Advanced Machine Learning - Exam 2018-08-29

Maximilian Pfundstein (maxpf364)

2019-10-14

Contents

1 Graphical Models

Importing the dataset.

```
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
```

1.1 Training a Bayesian Network

```
## Training the network
train_bayesian_network = function(structure = NULL,
                                  data = train,
                                  learning_algorithm = iamb,
                                  ...) {

  # Network
  if (is.null(structure)) {
    bayesian_network = learning_algorithm(data, ...)
  }
  else {
    bayesian_network = structure
  }

  # Parameters
  bayesian_network_fit = bn.fit(bayesian_network, data)
  bayesian_network_grain = compile(as.grain(bayesian_network_fit))

  return(list(bn_grain=bayesian_network_grain,
              bn_fit=bayesian_network_fit,
              bn_structure=bayesian_network))
}

bn = train_bayesian_network(structure = NULL, data = asia, learning_algorithm = hc)
```

Once we calculate the exact probabilites and the we use an approximation. The approximation will not necessarily add up to 1.

```
res_exact = querygrain(setEvidence(bn$bn_grain, c("X", "B"), c("yes", "yes")))$A
res_exact
```

```
## A
##     no    yes
## 0.9916 0.0084
```

```
res_approx_yes = cpquery(bn$bn_fit, event = (A == "yes"),
                     evidence = ((X == "yes") & (B == "yes")))
res_approx_yes
```

```
## [1] 0.005970149
```

```
res_approx_no = cpquery(bn$bn_fit, event = (A == "no"),
                     evidence = ((X == "no") & (B == "yes")))
res_approx_no
```

```
## [1] 0.9888393
```

1.2 Indepent Models

```
isEquivilant = function(graph) {
  res = all.equal(cpdag(graph), skeleton(graph))
  if (res == "TRUE") return(TRUE)
  return(FALSE)
}

random_networks = random.graph(c("a", "b", "c", "d", "e"), num=50000)

res = sapply(random_networks, isEquivilant)

sum(res)/length(res)
```

```
## [1] 0.31872
```

2 Hidden Markov Networks

```r
N = 10

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

# Defining Symbols S1, S2, ..., SN
symbols = paste(rep("S", N), 1:(N+1), sep = "")

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

# Transition Probabilities
transProbs = matrix(0, ncol = N, nrow = N)
# Staying in the current state with 0.5 probability is just die diagonal
diag(transProbs) = 0.5
# Moving to the next is also 0.5
diag(transProbs[,-1]) = 0.5
transProbs[10, 1] = 0.5

# Emission Probabilities
emissionProbs = matrix(0, ncol = N, nrow = N+1)

# 0.2 For i-2 to i+2
for (i in 1:N) {
  for (j in c(3:-1)) {
    emissionProbs[((i-j)%%N)+1,i] = 0.1
  }
}

emissionProbs[11,] = 0.5

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

```r
observations = c("S1", "S11", "S11", "S11")

# The library returns the probabilities logged, we we have to de-log
alpha = exp(forward(robot_hmm, observations))
beta = exp(backward(robot_hmm, observations))

# Smoothed
# Can either be done manually or using the function posterior (== smoothed) in
# this package
# AUTOMATIC
smoothed_automatically = posterior(robot_hmm, observations)
# MANUALLY (Instead of division prop.table would work as well)
smoothed_manually = alpha * beta / colSums(alpha * beta)
```

```
# Path
hmm_viterbi = viterbi(robot_hmm, observations)

# Print Smoothed
smoothed_automatically
```

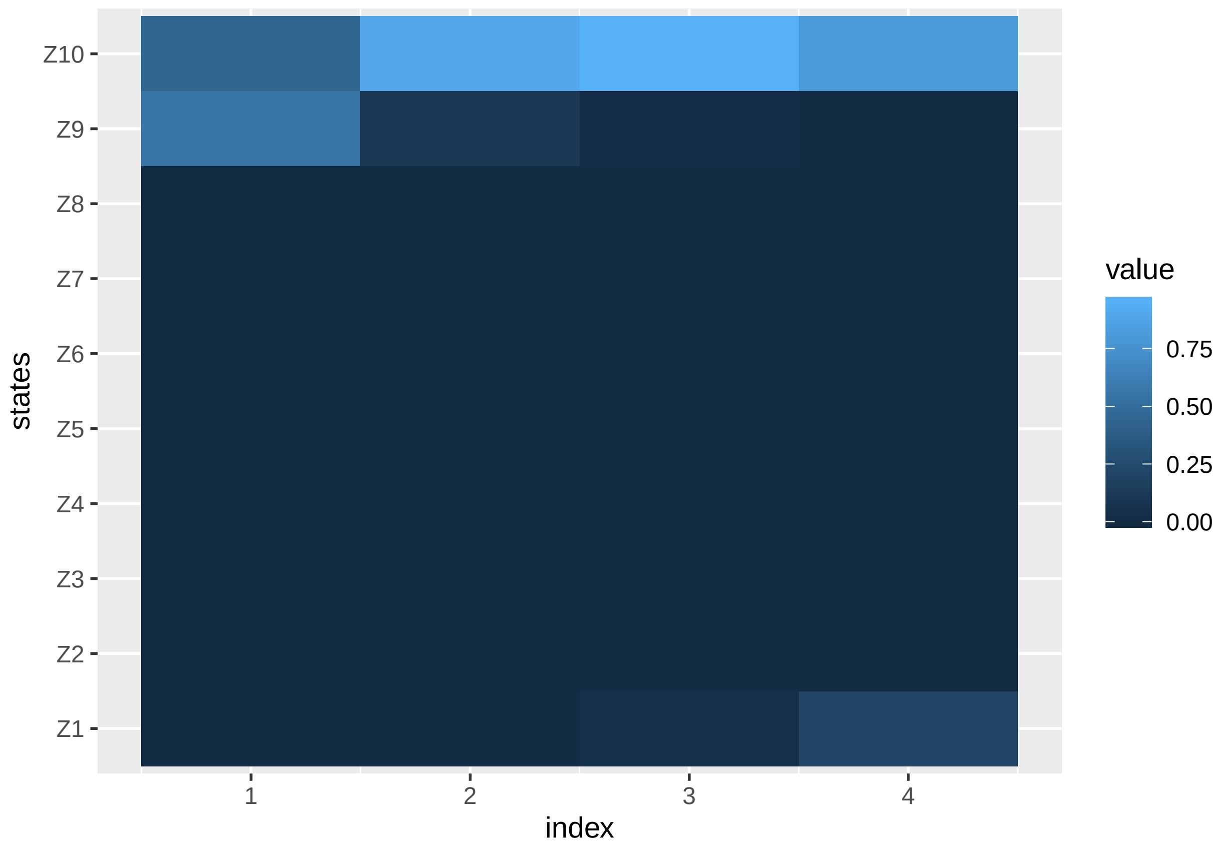
```
##       index
## states           1           2          3           4
##    Z1  0.002873563 0.005747126 0.03448276 0.192528736
##    Z2  0.000000000 0.000000000 0.00000000 0.000000000
##    Z3  0.000000000 0.000000000 0.00000000 0.000000000
##    Z4  0.000000000 0.000000000 0.00000000 0.000000000
##    Z5  0.000000000 0.000000000 0.00000000 0.000000000
##    Z6  0.000000000 0.000000000 0.00000000 0.000000000
##    Z7  0.000000000 0.000000000 0.00000000 0.000000000
##    Z8  0.000000000 0.000000000 0.00000000 0.000000000
##    Z9  0.548850575 0.103448276 0.01724138 0.002873563
##    Z10 0.448275862 0.890804598 0.94827586 0.804597701
```

```
smoothed_manually
```

```
##       index
## states           1           2          3           4
##    Z1  0.002873563 0.005747126 0.03448276 0.192528736
##    Z2  0.000000000 0.000000000 0.00000000 0.000000000
##    Z3  0.000000000 0.000000000 0.00000000 0.000000000
##    Z4  0.000000000 0.000000000 0.00000000 0.000000000
##    Z5  0.000000000 0.000000000 0.00000000 0.000000000
##    Z6  0.000000000 0.000000000 0.00000000 0.000000000
##    Z7  0.000000000 0.000000000 0.00000000 0.000000000
##    Z8  0.000000000 0.000000000 0.00000000 0.000000000
##    Z9  0.548850575 0.103448276 0.01724138 0.002873563
##    Z10 0.448275862 0.890804598 0.94827586 0.804597701
```

```
# Print Viterbi
hmm_viterbi
```

```
## [1] "Z9"  "Z10" "Z10" "Z10"
```



```
states = c("SS", "SR", "RS", "RR")
symbols = c("R", "S")
startProbs = c(0.25, 0.25, 0.25, 0.25)

transProbs = matrix(c(0.75, 0.25, 0, 0,
                      0, 0, 0.5, 0.5,
                      0.5, 0.5, 0, 0,
                      0, 0, 0.25, 0.75), nrow=4, ncol=4, byrow=TRUE)
colnames(transProbs) = states
rownames(transProbs) = states

emissionProbs = t(matrix(c(0.1, 0.9, 0.9, 0.1, 0.1, 0.9, 0.9, 0.1),
                         nrow=2, ncol=4, byrow=FALSE))
rownames(emissionProbs) = states
colnames(emissionProbs) = symbols

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

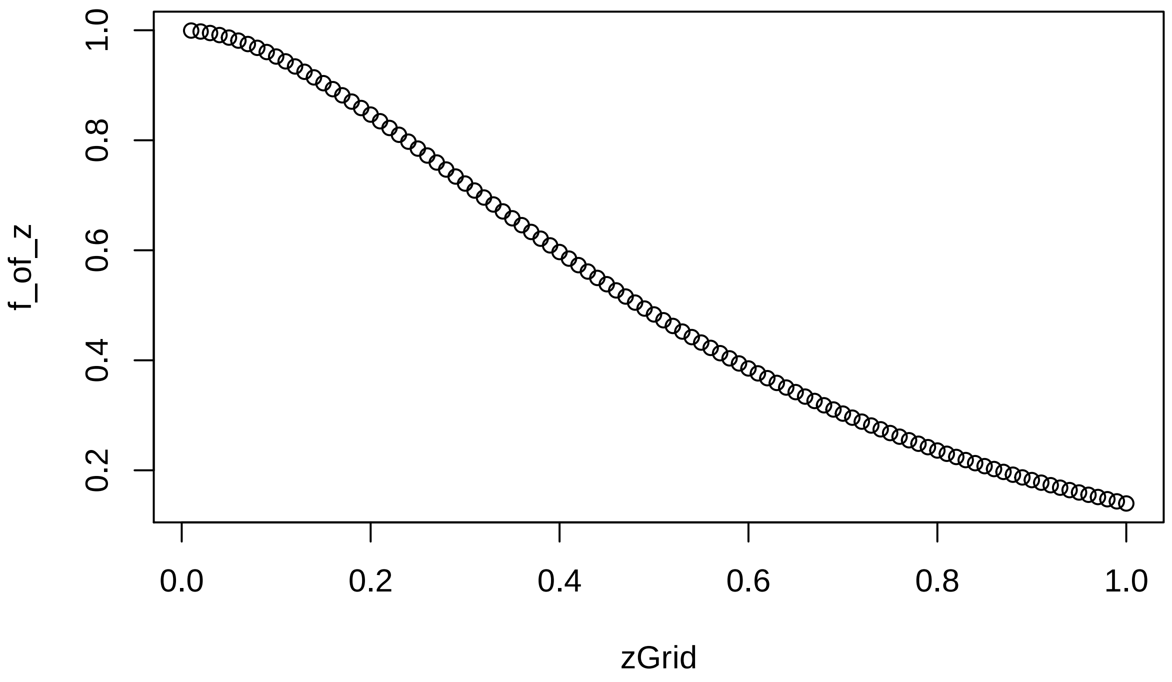
```
simulatedStates = simHMM(weather_hmm, 10)
simulatedStates
```

```
## $states
##  [1] "SS" "SR" "RS" "SR" "RS" "SR" "RR" "RR" "RR" "RR"
##
```

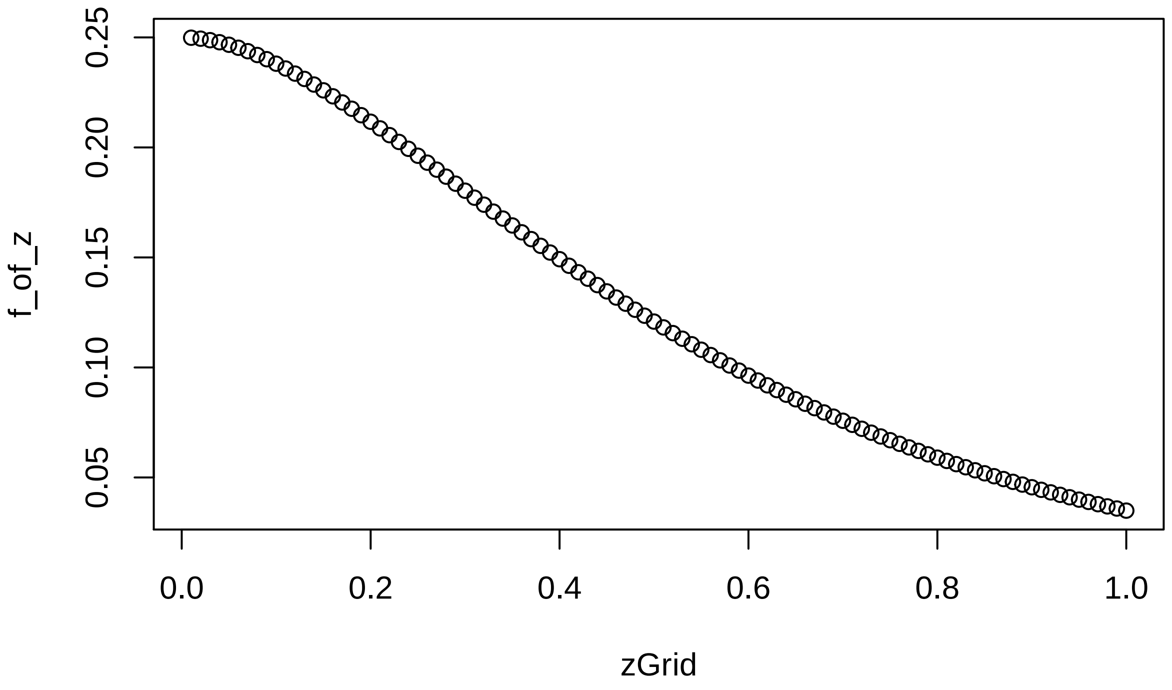
```
## $observation
##  [1] "S" "R" "S" "R" "S" "R" "R" "R" "R" "R"
```

3 Gaussian Processes

```r
# Matern32  kernel
k <- function(sigmaf = 1, ell = 1)
{
    rval <- function(x, y = NULL)
    {   r = sqrt(crossprod(x-y))
         return(sigmaf^2*(1+sqrt(3)*r/ell)*exp(-sqrt(3)*r/ell))
    }
    class(rval) <- "kernel"
    return(rval)
}

sigma_f_sq = 1
ell = 0.5

zGrid = seq(0.01, 1, 0.01)
f_of_z = vector(length = length(zGrid))

kernel = k(sigmaf = sigma_f_sq, ell = ell)

for (i in 1:length(zGrid)) {
  f_of_z[i] = kernel(0, zGrid[i])
}

plot(zGrid, f_of_z)
```



```r
sigma_f_sq = 0.5

kernel = k(sigmaf = sigma_f_sq, ell = ell)

for (i in 1:length(zGrid)) {
  f_of_z[i] = kernel(0, zGrid[i])
}

plot(zGrid, f_of_z)
```

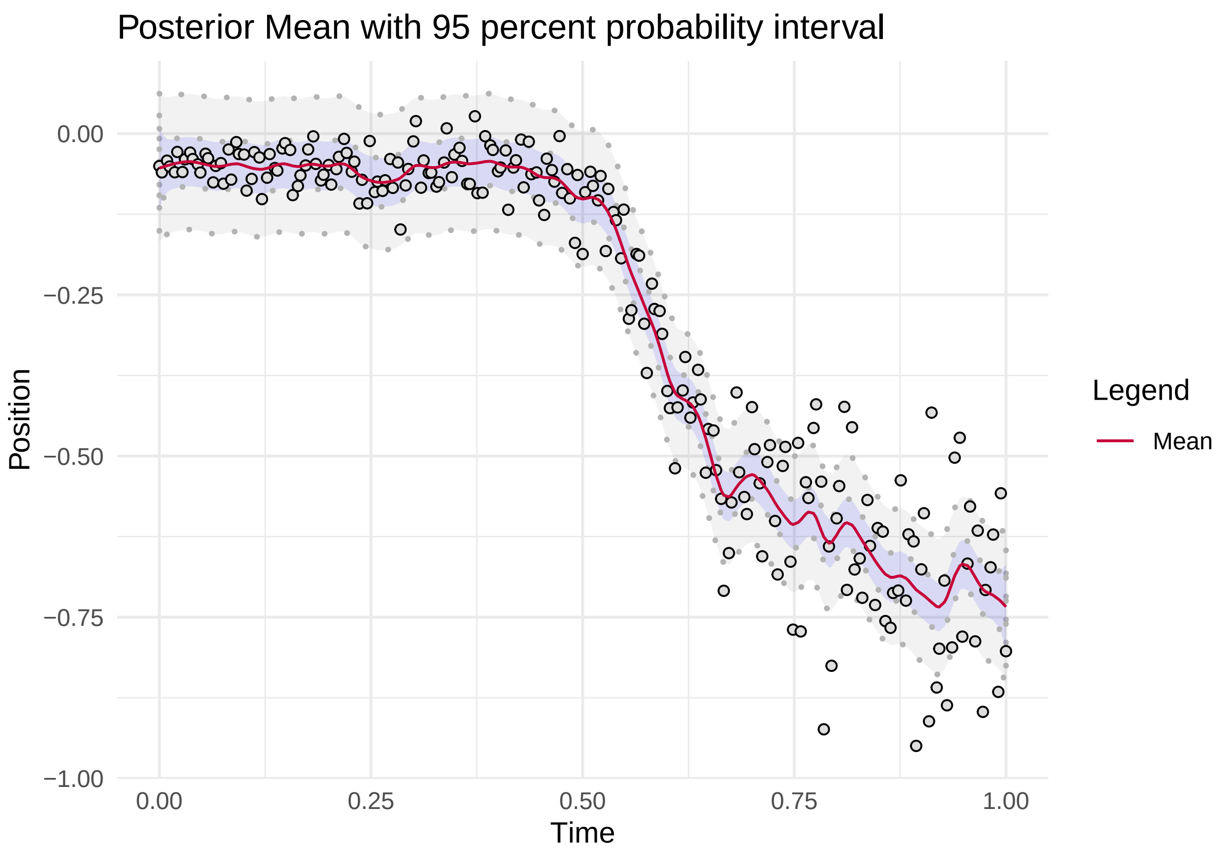


```r
#' gp_covariance
#'
#' @param x X, which will be scaled.
#' @param xss X_star, which will be scaled.
#'
#' @return The covariance matrix.
gp_covariance = function(x, xss, kernel, sigma_n) {
  # Copied from given scripts
  x = scale(x)
  xs = scale(xss)
  n = nrow(x)
  Kss = kernelMatrix(kernel = kernel, x = xs, y = xs)
  Kxx = kernelMatrix(kernel = kernel, x = x, y = x)
  Kxs = kernelMatrix(kernel = kernel, x = x, y = xs)
  Covf = Kss - t(Kxs) %*% solve(Kxx + sigma_n^2*diag(n), Kxs)
  return(Covf)
}
```

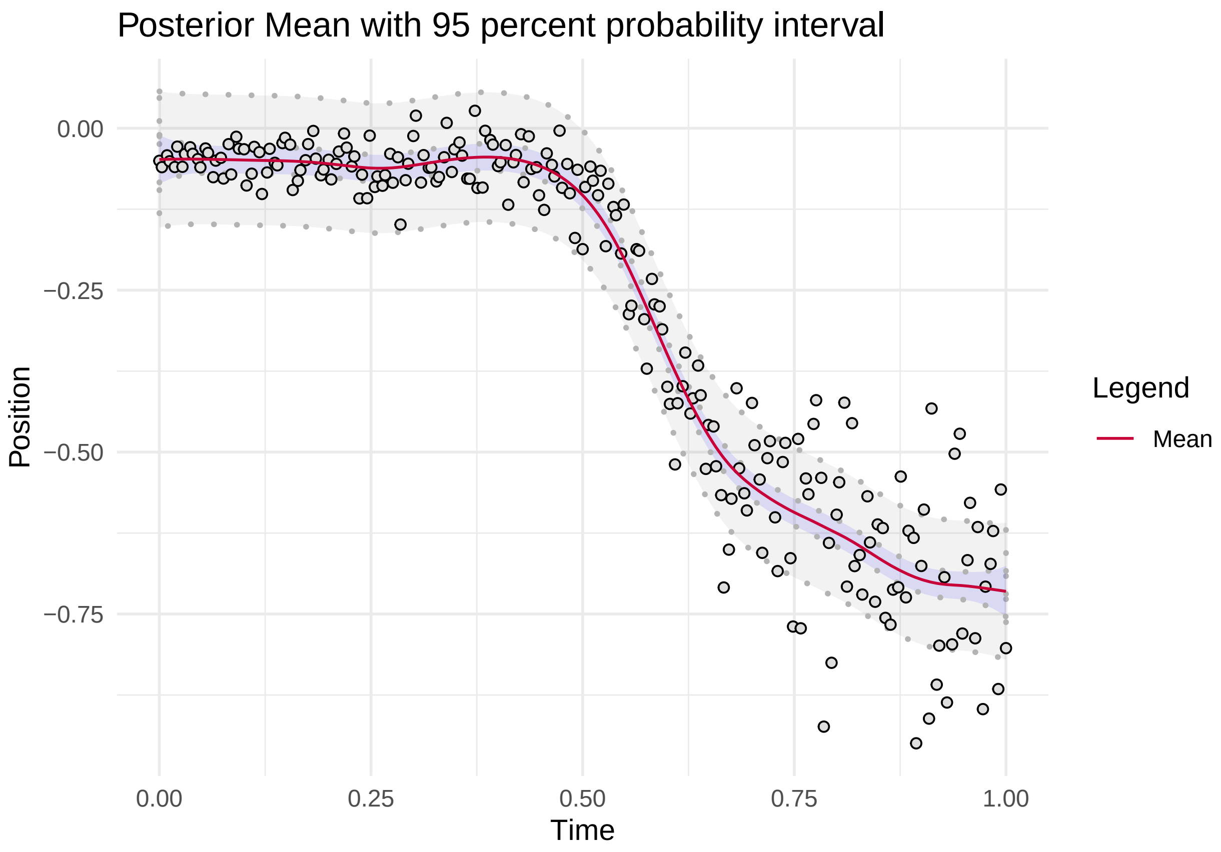
```r
library(kernlab)
```

```
##
## Attaching package: 'kernlab'
## The following object is masked from 'package:ggplot2':
##
##     alpha
```

```r
load("lidar.RData") # loading the data
sigmaNoise = 0.05
x = distance
y = logratio

# Set up the kernel function
kernelFunc <- k(sigmaf = 1, ell = 1)

GPfit = gausspr(x, y, kernel = kernelFunc, var = sigmaNoise^2)
meanPred = predict(GPfit, x)

varProb = diag(gp_covariance(x, x, kernel = kernelFunc, sigma_n = sigmaNoise))
varPred = varProb + sigmaNoise^2

df1 = data.frame(x,
                 y,
                 meanPred,
                 lowerProb = meanPred - 1.96 * sqrt(varProb),
                 higherProb = meanPred + 1.96 * sqrt(varProb),
                 lowerPred = meanPred - 1.96 * sqrt(varPred),
                 higherPred = meanPred + 1.96 * sqrt(varPred))

ggplot(df1) +
  geom_ribbon(aes(x = x, ymin = lowerPred, ymax = higherPred),
              alpha=0.05, linetype=3, colour="grey70", size=1, fill="black") +
  geom_ribbon(aes(x = x, ymin = lowerProb, ymax = higherProb),
              alpha=0.1, linetype=3, colour="grey70", size=1, fill="blue") +
  geom_point(aes(x = x, y = y), color = "black", fill = "#dedede", shape = 21) +
  geom_line(aes(x = x, y = meanPred, colour = "Mean")) +
  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()
```



```
# Set up the kernel function
kernelFunc <- k(sigmaf = 1, ell = 5)

GPfit = gausspr(x, y, kernel = kernelFunc, var = sigmaNoise^2)
meanPred = predict(GPfit, x)

varProb = diag(gp_covariance(x, x, kernel = kernelFunc, sigma_n = sigmaNoise))
varPred = varProb + sigmaNoise^2

df2 = data.frame(x,
                 y,
                 meanPred,
                 lowerProb = meanPred - 1.96 * sqrt(varProb),
                 higherProb = meanPred + 1.96 * sqrt(varProb),
                 lowerPred = meanPred - 1.96 * sqrt(varPred),
                 higherPred = meanPred + 1.96 * sqrt(varPred))

ggplot(df2) +
  geom_ribbon(aes(x = x, ymin = lowerPred, ymax = higherPred),
              alpha=0.05, linetype=3, colour="grey70", size=1, fill="black") +
  geom_ribbon(aes(x = x, ymin = lowerProb, ymax = higherProb),
              alpha=0.1, linetype=3, colour="grey70", size=1, fill="blue") +
  geom_point(aes(x = x, y = y), color = "black", fill = "#dedede", shape = 21) +
  geom_line(aes(x = x, y = meanPred, colour = "Mean")) +
  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()
```



3.1 c)

Ask domain experts, see which one works best, find a prior, use the data.

4 Sate Space Models

```
rtransition = function(x_t_1, sd=1) {
  return(rnorm(n = 1, x_t_1 + 1, sd = sd))
}

remission = function(x_t, sd=5) {
  return(rnorm(n = 1, mean = x_t, sd = sd))
}

demission =function(z_t, x_t, sd=5) {
  return(dnorm(z_t, mean = x_t, sd = sd))
}

rinit = function(n) {
  return(rnorm(n = n, mean = 50, sd = 10))
}
```

```r
sample_observations = function(n) {
  # First observation
  states = vector(length = n) # X
  observations = vector(length = n) # Z
  states[1] = rinit(1)
  observations[1] = remission(1, states[1])

  for(i in 2:n) {
    states[i] = rtransition(states[i-1])
    observations[i] = remission(states[i])
  }

  return(data.frame(states = states, obs = observations))
}
```

4.1 Kalman Filter

```r
kalman_filter = function(mu0, Sigma0, X, Z, R, Q, A=1, C=1) {

  Sigma0 = Sigma0^2
  R = R^2
  Q = R^2

  # Variables
  mu = vector(length = length(X))
  Sigma = vector(length = length(X))
  K = vector(length = length(X))
  E = vector(length = length(X))

  # Init
  mu[1] = mu0
  Sigma[1] = Sigma0

  # Has to start at 2 as R is stupid and doesn't like 0 indeces
  for (t in 2:length(X)) {
    mu_bar = mu[t-1] + 1
    Sigma_bar = A * Sigma[t-1] * A + R # A = 1
    K[t] = Sigma_bar * C / (C * Sigma_bar * C + Q) # C = 1
    mu[t] = mu_bar + K[t] * (X[t] - C * mu_bar)
    Sigma[t] = (1 - K[t] * C) * Sigma_bar
    E[t] = abs(X[t] - mu[t])
  }

  mu = mu[2:length(mu)]
  Sigma = Sigma[2:length(Sigma)]
  K = K[2:length(K)]
  E = E[2:length(E)]

  return(list(mu=mu, Sigma=Sigma, K=K, A=A, C=C, R=R, Q=Q, X=X, E=E))
}
```

```r
sample = sample_observations(10000)
Z = sample$states
X = sample$obs
R = 1 # transition
Q = 5 # semision

mu0 = 50
Sigma0 = 10

res = kalman_filter(mu0, Sigma0, X, Z, R, Q)

mean(res$E)
```

```
## [1] 1.87476
```

```r
sd(res$E)
```

```
## [1] 1.434007
```

4.2 Particle Filter

```r
particle_filter = function(observations, M=100, sd_emission=5, corr=TRUE) {

  T = length(observations)

  particle_plot_locations = c(1, round(T/3), round(T/3*2), T)

  particle_df = data.frame(matrix(nrow = M, ncol=length(particle_plot_locations)))

  X = matrix(nrow = T, ncol = M) # Posterior believe
  X_bar = matrix(nrow = T, ncol = M) # Prior believe
  W = matrix(nrow = T, ncol = M)
  Z = vector(length = T)
  S = vector(length = T)

  for (t in 1:T) {

    if (t == 1) {
      # Initialization
      X_temp = rinit(M)
      # Prediction
      X_bar[t,] = sapply(X_temp, rtransition)
    }
    else {
      # Prediction
      X_bar[t,] = sapply(X[t-1,], rtransition)
    }

    # Importance Weight
    W[t, ] = sapply(X_bar[t,], demission, z_t = observations[t], sd=sd_emission)
    # Normalize
    W[t, ] = W[t, ]/sum(W[t, ])
    # Correction
    if (corr) {
      prob = W[t,]
    }
    else {
      prob = rep(1, length(W[t,]))
    }
    X[t,] = sample(X_bar[t,], M, prob = prob, replace = TRUE)
    # Taken from Bishop
    #Z[t] = as.numeric(W[t, ] %*% X[t, ])
    Z[t] = mean(X[t,])
    S[t] = sd(X[t,])

    if (t %in% particle_plot_locations) {
      particle_df[, match(t, particle_plot_locations)] = X[t,]
    }
  }

  return(list(X=X, X_bar=X_bar, W=W, Z=Z, S=S, particles=particle_df))
}
```

```r
res_particle = particle_filter(X)

error_particle = abs(Z - res_particle$Z)
error_particle = error_particle[2:length(error_particle)]

mean(error_particle)
```

```
## [1] 1.771569
```

```r
sd(error_particle)
```

```
## [1] 1.337496
```

5 Source Code

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

head(asia)


## Training the network
train_bayesian_network = function(structure = NULL,
                                  data = train,
                                  learning_algorithm = iamb,
                                   ...) {

  # Network
  if (is.null(structure)) {
    bayesian_network = learning_algorithm(data, ...)
  }
  else {
    bayesian_network = structure
  }

  # Parameters
  bayesian_network_fit = bn.fit(bayesian_network, data)
  bayesian_network_grain = compile(as.grain(bayesian_network_fit))

  return(list(bn_grain=bayesian_network_grain,
              bn_fit=bayesian_network_fit,
              bn_structure=bayesian_network))
}

bn = train_bayesian_network(structure = NULL, data = asia, learning_algorithm = hc)


res_exact = querygrain(setEvidence(bn$bn_grain, c("X", "B"), c("yes", "yes")))$A
res_exact

res_approx_yes = cpquery(bn$bn_fit, event = (A == "yes"),
                     evidence = ((X == "yes") & (B == "yes")))
res_approx_yes

res_approx_no = cpquery(bn$bn_fit, event = (A == "no"),
                     evidence = ((X == "no") & (B == "yes")))
res_approx_no


isEquivilant = function(graph) {
  res = all.equal(cpdag(graph), skeleton(graph))
  if (res == "TRUE") return(TRUE)
  return(FALSE)
}

random_networks = random.graph(c("a", "b", "c", "d", "e"), num=50000)

res = sapply(random_networks, isEquivilant)

sum(res)/length(res)


N = 10

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

# Defining Symbols S1, S2, ..., SN
```

```r
symbols = paste(rep("S", N), 1:(N+1), sep = "")

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

# Transition Probabilities
transProbs = matrix(0, ncol = N, nrow = N)
# Staying in the current state with 0.5 probability is just die diagonal
diag(transProbs) = 0.5
# Moving to the next is also 0.5
diag(transProbs[,-1]) = 0.5
transProbs[10, 1] = 0.5

# Emission Probabilities
emissionProbs = matrix(0, ncol = N, nrow = N+1)

# 0.2 For i-2 to i+2
for (i in 1:N) {
  for (j in c(3:-1)) {
    emissionProbs[((i-j)%%N)+1,i] = 0.1
  }
}

emissionProbs[11,] = 0.5

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


observations = c("S1", "S11", "S11", "S11")

# The library returns the probabilities logged, we we have to de-log
alpha = exp(forward(robot_hmm, observations))
beta = exp(backward(robot_hmm, observations))

# Smoothed
# Can either be done manually or using the function posterior (== smoothed) in
# this package
# AUTOMATIC
smoothed_automatically = posterior(robot_hmm, observations)
# MANUALLY (Instead of division prop.table would work as well)
smoothed_manually = alpha * beta / colSums(alpha * beta)

# Path
hmm_viterbi = viterbi(robot_hmm, observations)

# Print Smoothed
smoothed_automatically
smoothed_manually
```

```r
# Print Viterbi
hmm_viterbi


ggplot(data = melt(smoothed_automatically),
       aes(y=states, x=index, fill=value)) +
       geom_raster()


states = c("SS", "SR", "RS", "RR")
symbols = c("R", "S")
startProbs = c(0.25, 0.25, 0.25, 0.25)

transProbs = matrix(c(0.75, 0.25, 0, 0,
                      0, 0, 0.5, 0.5,
                      0.5, 0.5, 0, 0,
                      0, 0, 0.25, 0.75), nrow=4, ncol=4, byrow=TRUE)
colnames(transProbs) = states
rownames(transProbs) = states

emissionProbs = t(matrix(c(0.1, 0.9, 0.9, 0.1, 0.1, 0.9, 0.9, 0.1),
                         nrow=2, ncol=4, byrow=FALSE))
rownames(emissionProbs) = states
colnames(emissionProbs) = symbols

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


simulatedStates = simHMM(weather_hmm, 10)
simulatedStates


# Matern32  kernel
k <- function(sigmaf = 1, ell = 1)
{
    rval <- function(x, y = NULL)
    {   r = sqrt(crossprod(x-y))
         return(sigmaf^2*(1+sqrt(3)*r/ell)*exp(-sqrt(3)*r/ell))
    }
    class(rval) <- "kernel"
    return(rval)
}

sigma_f_sq = 1
ell = 0.5

zGrid = seq(0.01, 1, 0.01)
f_of_z = vector(length = length(zGrid))
```

```r
kernel = k(sigmaf = sigma_f_sq, ell = ell)

for (i in 1:length(zGrid)) {
  f_of_z[i] = kernel(0, zGrid[i])
}

plot(zGrid, f_of_z)


sigma_f_sq = 0.5

kernel = k(sigmaf = sigma_f_sq, ell = ell)

for (i in 1:length(zGrid)) {
  f_of_z[i] = kernel(0, zGrid[i])
}

plot(zGrid, f_of_z)


#' gp_covariance
#'
#' @param x X, which will be scaled.
#' @param xss X_star, which will be scaled.
#'
#' @return The covariance matrix.
gp_covariance = function(x, xss, kernel, sigma_n) {
  # Copied from given scripts
  x = scale(x)
  xs = scale(xss)
  n = nrow(x)
  Kss = kernelMatrix(kernel = kernel, x = xs, y = xs)
  Kxx = kernelMatrix(kernel = kernel, x = x, y = x)
  Kxs = kernelMatrix(kernel = kernel, x = x, y = xs)
  Covf = Kss - t(Kxs) %*% solve(Kxx + sigma_n^2*diag(n), Kxs)
  return(Covf)
}


library(kernlab)
load("lidar.RData") # loading the data
sigmaNoise = 0.05
x = distance
y = logratio

# Set up the kernel function
kernelFunc <- k(sigmaf = 1, ell = 1)

GPfit = gausspr(x, y, kernel = kernelFunc, var = sigmaNoise^2)
meanPred = predict(GPfit, x)

varProb = diag(gp_covariance(x, x, kernel = kernelFunc, sigma_n = sigmaNoise))
varPred = varProb + sigmaNoise^2
```

```r
df1 = data.frame(x,
                y,
                meanPred,
                lowerProb = meanPred - 1.96 * sqrt(varProb),
                higherProb = meanPred + 1.96 * sqrt(varProb),
                lowerPred = meanPred - 1.96 * sqrt(varPred),
                higherPred = meanPred + 1.96 * sqrt(varPred))

ggplot(df1) +
  geom_ribbon(aes(x = x, ymin = lowerPred, ymax = higherPred),
              alpha=0.05, linetype=3, colour="grey70", size=1, fill="black") +
  geom_ribbon(aes(x = x, ymin = lowerProb, ymax = higherProb),
              alpha=0.1, linetype=3, colour="grey70", size=1, fill="blue") +
  geom_point(aes(x = x, y = y), color = "black", fill = "#dedede", shape = 21) +
  geom_line(aes(x = x, y = meanPred, colour = "Mean")) +
  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()



# Set up the kernel function
kernelFunc <- k(sigmaf = 1, ell = 5)

GPfit = gausspr(x, y, kernel = kernelFunc, var = sigmaNoise^2)
meanPred = predict(GPfit, x)

varProb = diag(gp_covariance(x, x, kernel = kernelFunc, sigma_n = sigmaNoise))
varPred = varProb + sigmaNoise^2

df2 = data.frame(x,
                y,
                meanPred,
                lowerProb = meanPred - 1.96 * sqrt(varProb),
                higherProb = meanPred + 1.96 * sqrt(varProb),
                lowerPred = meanPred - 1.96 * sqrt(varPred),
                higherPred = meanPred + 1.96 * sqrt(varPred))

ggplot(df2) +
  geom_ribbon(aes(x = x, ymin = lowerPred, ymax = higherPred),
              alpha=0.05, linetype=3, colour="grey70", size=1, fill="black") +
  geom_ribbon(aes(x = x, ymin = lowerProb, ymax = higherProb),
              alpha=0.1, linetype=3, colour="grey70", size=1, fill="blue") +
  geom_point(aes(x = x, y = y), color = "black", fill = "#dedede", shape = 21) +
  geom_line(aes(x = x, y = meanPred, colour = "Mean")) +
  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()


rtransition = function(x_t_1, sd=1) {
  return(rnorm(n = 1, x_t_1 + 1, sd = sd))
}

remission = function(x_t, sd=5) {
  return(rnorm(n = 1, mean = x_t, sd = sd))
}

demission =function(z_t, x_t, sd=5) {
  return(dnorm(z_t, mean = x_t, sd = sd))
}

rinit = function(n) {
  return(rnorm(n = n, mean = 50, sd = 10))
}

sample_observations = function(n) {
  # First observation
  states = vector(length = n) # X
  observations = vector(length = n) # Z
  states[1] = rinit(1)
  observations[1] = remission(1, states[1])

  for(i in 2:n) {
    states[i] = rtransition(states[i-1])
    observations[i] = remission(states[i])
  }

  return(data.frame(states = states, obs = observations))
}


kalman_filter = function(mu0, Sigma0, X, Z, R, Q, A=1, C=1) {

  Sigma0 = Sigma0^2
  R = R^2
  Q = R^2

  # Variables
  mu = vector(length = length(X))
  Sigma = vector(length = length(X))
  K = vector(length = length(X))
  E = vector(length = length(X))

  # Init
  mu[1] = mu0
  Sigma[1] = Sigma0

  # Has to start at 2 as R is stupid and doesn't like 0 indeces
  for (t in 2:length(X)) {
    mu_bar = mu[t-1] + 1
    Sigma_bar = A * Sigma[t-1] * A + R # A = 1
    K[t] = Sigma_bar * C / (C * Sigma_bar * C + Q) # C = 1
    mu[t] = mu_bar + K[t] * (X[t] - C * mu_bar)
    Sigma[t] = (1 - K[t] * C) * Sigma_bar
    E[t] = abs(X[t] - mu[t])
  }

  mu = mu[2:length(mu)]
  Sigma = Sigma[2:length(Sigma)]
  K = K[2:length(K)]
  E = E[2:length(E)]

  return(list(mu=mu, Sigma=Sigma, K=K, A=A, C=C, R=R, Q=Q, X=X, E=E))
}


sample = sample_observations(10000)
Z = sample$states
X = sample$obs
R = 1 # transition
Q = 5 # semision

mu0 = 50
Sigma0 = 10

res = kalman_filter(mu0, Sigma0, X, Z, R, Q)

mean(res$E)
sd(res$E)



particle_filter = function(observations, M=100, sd_emission=5, corr=TRUE) {

  T = length(observations)

  particle_plot_locations = c(1, round(T/3), round(T/3*2), T)

  particle_df = data.frame(matrix(nrow = M, ncol=length(particle_plot_locations)))

  X = matrix(nrow = T, ncol = M) # Posterior believe
  X_bar = matrix(nrow = T, ncol = M) # Prior believe
  W = matrix(nrow = T, ncol = M)
  Z = vector(length = T)
  S = vector(length = T)

  for (t in 1:T) {

    if (t == 1) {
      # Initialization
      X_temp = rinit(M)
      # Prediction
      X_bar[t,] = sapply(X_temp, rtransition)
    }
    else {
      # Prediction
      X_bar[t,] = sapply(X[t-1,], rtransition)
    }

    # Importance Weight
    W[t, ] = sapply(X_bar[t,], demission, z_t = observations[t], sd=sd_emission)
    # Normalize
    W[t, ] = W[t, ]/sum(W[t, ])
    # Correction
    if (corr) {
      prob = W[t,]
    }
    else {
      prob = rep(1, length(W[t,]))
    }
    X[t,] = sample(X_bar[t,], M, prob = prob, replace = TRUE)
    # Taken from Bishop
    #Z[t] = as.numeric(W[t, ] %*% X[t, ])
    Z[t] = mean(X[t,])
    S[t] = sd(X[t,])

    if (t %in% particle_plot_locations) {
      particle_df[, match(t, particle_plot_locations)] = X[t,]
    }
  }

  return(list(X=X, X_bar=X_bar, W=W, Z=Z, S=S, particles=particle_df))
}


res_particle = particle_filter(X)

error_particle = abs(Z - res_particle$Z)
error_particle = error_particle[2:length(error_particle)]

mean(error_particle)
sd(error_particle)
```