep(1/100,100)
hmm<-initHMM(States,Symbols,startProbs,transProbs,emissionProbs)

# If the robot observes a door, it can be in front of any of the three doors. If it then observes a lon
# sequence of non-doors, then it know that it was in front of the third door.

obs<-c(1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2)
pt<-prop.table(exp(forward(hmm,obs)),2)

which.maxima<-function(x){ # This function is needed since which.max only returns the first maximum.
  return(which(x==max(x)))
}
```

```
apply(pt,2,which.maxima)
```

```
## $`1`
## 10 11 12 20 21 22 30 31 32
## 10 11 12 20 21 22 30 31 32
##
## $`2`
## 11 12 21 22 31 32
## 11 12 21 22 31 32
##
## $`3`
## 12 22 32
## 12 22 32
##
## $`4`
## 13 23 33
## 13 23 33
##
## $`5`
## 14 24 34
## 14 24 34
##
## $`6`
## 15 25 35
## 15 25 35
##
## $`7`
## 16 26 36
## 16 26 36
##
## $`8`
## 17 27 37
## 17 27 37
##
## $`9`
## 18 28 38
## 18 28 38
##
## $`10`
## 19 29 39
## 19 29 39
##
## $`11`
## 40
## 40
##
## $`12`
## 41
## 41
##
## $`13`
## 42
## 42
##
```

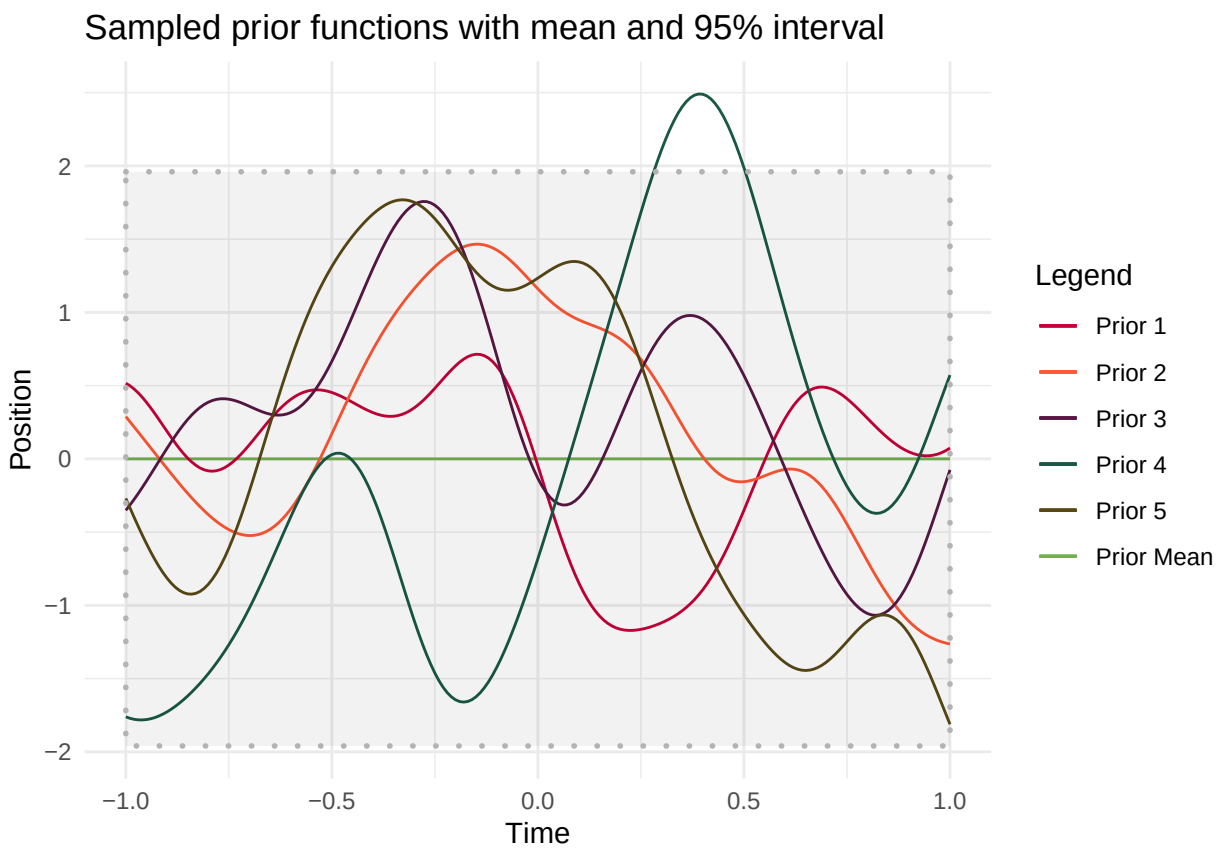
```
## $`14`
## 42
## 42
##
## $`15`
## 43
## 43
##
## $`16`
## 44
## 44
##
## $`17`
## 45
## 45
```

3 Gaussian Processes

3.1 A Priori

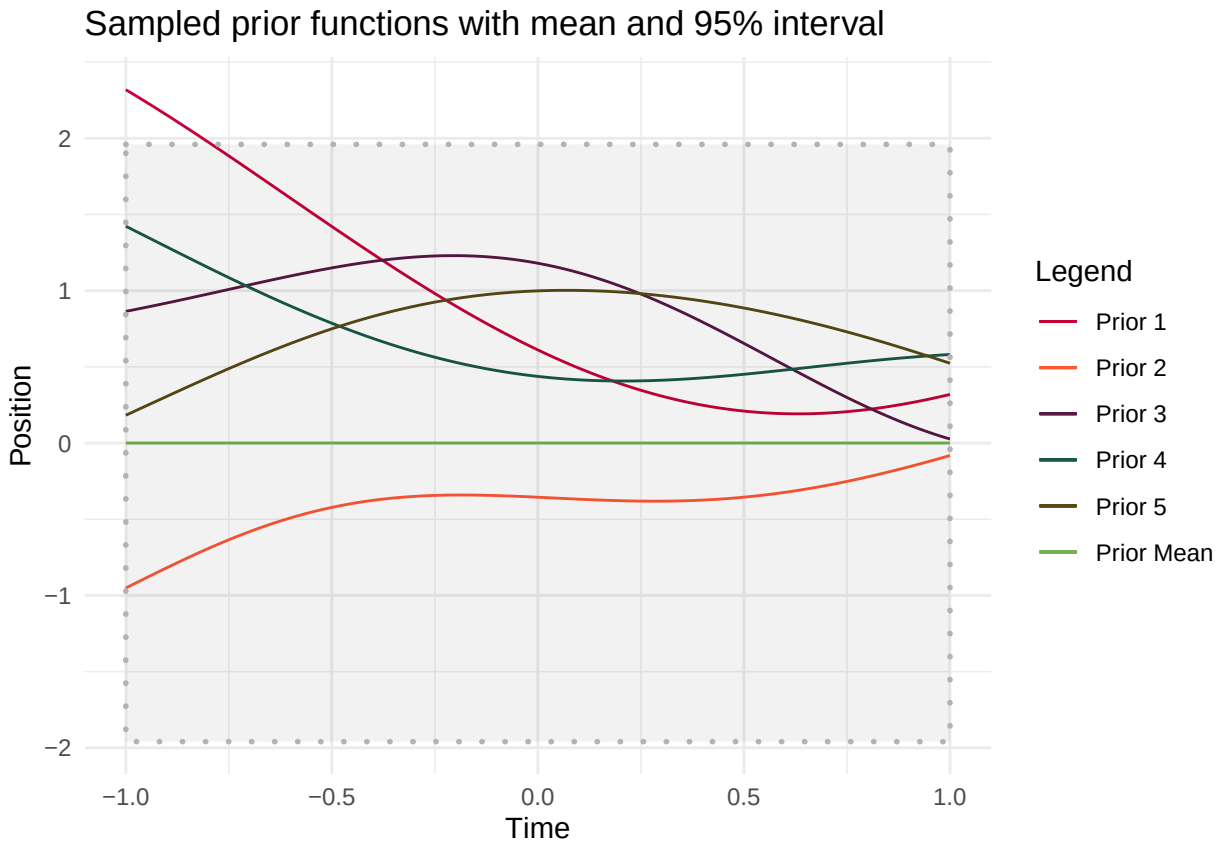
```
# From KernelCode.R:
# Squared exponential, k
k <- function(sigmaf = 1, ell = 1)
{
  rval <- function(x, y = NULL)
  {
    r = sqrt(crossprod(x-y))
    return(sigmaf^2*exp(-r^2/(2*ell^2)))
  }
  class(rval) <- "kernel"
  return(rval)
}

#' cov_kernel
#'
#' @param X
#' @param kernel A kernel functions.
#' @param ... Parameters for the kernel function.
#'
#' @return Returns the covariance matrix where the kernel is being applied.
cov_kernel = function(X, Y, kernel, ...) {
  if (!is.vector(X) || !is.vector(Y))
    stop("X and Y must be vectors.")

  K = matrix(NA, nrow = length(X), ncol = length(Y))

  for (i in 1:length(X)) {
    for (j in 1:length(Y)) {
      K[i, j] = kernel(X[i], Y[j], ...)
    }
  }
```

```
  return(K)
}

sigma_f = 1
sigma_f_sq = 1

ellOne = 0.2
ellTwo = 1

xGrid = seq(-1, 1, by=0.01)

myKernelOne = k(sigma_f, ellOne)

priorCovariance = cov_kernel(xGrid, xGrid, myKernelOne)
priorMean = rep(0, nrow(priorCovariance))

nFunctions = 5

functions = rmvnorm(nFunctions, mean = priorMean, sigma = priorCovariance)

ggplot() +
  geom_line(aes(x = xGrid, y = priorMean, colour = "Prior Mean")) +
  geom_line(aes(x = xGrid, y = functions[1,], colour = "Prior 1")) +
  geom_line(aes(x = xGrid, y = functions[2,], colour = "Prior 2")) +
  geom_line(aes(x = xGrid, y = functions[3,], colour = "Prior 3")) +
  geom_line(aes(x = xGrid, y = functions[4,], colour = "Prior 4")) +
  geom_line(aes(x = xGrid, y = functions[5,], colour = "Prior 5")) +
  geom_ribbon(aes(x = xGrid,
                  ymin = priorMean - 1.96 * diag(sqrt(priorCovariance)),
                  ymax = priorMean + 1.96 * diag(sqrt(priorCovariance))),
              alpha=0.05,
              linetype=3,
              colour="grey70",
              size=1,
              fill="black") +
  labs(title = "Sampled prior functions with mean and 95% interval",
    y = "Position", x = "Time", color = "Legend") +
  scale_color_manual(values = c("#C70039", "#FF5733", "#581845",
                                "#185845", "#584815", "#71AF50")) +
  theme_minimal()
```




```
myKernelTwo = k(sigma_f, ellTwo)

priorCovariance = cov_kernel(xGrid, xGrid, myKernelTwo)
priorMean = rep(0, nrow(priorCovariance))

nFunctions = 5

functions = rmvnorm(nFunctions, mean = priorMean, sigma = priorCovariance)

ggplot() +
  geom_line(aes(x = xGrid, y = priorMean, colour = "Prior Mean")) +
  geom_line(aes(x = xGrid, y = functions[1,], colour = "Prior 1")) +
  geom_line(aes(x = xGrid, y = functions[2,], colour = "Prior 2")) +
  geom_line(aes(x = xGrid, y = functions[3,], colour = "Prior 3")) +
  geom_line(aes(x = xGrid, y = functions[4,], colour = "Prior 4")) +
  geom_line(aes(x = xGrid, y = functions[5,], colour = "Prior 5")) +
  geom_ribbon(aes(x = xGrid,
                  ymin = priorMean - 1.96 * diag(sqrt(priorCovariance)),
                  ymax = priorMean + 1.96 * diag(sqrt(priorCovariance))),
              alpha=0.05,
              linetype=3,
              colour="grey70",
              size=1,
              fill="black") +
  labs(title = "Sampled prior functions with mean and 95% interval",
    y = "Position", x = "Time", color = "Legend") +
  scale_color_manual(values = c("#C70039", "#FF5733", "#581845",
```

```
                                "#185845", "#584815", "#71AF50")) +
  theme_minimal()
```



Correlations: The correlation between the function at $x = $0 and $x = 0.1$ is much higher than between $x = 0$ and $x = 0.5$, since the latter points are more distant. Thus, the correlation decays with distance in x-space. This decay is much more rapid when $\ell = 0.2$ than when $\ell = 1$. This is also visible in the simulations where realized functions are much more smooth when $\ell = 1$.

```
# ell = 0.2
myKernelOne(0, 0.1) # Note: here correlation=covariance since sigmaf = 1
```

```
##           [,1]
## [1,] 0.8824969
```

```
myKernelOne(0, 0.5)
```

```
##            [,1]
## [1,] 0.04393693
```

```
# ell = 1
myKernelTwo(0, 0.1) # Note: here correlation=covariance since sigmaf = 1
```

```
##           [,1]
## [1,] 0.9950125
```

```
myKernelTwo(0, 0.5)
```

```
##           [,1]
## [1,] 0.8824969
```

3.2 Posterior

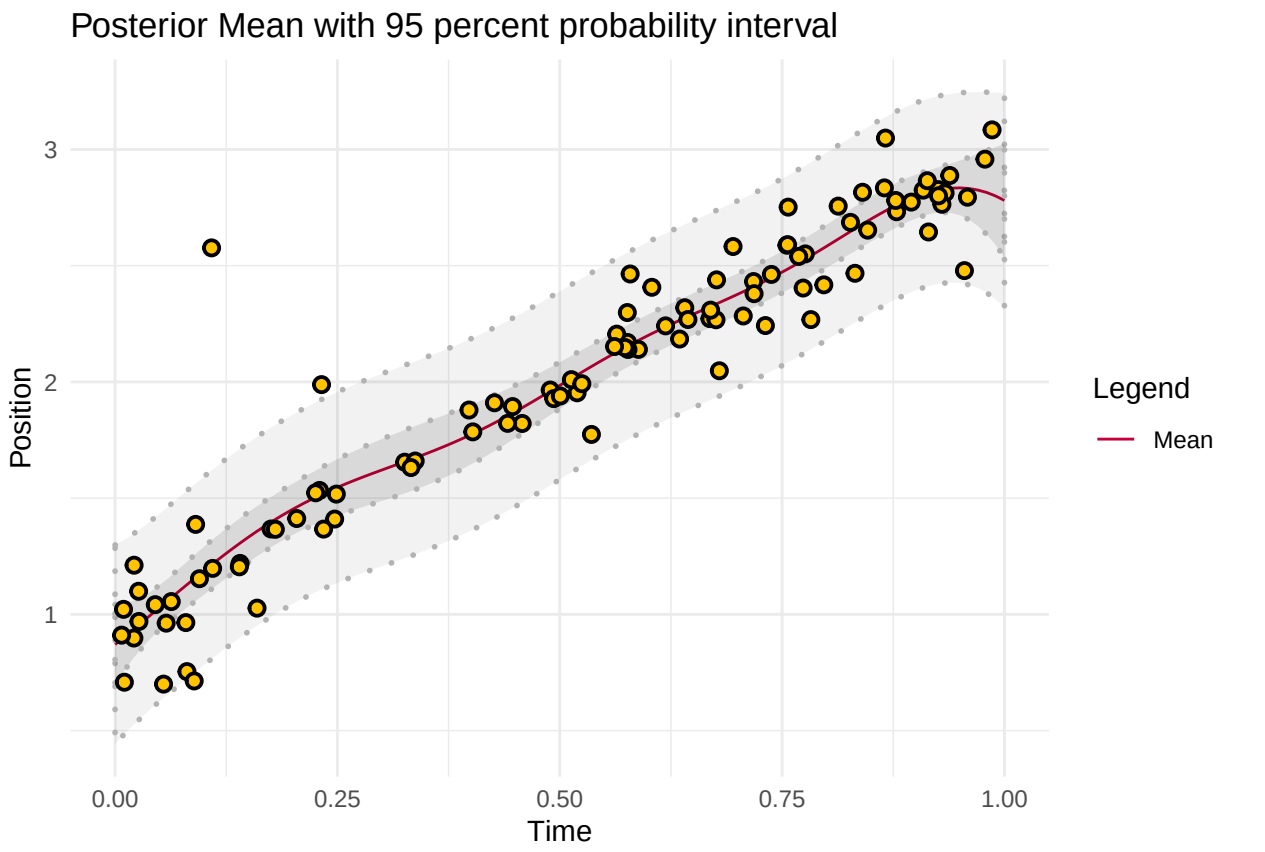
```
load("GPdata.RData")

#' posteriorGP
#'
#' @param X Vector of training inputs.
#' @param Y Vector of training targets/outputs.
#' @param XStar Vector of inputs where the posterior distribution is evaluated,
#'               i.e XStar.
#' @param sigmaNoise Noise standard deviation sigma_n.
#' @param kernel A kernel function.
#' @param ... Parameters for the kernel function.
#'
#' @return Vector with the posterior mean and variance of f evaluated on set XStar
posteriorGP = function(X, Y, XStar, sigmaNoise, kernel, ...) {

  K = t(cov_kernel(X, X, kernel, ...))
  L = t(chol(K + diag(sigmaNoise^2, length(X))))

  # Predictive Mean
  alpha = solve(t(L), solve(L, Y))
  K_star = cov_kernel(X, XStar, kernel, ...)
  f_star = t(K_star) %*% alpha

  # Predictive Variance
  V = solve(L, K_star)
  V_f_star = cov_kernel(XStar, XStar, kernel, ...) - t(V) %*% V

  # Log Marginal Likelihood
  n = length(X)
  lml = -0.5 %*% t(Y) %*% alpha - sum(log(diag(L))) - (n/2) * log(2*pi)

  gp_object = list(mean=f_star,
                   variance=V_f_star,
                   lml = lml,
                   X=X, Y=Y,
                   XStar=XStar,
                   sigmaNoise=sigmaNoise)

  class(gp_object) = "gp"
  return(gp_object)
}

#' plot.gp
#'
#' @param gp An Gaussian Process ("gp") object.
#' @param direct_plot TRUE by default. Determines if to plot directly or to
#'                    return the plot.
#'
#' @return If direct_plot is set to TRUE, the plot object.
plot.gp = function(gp, direct_plot=TRUE) {
  df = data.frame(x = gp$XStar,
```

```
                  y = gp$mean,
                  y_upper_prob = gp$mean + 1.96*sqrt(diag(gp$variance)),
                  y_lower_prob = gp$mean - 1.96*sqrt(diag(gp$variance)),
                  y_upper_pred = gp$mean + 1.96*sqrt(diag(gp$variance) +
                                                     gp$sigmaNoise^2),
                  y_lower_pred = gp$mean - 1.96*sqrt(diag(gp$variance) +
                                                     gp$sigmaNoise^2))

  points = data.frame(x = gp$X, y = gp$Y)

  p = ggplot() +
    geom_line(aes(x = df$x, y = df$y, colour = "Mean")) +
    geom_ribbon(aes(x = df$x, ymin = df$y_lower_pred, ymax = df$y_upper_pred),
                    alpha=0.05,
                    linetype=3,
                    colour="grey70",
                    size=1,
                    fill="black") +
    geom_ribbon(aes(x = df$x, ymin = df$y_lower_prob, ymax = df$y_upper_prob),
                    alpha=0.1,
                    linetype=3,
                    colour="grey70",
                    size=1,
                    fill="black") +
    geom_point(aes(x = points$x, y = points$y), color = "black",
                fill = "#FFC300", shape = 21, size = 2, stroke = 1) +
    labs(title = "Posterior Mean with 95 percent probability interval",
          y = "Position", x = "Time", color = "Legend") +
    scale_color_manual(values = c("#C70039", "#FF5733", "#581845")) +
    theme_minimal()

  if (!direct_plot)
    return(p)

  print(p)
}
```

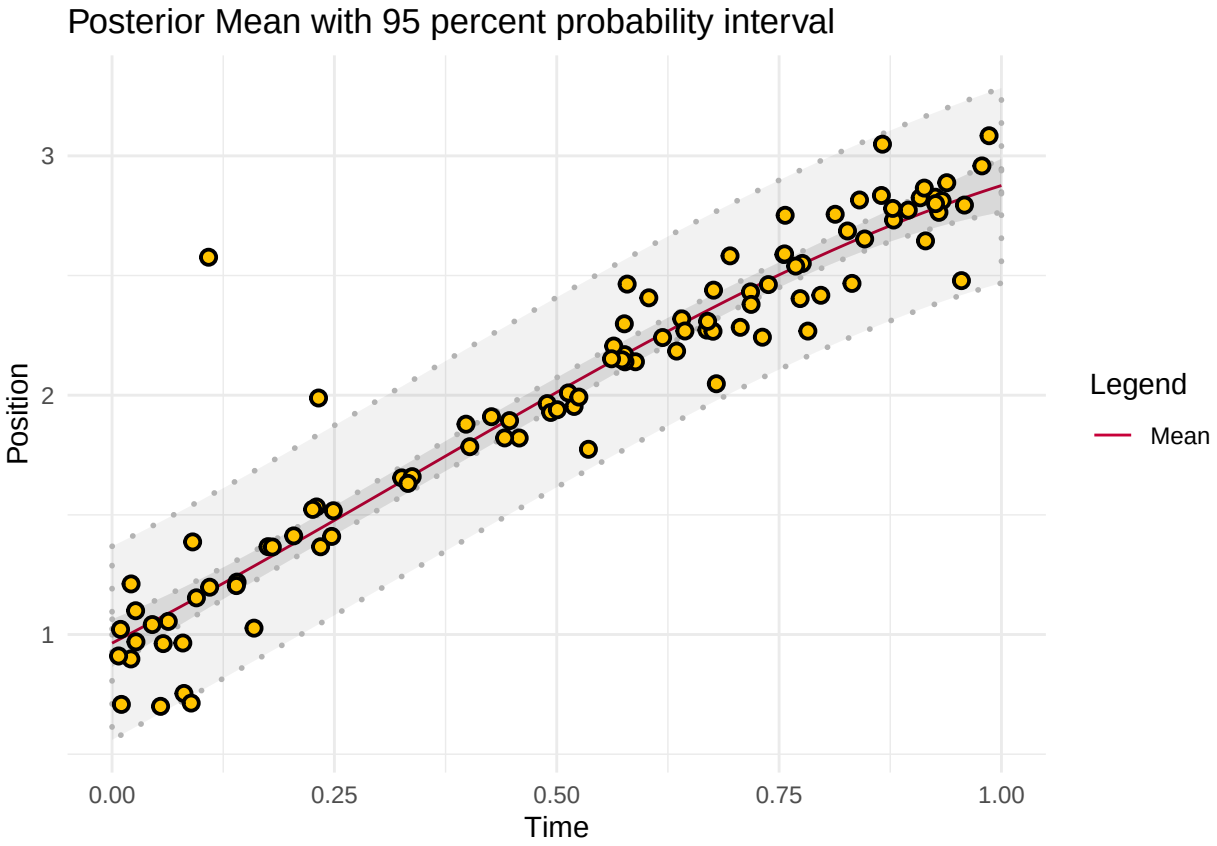
```
xGrid = seq(0, 1, by=0.01)

sigmaNoise = 0.2
sigma_f = 1

ellThree = 0.2
ellFour = 1

myKernelThree = k(sigma_f, ellThree)
myKernelFour = k(sigma_f, ellFour)

# Maybe in the exam he actually wants us to use the built-in function
# Then we should definitely do that, makes the plotting annoying though.
# GPfit = gausspr(x, y, kernel = kernelFunc, var = sigmaNoise^2)

# Also he wants us to use the custom built gp_covariance function...
# Pay attention to that
```

```
gp_post_0_2 = posteriorGP(x, y, xGrid, sigmaNoise, myKernelThree)
gp_post_1 = posteriorGP(x, y, xGrid, sigmaNoise, myKernelFour)
```

```
plot(gp_post_0_2)
```




```
plot(gp_post_1)
```



Question: Explain the difference between the results from ii) and iii).

Answer:

- ii) is about the uncertainty of the function f, which is the MEAN of y
- iii) is about the uncertainty of individual y values. They are uncertain for two reasons: you don't know f at the test point, and you don't know the error (epsilon) that will hit this individual observation.

Question: Discuss the differences in results from using the two length scales.

Answer: Shorter length scale gives less smooth f. We are overfitting the data.

Answer: Longer length scale gives more smoothness.

Question: Do you think a GP with a squared exponential kernel is a good model for this data? If not, why?

Answer: One would have to experiment with other length scales, or estimate the length scales (see question 3c), but this is not likely to help here. The issue is that the data seems to be have different smoothness for small x than it has for large x (where the function seems much more flat) The solution is probably to have different length scales for different x.

3.3 Bayesian View

For full points here you should mention EITHER of the following two approaches:

1. The marginal likelihood can be used to select optimal hyperparameters and also the noise variance. We can optimize the log marginal likelihood with respect to the hyperparameters. In Gaussian Process Regression the marginal likelihood is availble in closed form (a formula).

2. We can use sampling methods (e.g. MCMC) to sample from the marginal posterior of the hyperparameters. We need a prior $p(\theta)$ for the hyperparameter and then Bayes rule gives the marginal posterior $p(\theta|\text{data}) \propto p(\text{data}|\theta)p(\theta)$ $wherep(\text{data}|\theta)$ is the marginal likelihood (f has been integrated out).

If the noise variance is unknown, we can treat like any of the kernel hyperparameters and infer the noise variance jointly with the length scale and the prior variance σ_f.

4 Source Code

```
set.seed(12345)
library(bnlearn)
library(gRain)
library(caret)
library(HMM)
library(mvtnorm)
library(reshape2)
#if (!requireNamespace("BiocManager", quietly = TRUE))
#    install.packages("BiocManager")
#BiocManager::install("RBGL")
knitr::opts_chunk$set(echo = TRUE)
knitr::opts_chunk$set(echo = TRUE)

# Learning the Network
set.seed(123)
data("asia")
bayesian_network = hc(asia,restart=10,score="aic",iss=10)

# Learning the Parameters
bayesian_network_fit = bn.fit(bayesian_network, asia)
# Compile for faster computations
bayesian_network_grain = compile(as.grain(bayesian_network_fit))

plot(bayesian_network)


# Case: E is independent from L | S, B

case1 = setFinding(bayesian_network_grain, nodes=c("S", "B", "E"),states=c("yes","yes", "no"))
querygrain(case1, c("D"))

case2 = setFinding(bayesian_network_grain, nodes=c("S", "B", "E"),states=c("no","yes", "no"))
querygrain(case2, c("D"))


# Learn a BN from the Asia dataset, find a separation statement (e.g. B _|_ E | S, T) and, then, check
# it corresponds to a statistical independence.

# SOLUTION
set.seed(123)
data("asia")
hc3<-hc(asia,restart=10,score="bde",iss=10)
plot(hc3)
```

```
hc4<-bn.fit(hc3,asia,method="bayes")
hc5<-as.grain(hc4)
hc6<-compile(hc5)
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("yes","yes","yes"))
querygrain(hc7,c("B"))
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("yes","yes","no"))
querygrain(hc7,c("B"))
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("yes","no","yes"))
querygrain(hc7,c("B"))
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("yes","no","no"))
querygrain(hc7,c("B"))
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("no","yes","yes"))
querygrain(hc7,c("B"))
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("no","yes","no"))
querygrain(hc7,c("B"))
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("no","no","yes"))
querygrain(hc7,c("B"))
hc7<-setFinding(hc6,nodes=c("S","T","E"),states=c("no","no","no"))
querygrain(hc7,c("B"))
#B _|_ E | S, T


f = function(graph) {
  return(!is.character(all.equal(cpdag(graph), graph)))
}

N = 50000

# Randomly sample graphs
graphs = random.graph(c("A", "B", "C", "D", "E"), num = N, method="melancon")
unique_graphs = unique(graphs)

length(unique_graphs)/sum(sapply(unique_graphs, f))



set.seed(123)
N = 100

# Defining States Z1, Z2, ..., ZN
states = paste(rep("Z", N), 1:N, sep = "")

# Defining Symbols
symbols = c("Door", "Corridor")

# Starting Probabilities
startProbs = rep(1/N, N)

# Transition Probabilities
transProbs = matrix(0, ncol = N, nrow = N)
diag(transProbs) = 0.1
diag(transProbs[,-1]) = 0.9
transProbs[N, N] = 1.0
```

```
# Emission Probabilities
emissionProbs = matrix(NA, ncol = 2, nrow = N)
emissionProbs[,1] = 0.1
emissionProbs[,2] = 0.9
emissionProbs[10,] = c(0.9, 0.1)
emissionProbs[11,] = c(0.9, 0.1)
emissionProbs[12,] = c(0.9, 0.1)
emissionProbs[20,] = c(0.9, 0.1)
emissionProbs[21,] = c(0.9, 0.1)
emissionProbs[22,] = c(0.9, 0.1)
emissionProbs[30,] = c(0.9, 0.1)
emissionProbs[31,] = c(0.9, 0.1)
emissionProbs[32,] = c(0.9, 0.1)

robot_hmm = initHMM(States = states,
                    Symbols = symbols,
                    startProbs = startProbs,
                    transProbs = transProbs,
                    emissionProbs = emissionProbs)


nSim = 100
simulatedStates = simHMM(robot_hmm, nSim)
simulatedStates


custom_forward = function(hmm, observations) {

  Z = matrix(NA, ncol=length(hmm$States), nrow=length(observations))

  Z[1,] = hmm$emissionProbs[, observations[1]] * hmm$startProbs

  for (t in 2:length(observations)) {
    Z[t, ] = hmm$emissionProbs[, observations[t]] * (Z[t-1,] %*% hmm$transProbs)
  }

  return(t(Z))
}

start_time = Sys.time()
alpha = exp(forward(robot_hmm, simulatedStates$observation))
end_time = Sys.time()

builtInForward = end_time - start_time

start_time = Sys.time()
alpha = exp(custom_forward(robot_hmm, simulatedStates$observation))
end_time = Sys.time()

customForward = end_time - start_time

c(builtInForward, customForward)
```

```
filtered = prop.table(alpha, 2)

which.maxima = function(x){
  return(which(x==max(x)))
}

# For some reason we get werid behaviour at t=95. The system should be sure it
# is in the last state, shouldn't it?
apply(filtered, 2 ,which.maxima)[1:20]



# Sweet animation. Plots a lot, so you have to call it manually!
for (state in 1:(N-8)) {
  plot(x=1:100, y=filtered[,state], type='l')

  Sys.sleep((1-state/N) * 0.7)
}


set.seed(123)

States<-1:100
Symbols<-1:2 # 1=door

transProbs<-matrix(rep(0,length(States)*length(States)), nrow=length(States), ncol=length(States), byrow
for(i in 1:99){
  transProbs[i,i]<-.1
  transProbs[i,i+1]<-.9
}

emissionProbs<-matrix(rep(0,length(States)*length(Symbols)), nrow=length(States), ncol=length(Symbols),
for(i in States){
  if(i %in% c(10,11,12,20,21,22,30,31,32)){
    emissionProbs[i,1]<-.9
    emissionProbs[i,2]<-.1
  }
  else{
    emissionProbs[i,1]<-.1
    emissionProbs[i,2]<-.9
  }
}

startProbs<-rep(1/100,100)
hmm<-initHMM(States,Symbols,startProbs,transProbs,emissionProbs)

# If the robot observes a door, it can be in front of any of the three doors. If it then observes a lon
# sequence of non-doors, then it know that it was in front of the third door.

obs<-c(1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2)
pt<-prop.table(exp(forward(hmm,obs)),2)

which.maxima<-function(x){ # This function is needed since which.max only returns the first maximum.
```

```
  return(which(x==max(x)))
}

apply(pt,2,which.maxima)


# From KernelCode.R:
# Squared exponential, k
k <- function(sigmaf = 1, ell = 1)
{
  rval <- function(x, y = NULL)
  {
    r = sqrt(crossprod(x-y))
    return(sigmaf^2*exp(-r^2/(2*ell^2)))
  }
  class(rval) <- "kernel"
  return(rval)
}

#' cov_kernel
#'
#' @param X
#' @param kernel A kernel functions.
#' @param ... Parameters for the kernel function.
#'
#' @return Returns the covariance matrix where the kernel is being applied.
cov_kernel = function(X, Y, kernel, ...) {
  if (!is.vector(X) || !is.vector(Y))
    stop("X and Y must be vectors.")

  K = matrix(NA, nrow = length(X), ncol = length(Y))

  for (i in 1:length(X)) {
    for (j in 1:length(Y)) {
      K[i, j] = kernel(X[i], Y[j], ...)
    }
  }

  return(K)
}

sigma_f = 1
sigma_f_sq = 1

ellOne = 0.2
ellTwo = 1

xGrid = seq(-1, 1, by=0.01)

myKernelOne = k(sigma_f, ellOne)

priorCovariance = cov_kernel(xGrid, xGrid, myKernelOne)
priorMean = rep(0, nrow(priorCovariance))
```

```
nFunctions = 5

functions = rmvnorm(nFunctions, mean = priorMean, sigma = priorCovariance)

ggplot() +
  geom_line(aes(x = xGrid, y = priorMean, colour = "Prior Mean")) +
  geom_line(aes(x = xGrid, y = functions[1,], colour = "Prior 1")) +
  geom_line(aes(x = xGrid, y = functions[2,], colour = "Prior 2")) +
  geom_line(aes(x = xGrid, y = functions[3,], colour = "Prior 3")) +
  geom_line(aes(x = xGrid, y = functions[4,], colour = "Prior 4")) +
  geom_line(aes(x = xGrid, y = functions[5,], colour = "Prior 5")) +
  geom_ribbon(aes(x = xGrid,
                  ymin = priorMean - 1.96 * diag(sqrt(priorCovariance)),
                  ymax = priorMean + 1.96 * diag(sqrt(priorCovariance))),
              alpha=0.05,
              linetype=3,
              colour="grey70",
              size=1,
              fill="black") +
  labs(title = "Sampled prior functions with mean and 95% interval",
    y = "Position", x = "Time", color = "Legend") +
  scale_color_manual(values = c("#C70039", "#FF5733", "#581845",
                                "#185845", "#584815", "#71AF50")) +
  theme_minimal()



myKernelTwo = k(sigma_f, ellTwo)

priorCovariance = cov_kernel(xGrid, xGrid, myKernelTwo)
priorMean = rep(0, nrow(priorCovariance))

nFunctions = 5

functions = rmvnorm(nFunctions, mean = priorMean, sigma = priorCovariance)

ggplot() +
  geom_line(aes(x = xGrid, y = priorMean, colour = "Prior Mean")) +
  geom_line(aes(x = xGrid, y = functions[1,], colour = "Prior 1")) +
  geom_line(aes(x = xGrid, y = functions[2,], colour = "Prior 2")) +
  geom_line(aes(x = xGrid, y = functions[3,], colour = "Prior 3")) +
  geom_line(aes(x = xGrid, y = functions[4,], colour = "Prior 4")) +
  geom_line(aes(x = xGrid, y = functions[5,], colour = "Prior 5")) +
  geom_ribbon(aes(x = xGrid,
                  ymin = priorMean - 1.96 * diag(sqrt(priorCovariance)),
                  ymax = priorMean + 1.96 * diag(sqrt(priorCovariance))),
              alpha=0.05,
              linetype=3,
              colour="grey70",
              size=1,
              fill="black") +
  labs(title = "Sampled prior functions with mean and 95% interval",
    y = "Position", x = "Time", color = "Legend") +
```

```
  scale_color_manual(values = c("#C70039", "#FF5733", "#581845",
                                "#185845", "#584815", "#71AF50")) +
  theme_minimal()


# ell = 0.2
myKernelOne(0, 0.1) # Note: here correlation=covariance since sigmaf = 1
myKernelOne(0, 0.5)

# ell = 1
myKernelTwo(0, 0.1) # Note: here correlation=covariance since sigmaf = 1
myKernelTwo(0, 0.5)


load("GPdata.RData")

#' posteriorGP
#'
#' @param X Vector of training inputs.
#' @param Y Vector of training targets/outputs.
#' @param XStar Vector of inputs where the posterior distribution is evaluated,
#               i.e XStar.
#' @param sigmaNoise Noise standard deviation sigma_n.
#' @param kernel A kernel function.
#' @param ... Parameters for the kernel function.
#'
#' @return Vector with the posterior mean and variance of f evaluated on set XStar
posteriorGP = function(X, Y, XStar, sigmaNoise, kernel, ...) {

  K = t(cov_kernel(X, X, kernel, ...))
  L = t(chol(K + diag(sigmaNoise^2, length(X))))

  # Predictive Mean
  alpha = solve(t(L), solve(L, Y))
  K_star = cov_kernel(X, XStar, kernel, ...)
  f_star = t(K_star) %*% alpha

  # Predictive Variance
  V = solve(L, K_star)
  V_f_star = cov_kernel(XStar, XStar, kernel, ...) - t(V) %*% V

  # Log Marginal Likelihood
  n = length(X)
  lml = -0.5 %*% t(Y) %*% alpha - sum(log(diag(L))) - (n/2) * log(2*pi)

  gp_object = list(mean=f_star,
                   variance=V_f_star,
                   lml = lml,
                   X=X, Y=Y,
                   XStar=XStar,
                   sigmaNoise=sigmaNoise)

  class(gp_object) = "gp"
```

```
  return(gp_object)
}

#' plot.gp
#'
#' @param gp An Gaussian Process ("gp") object.
#' @param direct_plot TRUE by default. Determines if to plot directly or to
#'                    return the plot.
#'
#' @return If direct_plot is set to TRUE, the plot object.
plot.gp = function(gp, direct_plot=TRUE) {
  df = data.frame(x = gp$XStar,
                  y = gp$mean,
                  y_upper_prob = gp$mean + 1.96*sqrt(diag(gp$variance)),
                  y_lower_prob = gp$mean - 1.96*sqrt(diag(gp$variance)),
                  y_upper_pred = gp$mean + 1.96*sqrt(diag(gp$variance) +
                                                       gp$sigmaNoise^2),
                  y_lower_pred = gp$mean - 1.96*sqrt(diag(gp$variance) +
                                                       gp$sigmaNoise^2))

  points = data.frame(x = gp$X, y = gp$Y)

  p = ggplot() +
    geom_line(aes(x = df$x, y = df$y, colour = "Mean")) +
    geom_ribbon(aes(x = df$x, ymin = df$y_lower_pred, ymax = df$y_upper_pred),
                    alpha=0.05,
                    linetype=3,
                    colour="grey70",
                    size=1,
                    fill="black") +
    geom_ribbon(aes(x = df$x, ymin = df$y_lower_prob, ymax = df$y_upper_prob),
                    alpha=0.1,
                    linetype=3,
                    colour="grey70",
                    size=1,
                    fill="black") +
    geom_point(aes(x = points$x, y = points$y), color = "black",
                fill = "#FFC300", shape = 21, size = 2, stroke = 1) +
    labs(title = "Posterior Mean with 95 percent probability interval",
          y = "Position", x = "Time", color = "Legend") +
    scale_color_manual(values = c("#C70039", "#FF5733", "#581845")) +
    theme_minimal()

  if (!direct_plot)
    return(p)

  print(p)
}


xGrid = seq(0, 1, by=0.01)

sigmaNoise = 0.2
```

```
sigma_f = 1

ellThree = 0.2
ellFour = 1

myKernelThree = k(sigma_f, ellThree)
myKernelFour = k(sigma_f, ellFour)

# Maybe in the exam he actually wants us to use the built-in function
# Then we should definitely do that, makes the plotting annoying though.
# GPfit = gausspr(x, y, kernel = kernelFunc, var = sigmaNoise^2)

# Also he wants us to use the custom built gp_covariance function...
# Pay attention to that

gp_post_0_2 = posteriorGP(x, y, xGrid, sigmaNoise, myKernelThree)
gp_post_1 = posteriorGP(x, y, xGrid, sigmaNoise, myKernelFour)


plot(gp_post_0_2)


plot(gp_post_1)
```