AllLabs

Elvin Granat

2018-10-18

```
knitr::opts_chunk$set(collapse = TRUE, comment = "#>")
knitr::opts_chunk$set(fig.pos='H', fig.align='center')
```

Lab 1 - Bayesian Networks

Bayesian Networks using the "asia" dataset

```
# Libraries
# install.packages("bnlearn")
# source("https://bioconductor.org/biocLite.R")
# biocLite("RBGL")
# install.packages("gRain")
# source("https://bioconductor.org/biocLite.R")
# biocLite("Rgraphviz")
require(knitr)
#> Loading required package: knitr
library("gRain")
#> Loading required package: gRbase
#> Warning: package 'gRbase' was built under R version 3.4.4
library("bnlearn")
#> Warning: package 'bnlearn' was built under R version 3.4.4
#>
#> Attaching package: 'bnlearn'
#> The following objects are masked from 'package:gRbase':
#>
#>     ancestors, children, parents
#> The following object is masked from 'package:stats':
#>
#>     sigma

# Setup data
data("asia")
data.asia = asia
```

Lab 1 Task 1

Learn structure with different configurations (network scores and number of restarts)

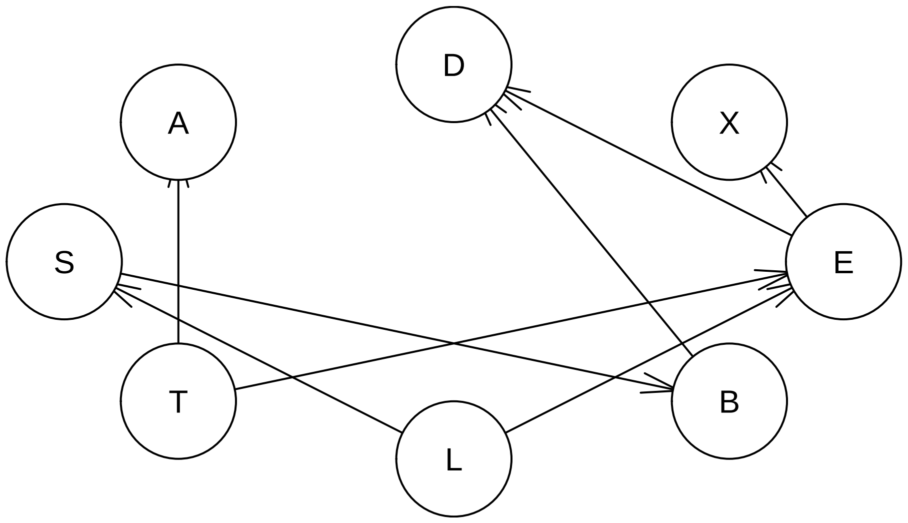
```r
set.seed(123)
network.scores = c("loglik", "bde", "bic")
equal.count = 0
n = 100


equal_graphs = function(BN1, BN2) {
  CP1 = cpdag(BN1)
  CP2 = cpdag(BN2)

  return (ifelse(all.equal(CP1, CP2) == TRUE, 1, 0))
}

for (i in 1:n) {

  # construct two BNs with the same configuration but different seeds.
  BN1 = hc(data.asia, restart = 10, score = network.scores[sample(1:3,1)])
  BN2 = hc(data.asia, restart = 10, score = network.scores[sample(1:3,1)])

  # keep track of how many are equal
  equal.count = equal.count + equal_graphs(BN1, BN2)
}

# could also try same scores but different seed values, number of restarts etc.

paste(equal.count, "equal out of", n)
#> [1] "39 equal out of 100"
```

The hill climbing algorithm is completely deterministic so unless we do random restarts the result is the same. The random restarts means that we begin the algorithm with different initial graph structures which may possibly end up in different local optima.

Lab 1 Task 2

Learn appropriate BN from 80 % of the data.

```r
# Setup test/train
n = dim(data.asia)[1]
train = data.asia[1:4000,]
test = subset(data.asia[4001:n, ], select = -c(S))
y.test = data.asia[4001:n, ]$S

# Learn a BN from the train data
set.seed(123)
BN.structure = hc(train, restart = 10, score = "bde")
plot(BN.structure)
```



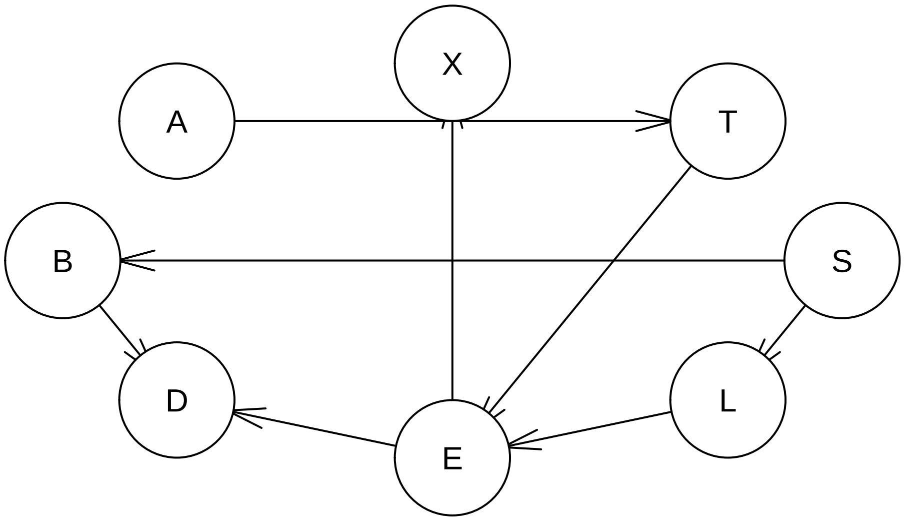
Exact inference

```r
# Function to get confusion matrix
get_confusion_matrix = function(y, y.pred) {
  return(table(y, y.pred, dnn=c("TRUE", "PRED")))
}

# Function to get missclassification rate
get_missclassifcation_rate = function(y, y.pred) {
  n = length(y)
  k = sum(abs(ifelse(y == "yes", 1, 0) - ifelse(y.pred == "yes", 1, 0)))
  return(k/n)
}

# Function to perform exact inference and get predictions
get_predictions_exact_inference = function(BN.structure, test, y.test, nodes) {
  # Fit
  BN.fit = bn.fit(BN.structure, train)
  BN.grain = compile(as.grain(BN.fit))

  # Setup predictions
  n.test = length(y.test)
  predictions = numeric()

  # Make predictions
  for (i in 1:n.test) {

    # Extract state
    z = NULL
    for (j in nodes) {
      if (test[i,j] == "yes") z = c(z, "yes")
      else z = c(z, "no")
    }

    # Set evidence
    evidence = setEvidence(BN.grain, nodes = nodes, states = z)

    # Perform query
    query = querygrain(evidence, c("S"))

    # Make prediction
    if (query$S["yes"] >= 0.5) predictions[i] = "yes"
    else predictions[i] = "no"

    # Approximate: prop.table(table(cpdist(BN.fit, "S", (X == "yes" & B == "yes"))))
  }

  return(predictions)
}
nodes = colnames(test) # observations on all nodes
y.pred = get_predictions_exact_inference(BN.structure, test, y.test, nodes)

get_confusion_matrix(y.test, y.pred)
```

```
#>       PRED
#> TRUE   no yes
#>   no  358 147
#>   yes 120 375
paste("MCR:", get_missclassifcation_rate(y.test, y.pred))
#> [1] "MCR: 0.267"
```

Create the true network:

```
BN.true = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
plot(BN.true)
```



Compare to true network. We can see that the results are exactly the same, this is because the Markov blanket is the same for both networks.

```
y.pred = get_predictions_exact_inference(BN.true, test, y.test, nodes)

get_confusion_matrix(y.test, y.pred)
#>       PRED
#> TRUE   no yes
#>   no  358 147
#>   yes 120 375
paste("MCR (TRUE):", get_missclassifcation_rate(y.test, y.pred))
#> [1] "MCR (TRUE): 0.267"
```

Lab 1 Task 3

Same as Task 2 but using only the markov blanket as predictors. We can see that the results are the same. As stated in previous example, the networks yield the same result,given the markov blanket values the resulting predictions are the same. This is because the node S is independent all other nodes under the MB criteria.

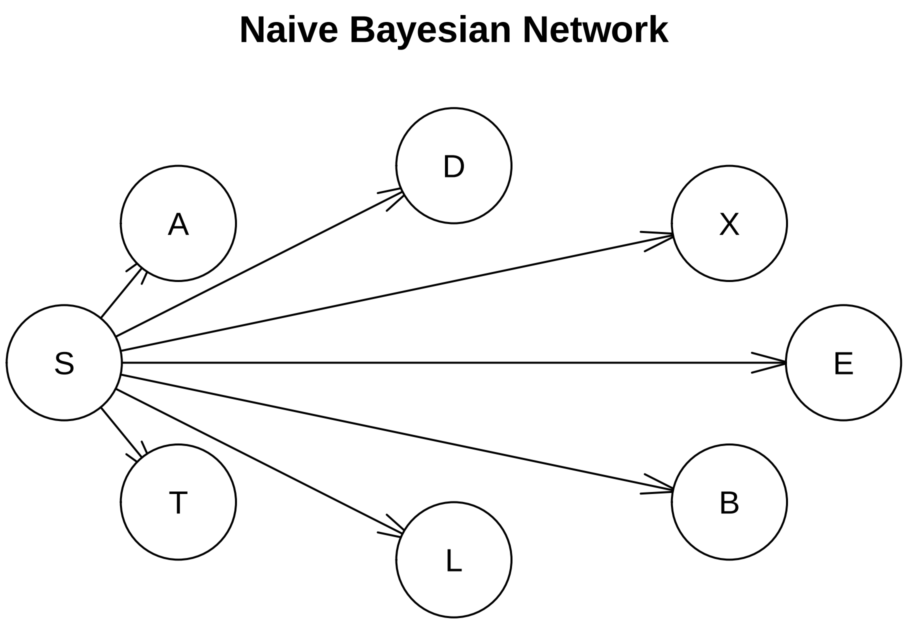
```
nodes = mb(BN.structure, c("S")) # Observations of the markov blanket
y.pred = get_predictions_exact_inference(BN.structure, test, y.test, nodes)

get_confusion_matrix(y.test, y.pred)
#>      PRED
#> TRUE   no yes
#>   no  358 147
#>   yes 120 375
paste("MCR:", get_missclassifcation_rate(y.test, y.pred))
#> [1] "MCR: 0.267"
```

```
nodes = mb(BN.true, c("S")) # Observations of the markov blanket
y.pred = get_predictions_exact_inference(BN.true, test, y.test, nodes)

get_confusion_matrix(y.test, y.pred)
#>      PRED
#> TRUE   no yes
#>   no  358 147
#>   yes 120 375
paste("MCR (TRUE):", get_missclassifcation_rate(y.test, y.pred))
#> [1] "MCR (TRUE): 0.267"
```

Lab 1 Task 4

Naive Bayesian Network, where the predictive variables are independent each other given the class variable.

```
BN.naive = empty.graph(c("A", "S", "T", "L", "B", "E", "X", "D"))
arc.set = matrix(c(
  "S", "A",
  "S", "T",
  "S", "L",
  "S", "B",
  "S", "E",
  "S", "X",
  "S", "D"
  ), ncol = 2, byrow = TRUE, dimnames = list(NULL, c("from", "to")))

arcs(BN.naive) = arc.set
plot(BN.naive, main="Naive Bayesian Network")
```



Slightly worse performance than previous examples, given that the information gained in this structure is in reality independent makes the model either overfit or just straight up incorrect

```
nodes = colnames(test)
y.pred = get_predictions_exact_inference(BN.naive, test, y.test, nodes)

get_confusion_matrix(y.test, y.pred)
#>      PRED
#> TRUE   no yes
#>   no  389 116
#>   yes 180 315
paste("MCR (TRUE):", get_missclassifcation_rate(y.test, y.pred))
#> [1] "MCR (TRUE): 0.296"
```

Lab 1 Task 5

A node is conditionally independent of all nodes not in its Markov blanket, given its Markov blanket, due to D-separation implying independence. Therefore, conditioning on all explanatory variables and conditioning on the Markov blanket yields the same result, which is why the results of 2) and 3) are equal.

The Naive Bayes method assumes conditional independence of all explanatory variables given the response variable. Looking at the figure we can see that this is not the case in neither the learned or true network structure. This naive network structure leads to different results in 4) compared to 2) and 3).

Lab 2 - Hidden Markov Models (HMM)

Hidden Markov Models using simulated robot movement of which we are to predict using filtering, smoothing and most probable path (using viterbi algorithm).

```r
# Libraries
library("HMM")
library("entropy")
#>
#> Attaching package: 'entropy'
#> The following object is masked from 'package:bnlearn':
#>
#>     discretize
```

Lab 2 Task 1

Build a HMM describing the given scenario

```r
states = paste("z", 1:10, sep="") # hidden states
symbols = paste("x", 1:10, sep="") # observed emissions
startProbs = rep(0.1, 10)
transProbs <- diag(1/2, 10) +
  diag(1/2, 10)[, c(10, 1:9)]
emissionProbs <-
  diag(1/5, 10)[, c(3:10, 1:2)] +
  diag(1/5, 10)[, c(2:10, 1)] +
  diag(1/5, 10) +
  diag(1/5, 10)[, c(10, 1:9)] +
  diag(1/5, 10)[, c(9:10, 1:8)]
transProbs = t(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
    ),
  nrow = 10, ncol = 10
))
emissionProbs = t(matrix(
  c(
    0.2, 0.2, 0.2, 0, 0, 0, 0, 0, 0.2, 0.2,
    0.2, 0.2, 0.2, 0.2, 0, 0, 0, 0, 0, 0.2,
    0.2, 0.2, 0.2, 0.2, 0.2, 0, 0, 0, 0, 0,
    0, 0.2, 0.2, 0.2, 0.2, 0.2, 0, 0, 0, 0,
    0, 0, 0.2, 0.2, 0.2, 0.2, 0.2, 0, 0, 0,
    0, 0, 0, 0.2, 0.2, 0.2, 0.2, 0.2, 0, 0,
    0, 0, 0, 0, 0.2, 0.2, 0.2, 0.2, 0.2, 0,
    0, 0, 0, 0, 0, 0.2, 0.2, 0.2, 0.2, 0.2,
    0.2, 0, 0, 0, 0, 0, 0.2, 0.2, 0.2, 0.2,
    0.2, 0.2, 0, 0, 0, 0, 0, 0.2, 0.2, 0.2
  ),
  nrow = 10, ncol = 10
))
hmm = initHMM(states, symbols, startProbs, transProbs, emissionProbs)
```

Lab 2 Task 2

Simulate the HMM for T = 100 time steps

```
T = 100
set.seed(123)
sim = simHMM(hmm, T)
```

Lab 2 Task 3

Compute filtered- & smoother probability distributions, as well as the most probable path (using the viterbi algorithm).

```
# Calculate alpha and beta from observations
obs = sim$observation
alpha = exp(forward(hmm, obs))
beta = exp(backward(hmm, obs))

# Setup calculation of filtering & smoothing
filtering = matrix(NA, 10, T)
smoothing = matrix(NA, 10, T)

# Calculate filtering & smoothing
alphaSum = apply(alpha, 2, sum)
alphaBetaSum = apply(alpha * beta, 2, sum)
for(t in 1:T) {
  filtering[,t] = alpha[,t] / alphaSum[t]
  smoothing[,t] = (alpha[,t] * beta[,t]) / alphaBetaSum[t]
}

# Most probable path
y.path = viterbi(hmm, obs)
```

Lab 2 Task 4

Compute accuracy of filtering, smoothing and most probable path

```
y.filtering = c()
y.smoothing = c()
for(t in 1:T) {
  y.filtering[t] = states[which.max(filtering[,t])] # break ties using which.is.max in package "nnet"
  y.smoothing[t] = states[which.max(smoothing[,t])]
}
```

```r
y = sim$states
pred.filtering = y.filtering == y
pred.smoothing = y.smoothing == y
pred.path = y.path == y

accuracy = function(pred) {
  return(mean(pred))
}

accuracy.filtering = accuracy(pred.filtering)
accuracy.smoothing = accuracy(pred.smoothing)
accuracy.path = accuracy(pred.path)

paste("filtering:", accuracy.filtering)
#> [1] "filtering: 0.53"
paste("smoothing:", accuracy.smoothing)
#> [1] "smoothing: 0.64"
paste("most probable path", accuracy.path)
#> [1] "most probable path 0.36"
```

Task 5

Repeat previous excersise with different simulated samples, see which performs best and explain why.

```r
set.seed(123)
best = c(0, 0, 0) # counter of best accuracy of the algorithms
for (i in 1:10) {
  T = 100
  sim = simHMM(hmm, T)

  # Calculate alpha & beta from observations
  obs = sim$observation
  alpha = exp(forward(hmm, obs))
  beta = exp(backward(hmm, obs))

  # Setup calculation of filtering & smoothing
  filtering = matrix(0, 10, T)
  smoothing = matrix(0, 10, T)

  # Calculate filtering & smoothing
  alphaSum = apply(alpha, 2, sum)
  alphaBetaSum = apply(alpha * beta, 2, sum)
  for (t in 1:T) {
    filtering[,t] = alpha[,t] / alphaSum[t]
    smoothing[,t] = (alpha[,t] * beta[,t]) / alphaBetaSum[t]
  }

  # Predict most probable path
  y.path = viterbi(hmm, obs)

  # Extract predicted states of filtering & smoothing
  y.filtering = c()
  y.smoothing = c()
  for (t in 1:T) {
    y.filtering[t] = states[which.max(filtering[,t])]
    y.smoothing[t] = states[which.max(smoothing[,t])]
  }

  # Determine correct prediction states of filtering, smoothing and viterbi in comparison to the true p
  y = sim$states
  pred.filtering = y.filtering == y
  pred.smoothing = y.smoothing == y
  pred.path = y.path == y

  # Provide missclassification rates of the different methods of prediction
  accuracy.filtering = accuracy(pred.filtering)
  accuracy.smoothing = accuracy(pred.smoothing)
  accuracy.path = accuracy(pred.path)

  ### Task 5
  which.best = which.max(c(accuracy.filtering, accuracy.smoothing, accuracy.path))
  best[which.best] = best[which.best] + 1
}
kable(t(best), col.names=c("filtering", "smoothing", "viterbi"))
```

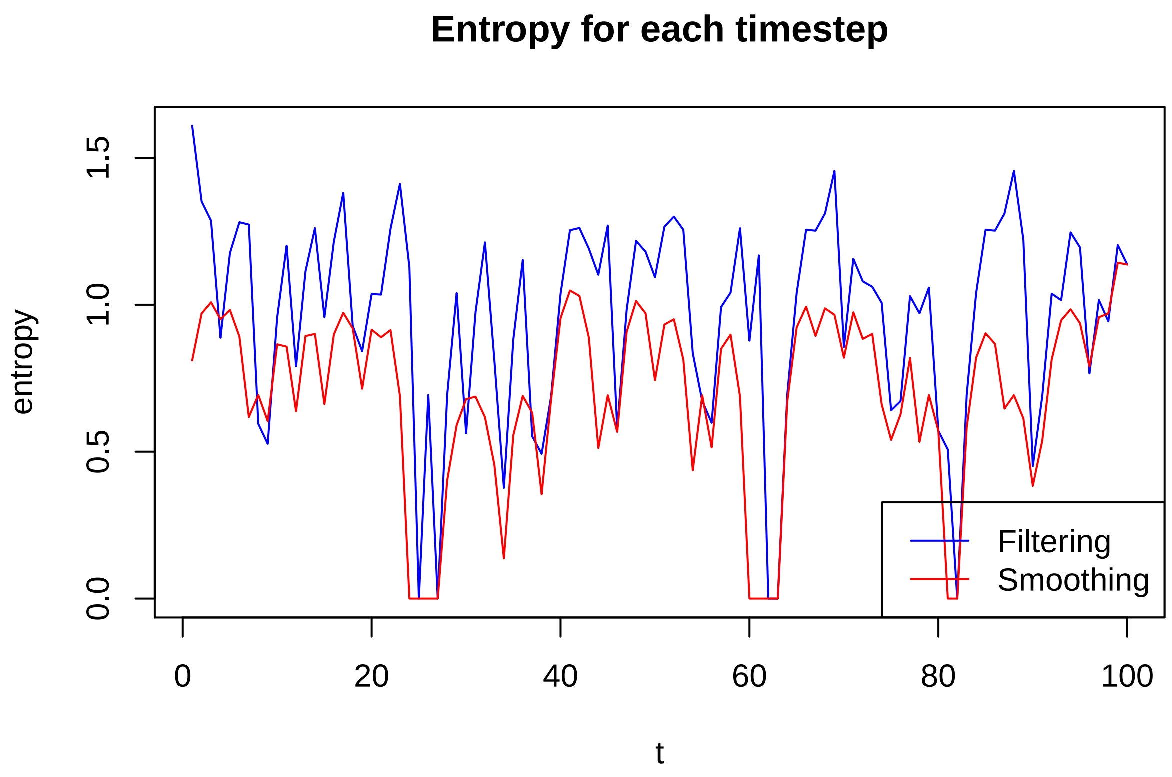
filtering	smoothing	viterbi
0	10	0

Results Filtering: 0 Smoothing: 10 Viterbi: 0 Smoothing outperforms the other two each simulation. Smoothing outperforms filtering since it takes the future t+1 into account (via beta) Smoothing outperforms most probable path since it focuses on making the best prediction on current t compared to the entire path of most probable path

Lab 2 Task 6

More observations -> More accurate?

```
entropy.filtering = c()
entropy.smoothing = c()
for(t in 1:T) {
  entropy.filtering[t] = entropy.empirical(filtering[,t])
  entropy.smoothing[t] = entropy.empirical(smoothing[,t])
}
# Comment:
# Entropi = hur rörig din information är, hur mycket osäkerhet i din fördelning, likformig = max entrop
# Går nedåt -> Vet bättre, om det går uppåt -> Vet mindre
plot(1:T, entropy.filtering, type="l", col="blue", main="Entropy for each timestep", xlab="t", ylab="en
lines(1:T, entropy.smoothing, col="red")
legend("bottomright", legend=c("Filtering", "Smoothing"), lty=c(1,1), col=c("blue", "red"))
```



We can see that we do not know more about the location of the robot given more observations. The entropy remains random even while increasing the number of observations added to the hmm. This is because it is markovian and only depends on the previous observation.

Lab 2 Task 7

Probabilty of the hidden state at timestep t = 101. Making the prediction of t = 101 using the known transformation matrix and the prediction on t = 100.

```
### Task 7
p.100 = filtering[, 100] # same as smoothing, since its based on the last observation (beta = 1)'
p.101 = t(transProbs) %*% p.100 # transformation matrix
kable(cbind(p.100, as.vector(p.101)), col.names=c("t = 100", "t = 101"), digits= 3, caption="Predictions
```

Table 2: Predictions

t = 100	t = 101
0.171	0.100
0.399	0.285
0.401	0.400
0.000	0.201
0.000	0.000
0.000	0.000
0.000	0.000
0.000	0.000
0.000	0.000
0.029	0.014

Lab 3 - State Space Models

Lab 3 Task 1

Simulation function setup

```
# Drawing transition sample given its previous state
z_transition_sample = function(z_prev) {
  pick = sample(1:3, 1)
  switch(pick,
     "first" = {
       z = rnorm(1, z_prev, 1)
     },
     "second" = {
       z = rnorm(1, z_prev + 1, 1)
     },
     "third" = {
       z = rnorm(1, z_prev + 2, 1)
     }
   )
  return(z)
}

# Drawing emission from a mixed normal given the true state z
x_emission_sample = function(z, sigmaE) {
  pick = sample(1:3, 1)
  switch(pick,
     "first" = {
       x = rnorm(1, z, sigmaE)
```

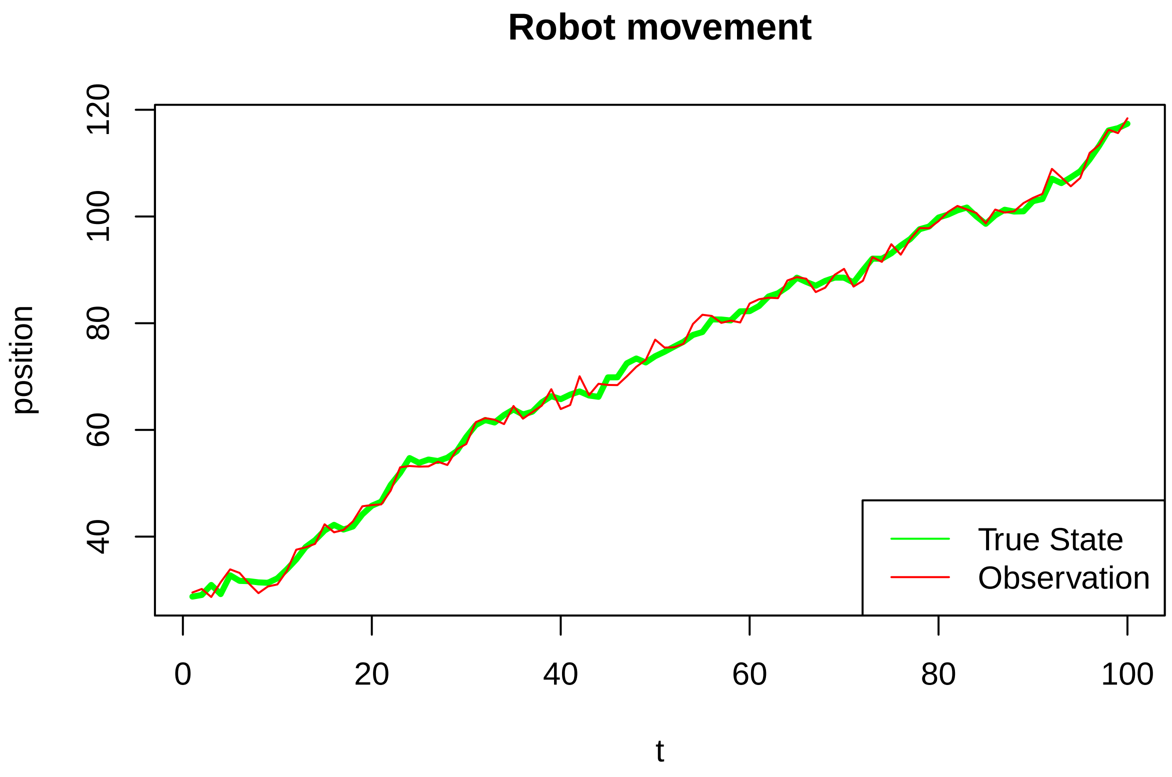
```r
    },
    "second" = {
      x = rnorm(1, z - 1, sigmaE)
    },
    "third" = {
      x = rnorm(1, z + 1, sigmaE)
    }
   )
  return(x)
}

# Simulating T = 100 steps of simulation, where the initial state is uniform
simulation = function(sigmaE) {
  set.seed(123)
  T = 100
  z = c() # true states
  x = c() # observed states
  for (t in 1:T) {
    # z
    if (t == 1) z[t] = runif(1, 0, 100)
    else z[t] = z_transition_sample(z[t-1])

    # x
    x[t] = x_emission_sample(z[t], sigmaE)
  }
  return (data.frame(x, z))
}
```

Particle filtering

```r
w_density = function(x, z, sigmaE) {
  w = c()
  M = length(x)
  for(m in 1:M) {
    w1 = dnorm(x, mean = z, sd = sigmaE, log = FALSE)
    w2 = dnorm(x, mean = z - 1, sd = sigmaE, log = FALSE)
    w3 = dnorm(x, mean = z + 1, sd = sigmaE, log = FALSE)
    w[m] = (w1 + w2 + w3) / 3
  }
  return(w)
}

particle_draw = function(z_particles, w) {
  M = length(w)
  res = c()
  draws = rmultinom(1, M, w)
  for (i in 1:M) {
    if (draws[i]) {
      for (j in 1:draws[i]) {
        res = append(res, z_particles[i])
      }
    }
  }
  return(res)
}

particle_filtering = function(sim, sigmaE, equalWeight = FALSE) {
  set.seed(123)
  x = sim$x # extract observations

  T = 100
  M = 100
  Z = matrix(0, nrow = M, ncol = T)
  Zbar = matrix(0, nrow = M, ncol = T)

  for (t in 1:T) {
    z_particles = c()
    w = c()
    for (m in 1:M) {
      if (t == 1) z_particles[m] = runif(1, 0, 100)
      else z_particles[m] = z_transition_sample(Z[m,t-1])

      if (equalWeight) w[m] = 1
      else w[m] = w_density(x[t], z_particles[m], sigmaE)
    }
    Zbar[,t] = z_particles
    Z[,t] = particle_draw(Zbar[,t], w)
  }
  return(Z)
}
```

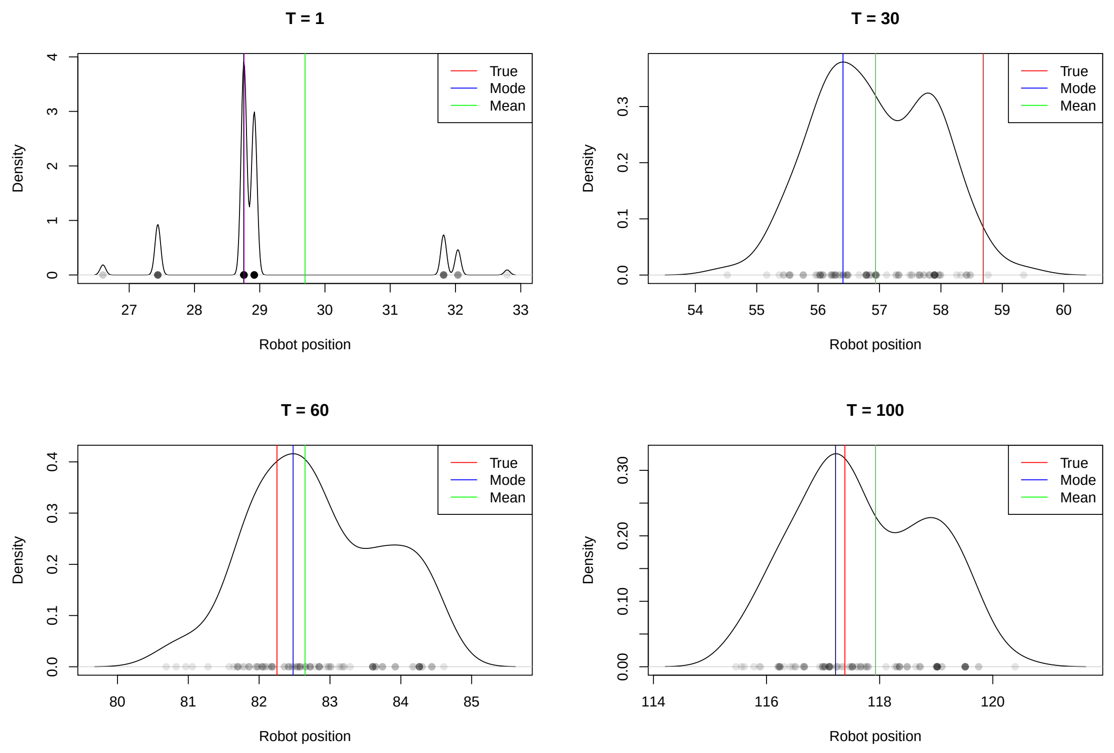
Execution of simulation

```
sigma_emission1 = 1
sim1 = simulation(sigma_emission1)
plot_movement = function(sim) {
  plot(1:100, sim$z, type="l", col="green", main="Robot movement", xlab="t", ylab="position", lwd=3)
  lines(1:100, sim$x, col="red")
  legend("bottomright", legend=c("True State", "Observation"), col=c("green","red"), lty=c(1,1))
}
plot_movement(sim1)
```



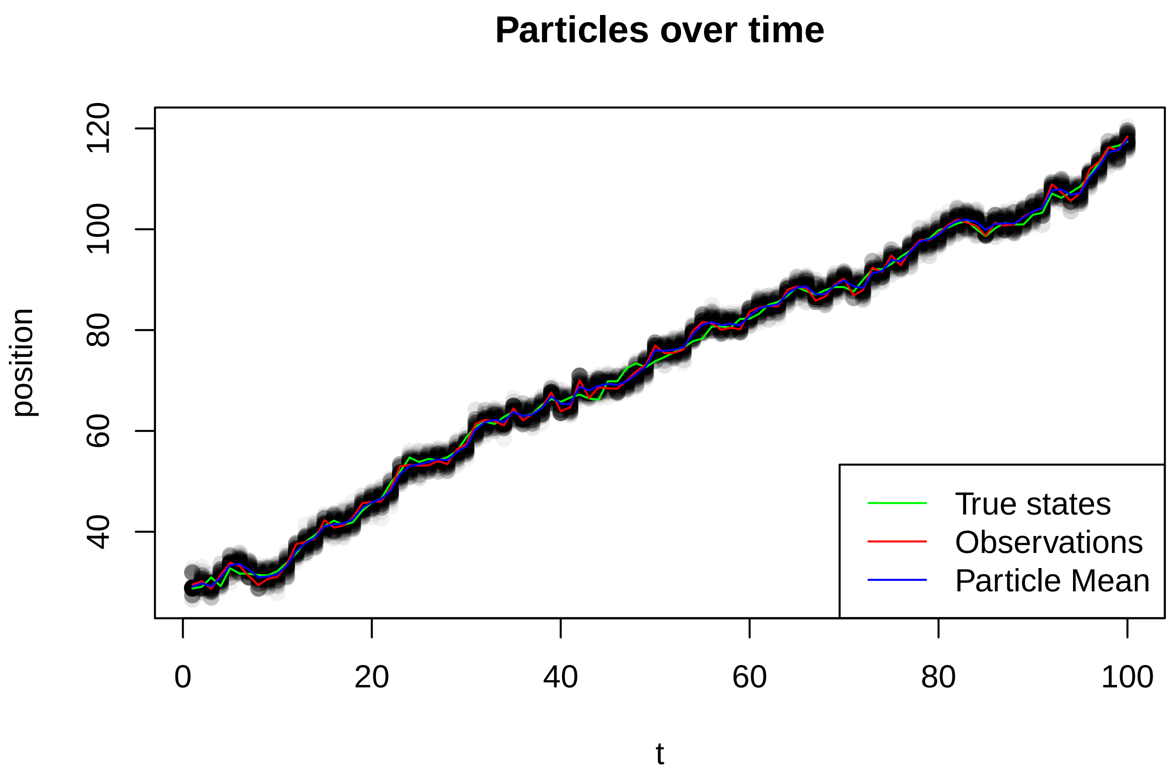
Execution of particle filtering

```r
# ```{r, out.width="45%", fig.show="hold"}
set.seed(123)
Z_particles1 = particle_filtering(sim1, sigma_emission1)

visualize = function(z, x, Z, t) {
  d = density(Z[,t])
  plot(d, main=paste("T =", t), xlab = "Robot position")
  points(Z[,t], rep(0, 100), pch=19, col=rgb(0,0,0,0.1))
  abline(v=z[t], col="red")
  abline(v=d$x[which.max(d$y)], col="blue")
  abline(v=mean(d$x), col="green")
  legend("topright",
         legend = c("True", "Mode", "Mean"),
         col=c("red", "blue", "green"),
         lty=1)
}
visualize(sim1$z, sim1$x, Z_particles1, 1)
visualize(sim1$z, sim1$x, Z_particles1, 30)
visualize(sim1$z, sim1$x, Z_particles1, 60)
visualize(sim1$z, sim1$x, Z_particles1, 100)
```



Plot particles over time

```
plot_particles = function(sim, Z_particles) {
  plot(NA, xlim=c(1,100), ylim=c(min(Z_particles), max(Z_particles)), main="Particles over time", xlab=
  for(i in 1:100) {
    points(rep(i,100), Z_particles[,i], col=rgb(0,0,0,0.05), pch=19)
  }
  lines(1:100, sim$z, col="green")
  lines(1:100, sim$x, col="red")
  lines(1:100, apply(Z_particles,2,mean), col="blue")
  legend("bottomright", legend=c("True states", "Observations", "Particle Mean"),
         col=c("green", "red", "blue"), lty=c(1,1,1))
}
plot_particles(sim1, Z_particles1)
```

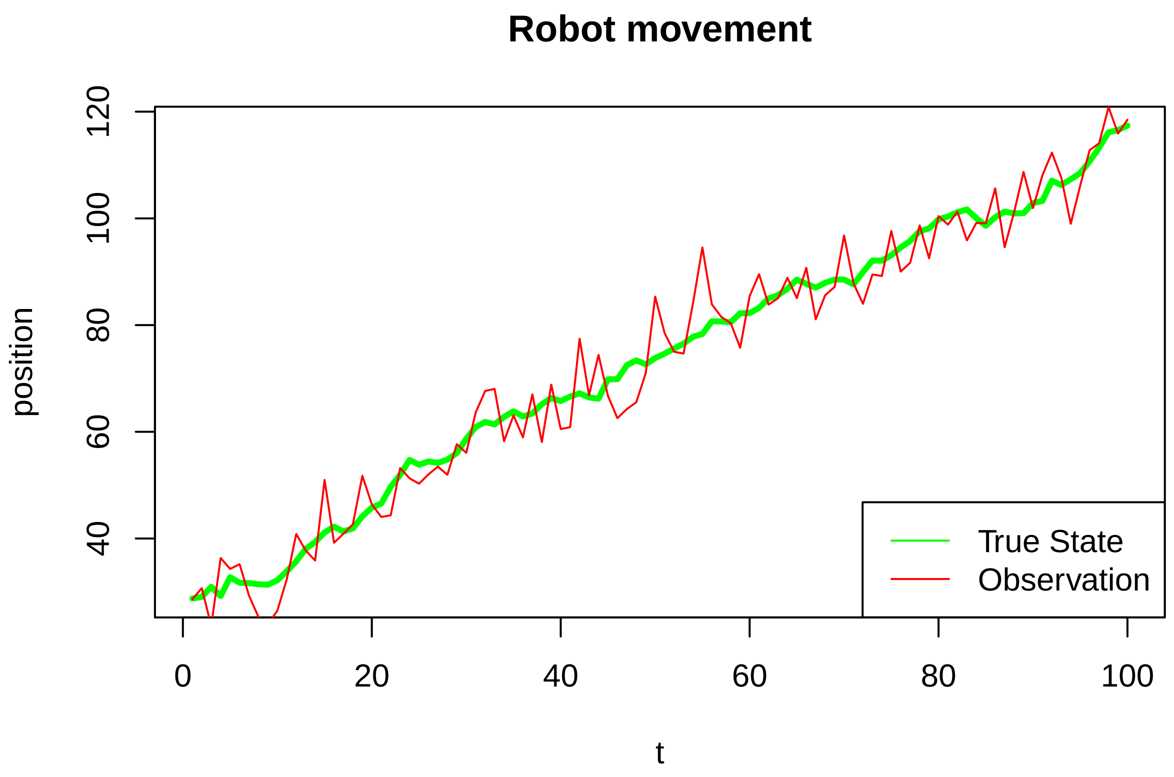


Lab 3 Task 2

Same as task 1 but with different emission standard deviations

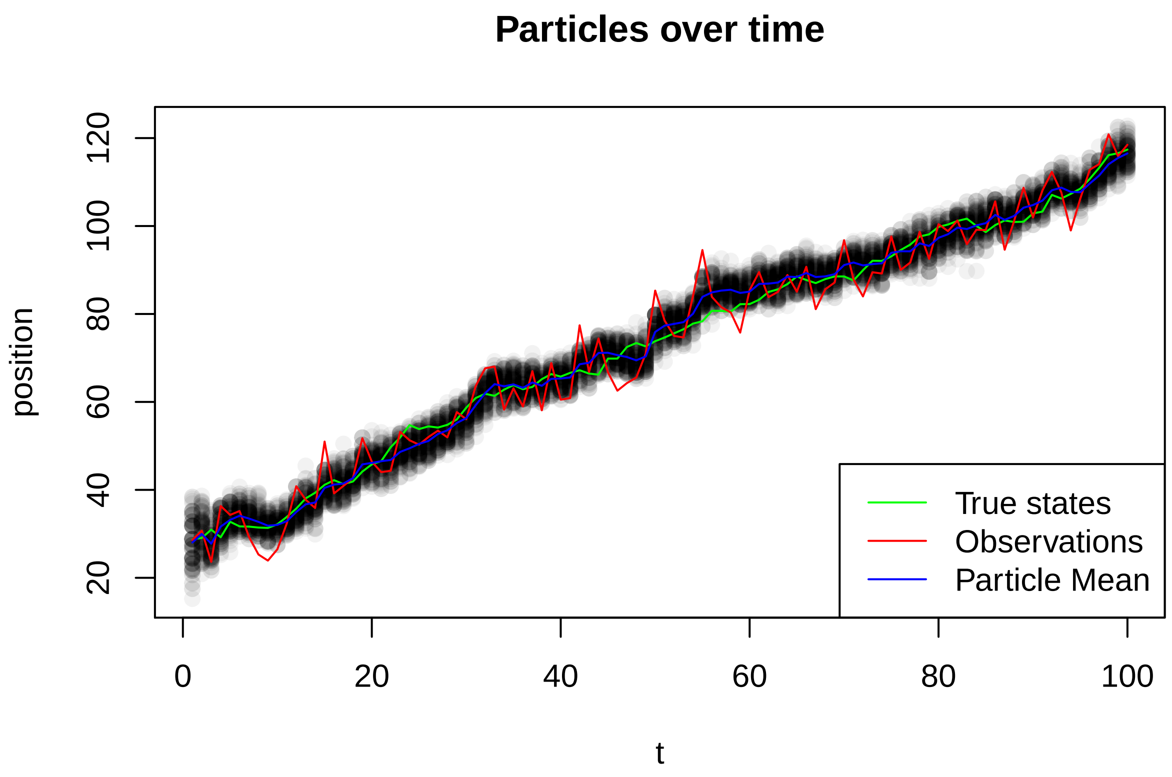
Sigma = 5

```r
sigma_emission5 = 5
sim5 = simulation(sigma_emission5)
plot_movement(sim5)
```



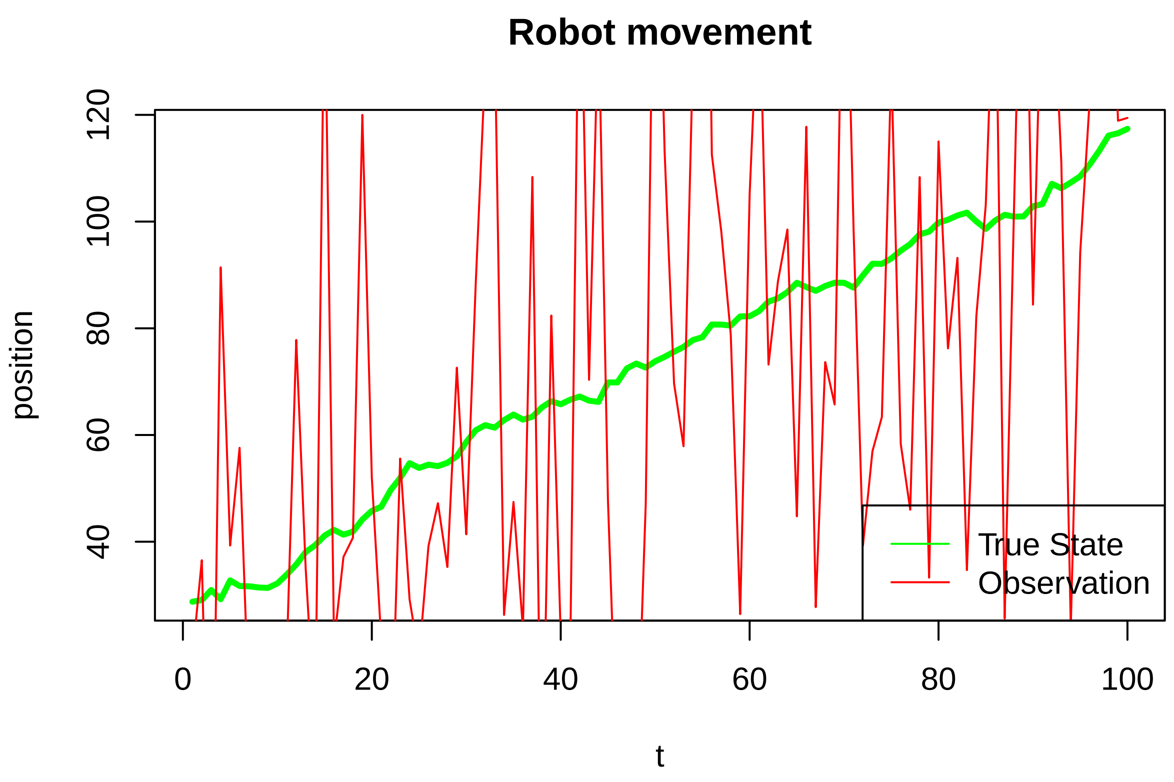
Particle filtering

```r
Z_particles5 = particle_filtering(sim5, sigma_emission5)
plot_particles(sim5, Z_particles5)
```

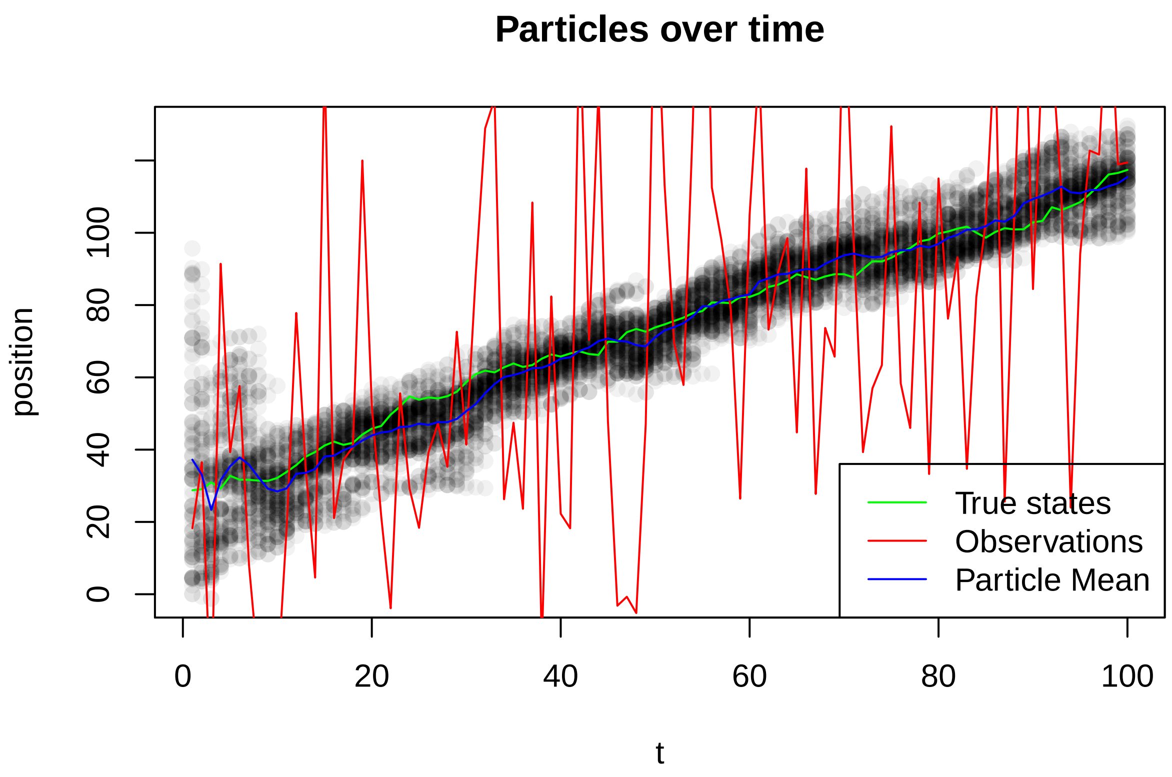

Particles over time
position
t
0
20
40
60
80
100
20
40
60
80
100
120
True states
Observations
Particle Mean

Sigma = 50

```r
sigma_emission50 = 50
sim50 = simulation(sigma_emission50)
plot_movement(sim50)
```



```
Z_particles50 = particle_filtering(sim50, sigma_emission50)
plot_particles(sim50, Z_particles50)
```

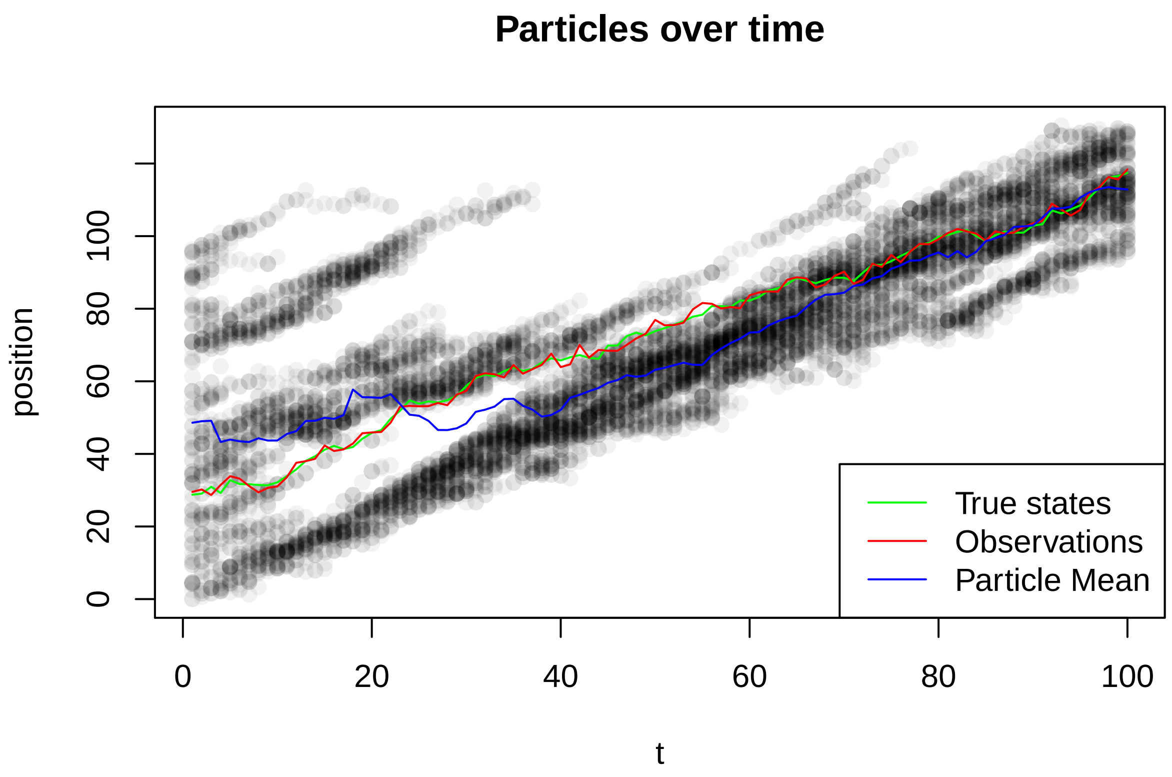


We can see that even tough the standard deviation is extremly high the general trend of the particles is following the movement of the actual robot. Note that the knowledge of the true model makes this possible, whereas observing this in reality would be hard.

Lab 3 Task 3

Equal weights for the different particles

```r
Z_particles_equal = particle_filtering(sim1, sigma_emission1, TRUE)
plot_particles(sim1, Z_particles_equal)
```



Equal weight gives no reasonable pattern in predicting the movement of the robot.

Lab 4 - Gaussian Processes

GP Regression & GP Classification

```
library("kernlab")
library(AtmRay)
```

Lab 4 Task 1 - Implementing GP Regression

```

confusion_matrix = function(y, y.pred) {
  return(table(y, y.pred, dnn=c("TRUE", "PRED")))
}

accuracy = function(y, y.pred) {
  return(1-mean(abs(y-y.pred)))
}

Visualize = function(X, y, XStar, res) {
  mean = res$mean
  sd = sqrt(res$variance)
  plot(XStar, mean, type="l", ylim=c(-3, 4),
       xlab="x", ylab="y", main="Posterior Gaussian Process")
  lines(XStar, mean - 1.96 * sd, type="l", col="blue")
  lines(XStar, mean + 1.96 * sd, type="l", col="blue")
  points(X, y, pch=19)
  legend("topright",
         legend=c("posterior mean", "probability band", "observations"),
         col=c("black", "blue", "black"), lty=c(1,1,0), pch=c(-1,-1,19))
}
```

Squared Exponential Kernel

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

1.1 Posterior of the gaussian process

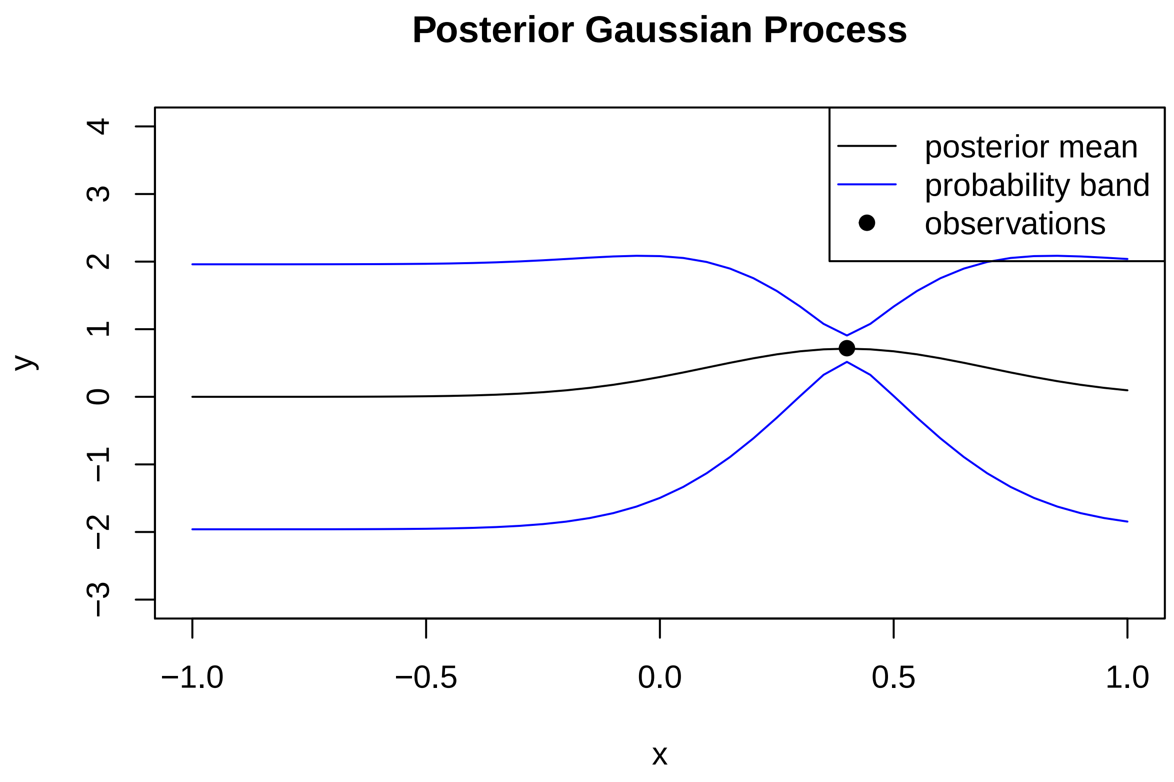
```
posteriorGP = function(X, y, XStar, sigmaNoise, K, ...) { # normalize (scale) X values if not mean 0 al
  n = length(X)
  I = diag(n)
  L = t(chol(K(X,X, ...) + sigmaNoise^2 * I))
  KStar = K(X, XStar, ...)

  # Predictive Mean
  alpha = solve(t(L), solve(L, y))
  fStarBar = t(KStar) %*% alpha

  # Predictive Variance
  v = solve(L, KStar)
  V = K(XStar, XStar, ...) - t(v) %*% v

  return(data.frame(mean = fStarBar, variance = diag(V)))
}
```

Setup interval

```
XStar = seq(-1, 1, 0.05)
sigmaN = 0.1
sigmaF = 1
l = 0.3
```

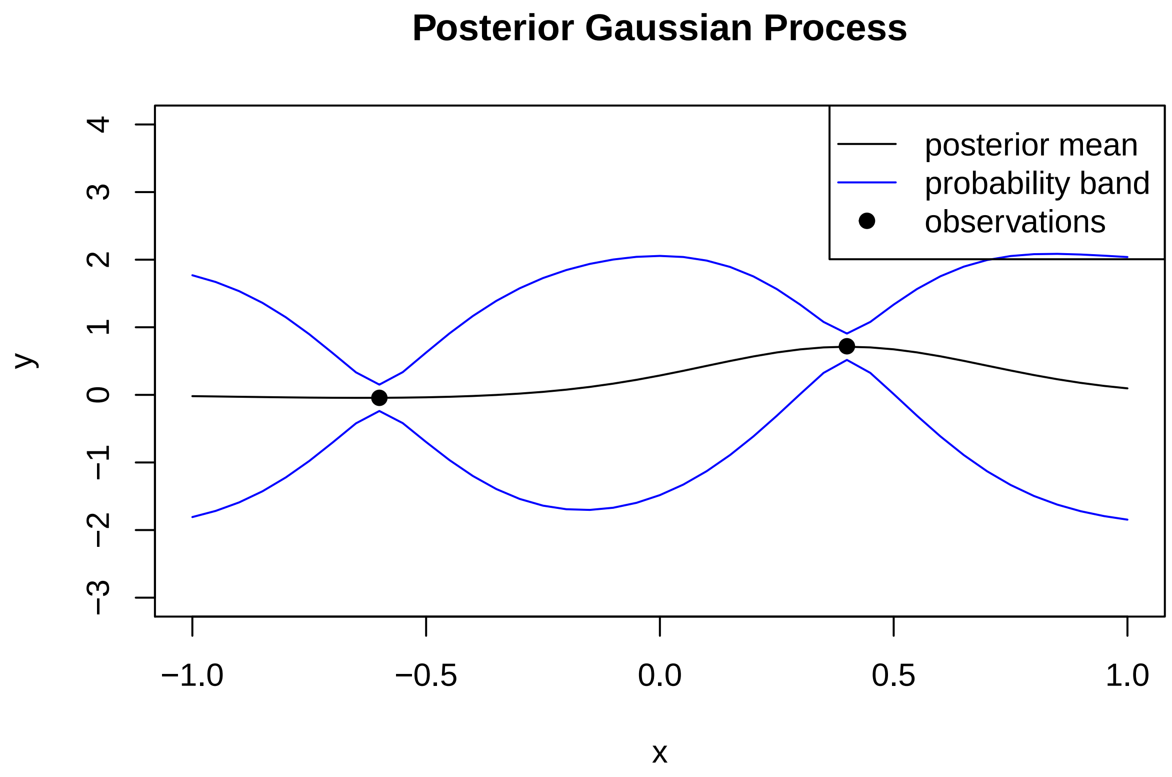
1.2 Single observation

```
X2 = c(0.4)
y2 = c(0.719)
posterior2 = posteriorGP(X2, y2, XStar, sigmaN, SquaredExpKernel, sigmaF, l)
Visualize(X2, y2, XStar, posterior2)
```



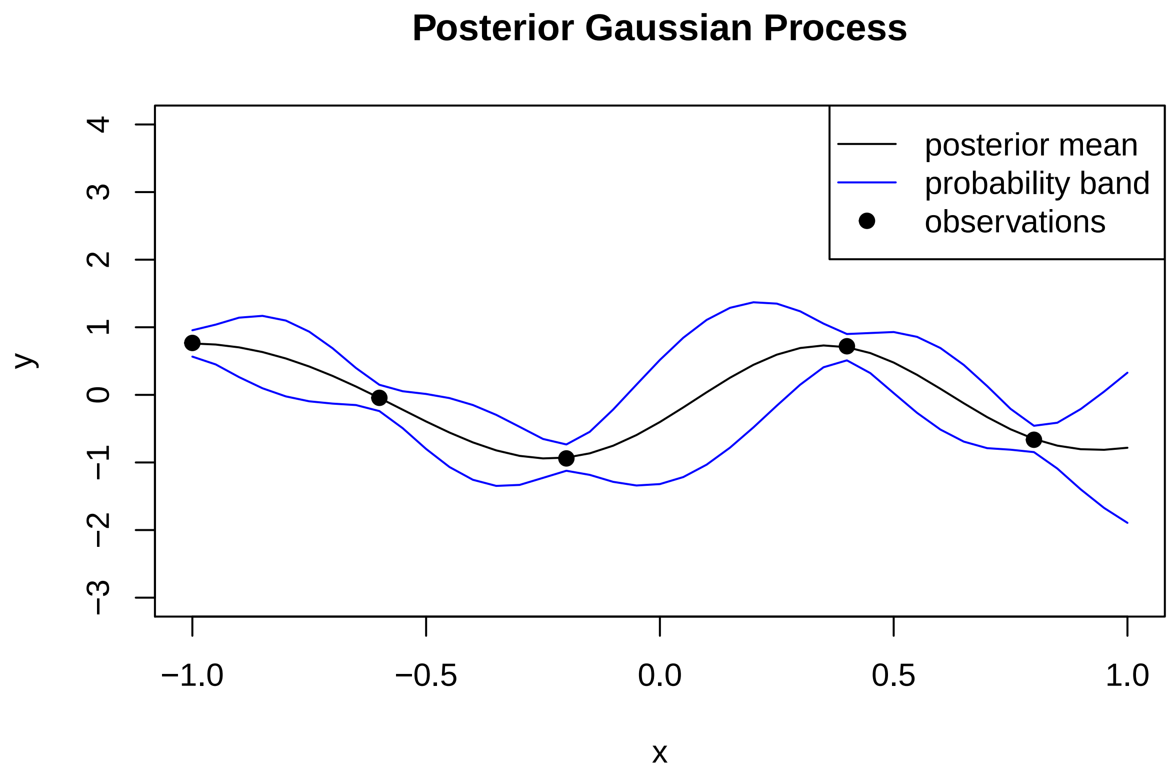
1.3 Two observations

```
X3 = c(-0.6, 0.4)
y3 = c(-0.044, 0.719)
posterior3 = posteriorGP(X3, y3, XStar, sigmaN, SquaredExpKernel, sigmaF, l)
Visualize(X3, y3, XStar, posterior3)
```



1.4 Five observations

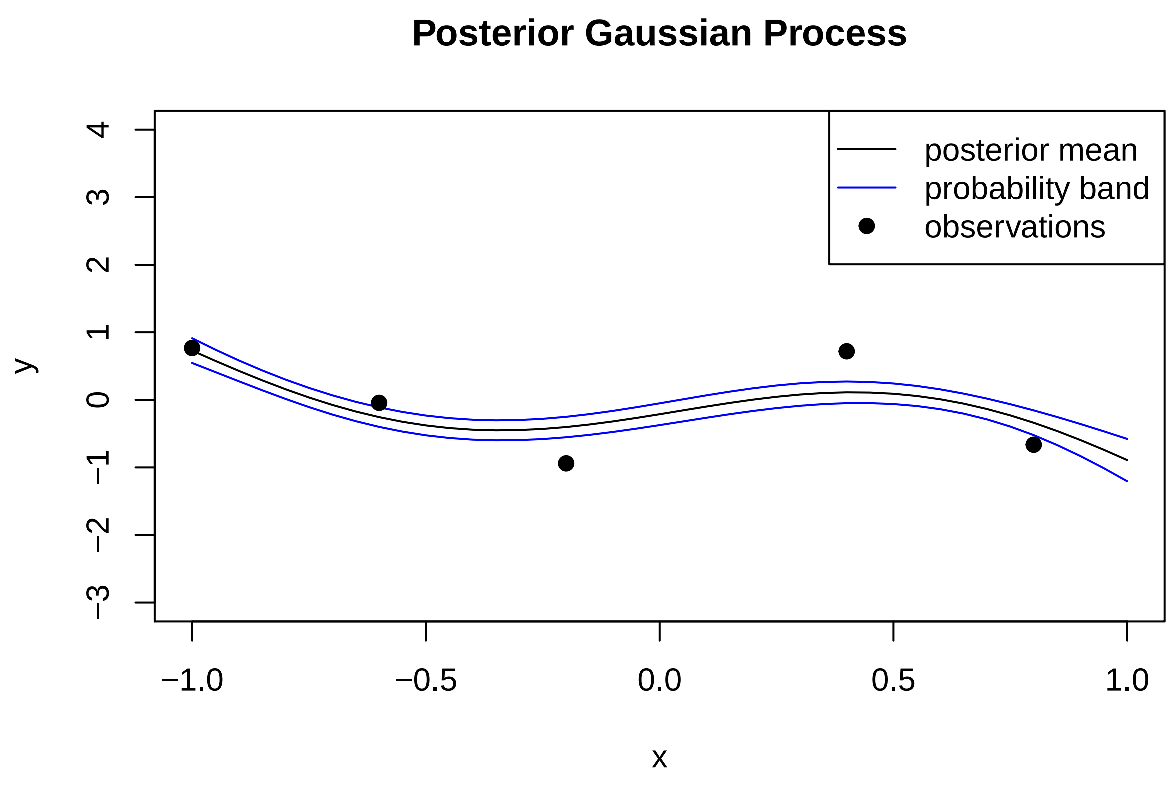
```r
X4 = c(-1.0, -0.6, -0.2, 0.4, 0.8)
y4 = c(0.768, -0.044, -0.940, 0.719, -0.664)
posterior4 = posteriorGP(X4, y4, XStar, sigmaN, SquaredExpKernel, sigmaF, l)
Visualize(X4, y4, XStar, posterior4)
```



More certaintly given more observations.

1.5 Different hyperparameters

```
X5 = c(-1.0, -0.6, -0.2, 0.4, 0.8)
y5 = c(0.768, -0.044, -0.940, 0.719, -0.664)
posterior5 = posteriorGP(X5, y5, XStar, sigmaN, SquaredExpKernel, sigmaF = 1, l = 1)
Visualize(X5, y5, XStar, posterior5)
```



Smoother fit with higher ell value. Does look like a worse fit though.

Lab 4 Task 2

Download and subsample data

```
# Download data
data.temp = read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/TempTulli
time = seq(1, 2190, 1)
day = rep(seq(1, 365, 1), 6)
data.temp["time"] = time
data.temp["day"] = day

# Subsample data
n = dim(data.temp)[1]
data.temp = data.temp[seq(1, n, 5),]
```

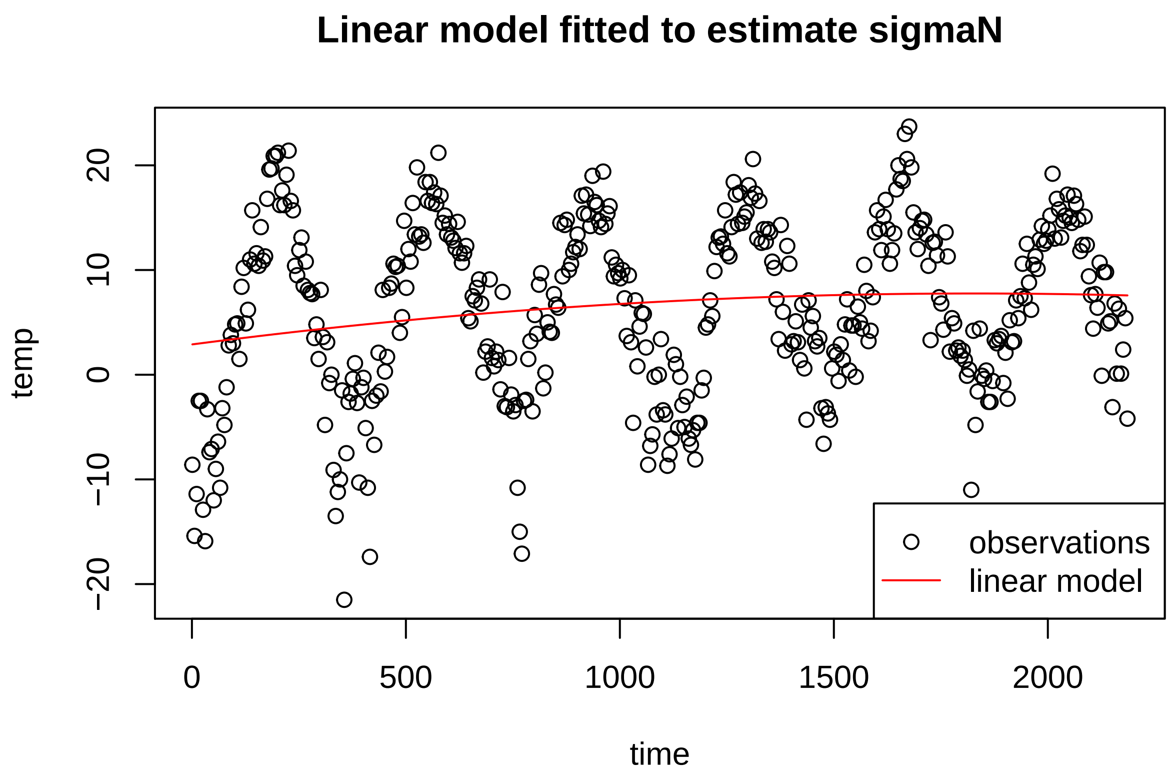
2.1

```
# SEkernel <- rbfdot(sigma = 1/(2*ell^2))
SEkernel = function(sigmaF = 1, ell = 1)
{
  SquaredExpKernel <- function(x, y = NULL) {
    # K = sigmaF^2 * exp(-0.5 * ( (x - y) / ell)^2 )
    n1 = length(x)
    n2 = length(y)
    K = matrix(NA, n1, n2)
    for(i in 1:n2) {
      for(j in 1:n1) {
        K[j,i] = sigmaF^2 * exp(-0.5 * ( (x[i] - y[j]) / ell)^2 )
      }
    }
    return(K)
  }
  class(SquaredExpKernel) <- "kernel"
  return(SquaredExpKernel)
}
k = SEkernel()

## Evaluate kernel
# x = 1, x' = 2
k(1,2)
#>           [,1]
#> [1,] 0.6065307

# X = (1, 3, 4), X' = (2, 3, 4)
kernelMatrix(k, c(1,3,4), c(2,3,4))
#> An object of class "kernelMatrix"
#>           [,1]      [,2]      [,3]
#> [1,] 0.6065307 0.1353353 0.0111090
#> [2,] 0.6065307 1.0000000 0.6065307
#> [3,] 0.1353353 0.6065307 1.0000000
```

Estimate the residual variance using a linear model

```
lm = lm(temp ~ time + I(time^2), data = data.temp)
sigmaN = sd(lm$residuals)
paste("sigmaN:", sigmaN)
```

```
#> [1] "sigmaN: 8.17628797852329"
plot(data.temp$time, data.temp$temp, main="Linear model fitted to estimate sigmaN", xlab="time", ylab="
lines(data.temp$time, lm$fitted.values, col="red")
legend("bottomright", legend=c("observations", "linear model"), lty=c(-1,1), pch=c(1,-1), col=c(1, "red
```

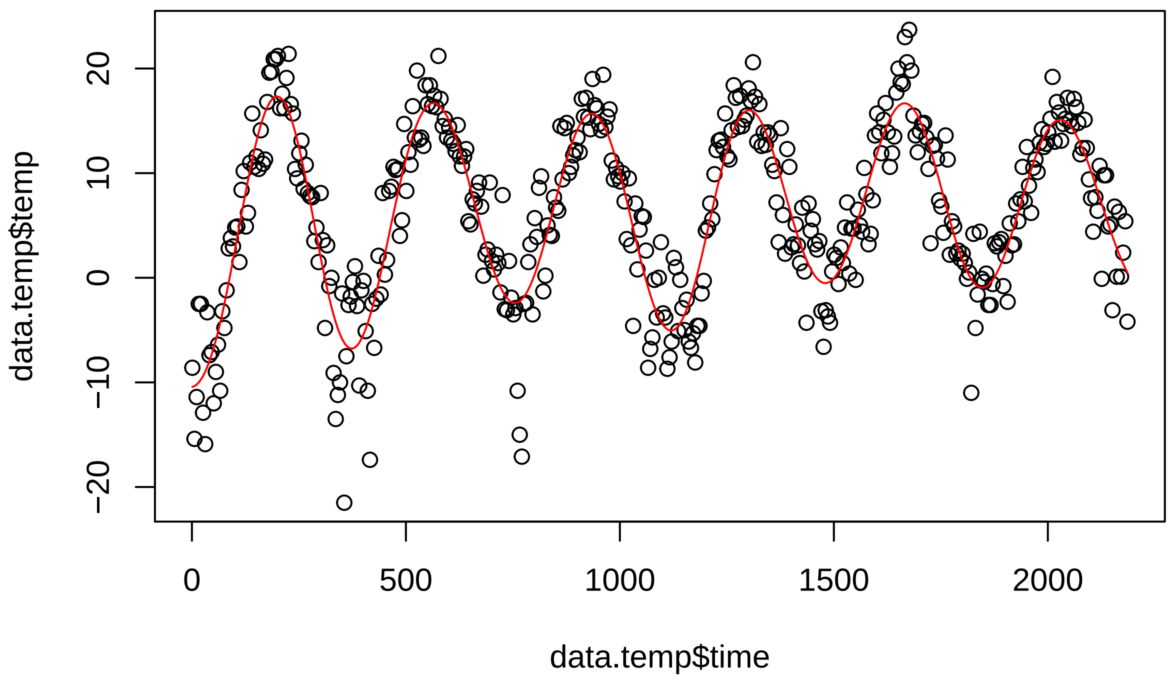


2.2 Time Estimate gaussian process using time as covariate and the squared exponential kernel

```
kernel= SEkernel(sigmaF = 20, ell = 0.2)
GPTime = gausspr(temp ~ time, data = data.temp, kernel = kernel, var = sigmaN^2)
# GPTime = gausspr(x = data.temp$time, y = data.temp$temp, kernel = SEkernel,
#                   kpar = list(sigmaF = 20, ell = 0.2), var = sigmaN^2)
```

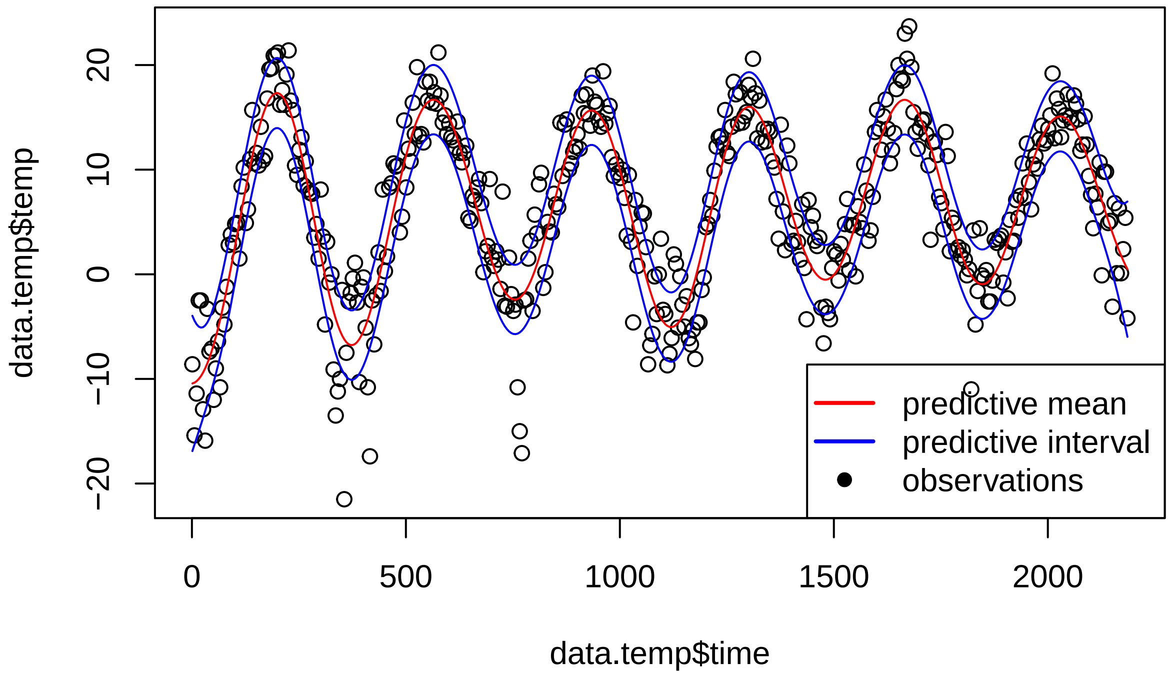
Posterior mean

```
# Posterior mean
postMeanTime = predict(GPTime, data.temp)

# Plot
plot(data.temp$time, data.temp$temp)
lines(data.temp$time, postMeanTime, col="red")
```



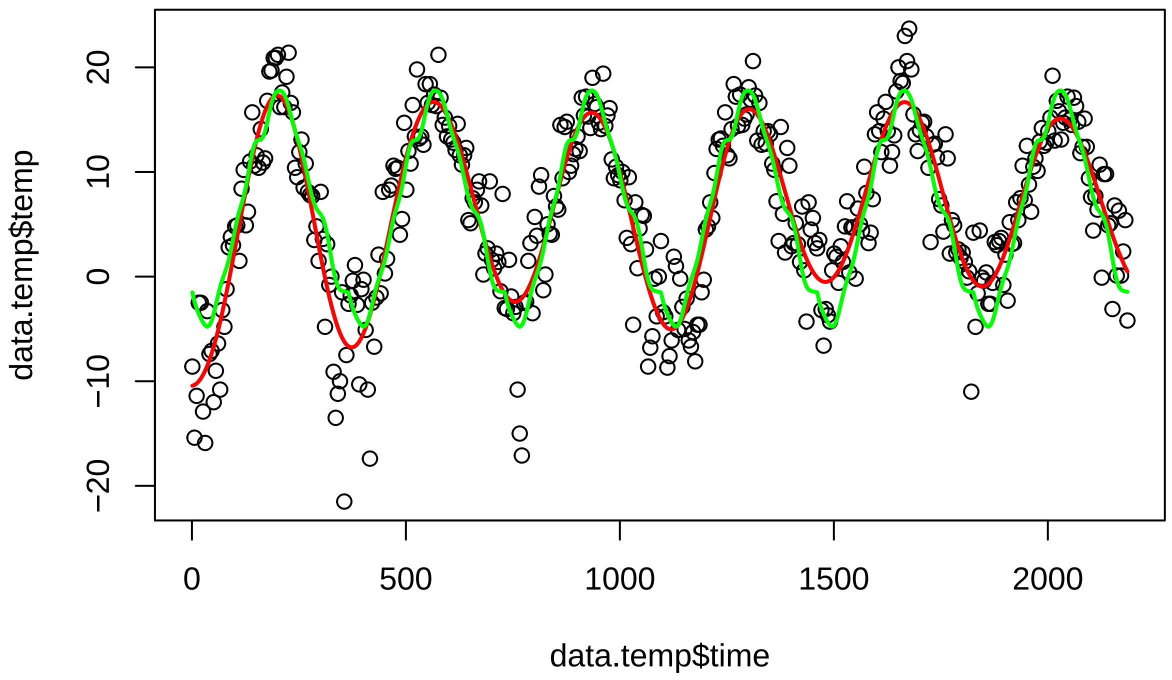
2.3 Posterior variance of f

```r
X = scale(data.temp$time)
XStar = scale(X)
n = length(X)

KStarStar = kernelMatrix(kernel = kernel, x = XStar, y = XStar)
KStar = kernelMatrix(kernel = kernel, x = X, y = XStar)
K = kernelMatrix(kernel = kernel, x = X, y = XStar)
V = diag(KStarStar - t(KStar) %*% solve(K + sigmaN^2 * diag(n), KStar))
# V = posteriorGP(X, data.temp$temp, XStar, sigmaN, kernel)$variance # same result

# Plot
plot(data.temp$time, data.temp$temp)
lines(data.temp$time, postMeanTime, col="red")
lines(data.temp$time, postMeanTime - 1.96 * sqrt(V), col="blue")
lines(data.temp$time, postMeanTime + 1.96 * sqrt(V), col="blue")
legend("bottomright", legend=c("predictive mean", "predictive interval", "observations"),
       col=c("red", "blue", "black"), lty=c(1,1,0), pch=c(-1,-1,19), lwd=c(2,2,0))
```



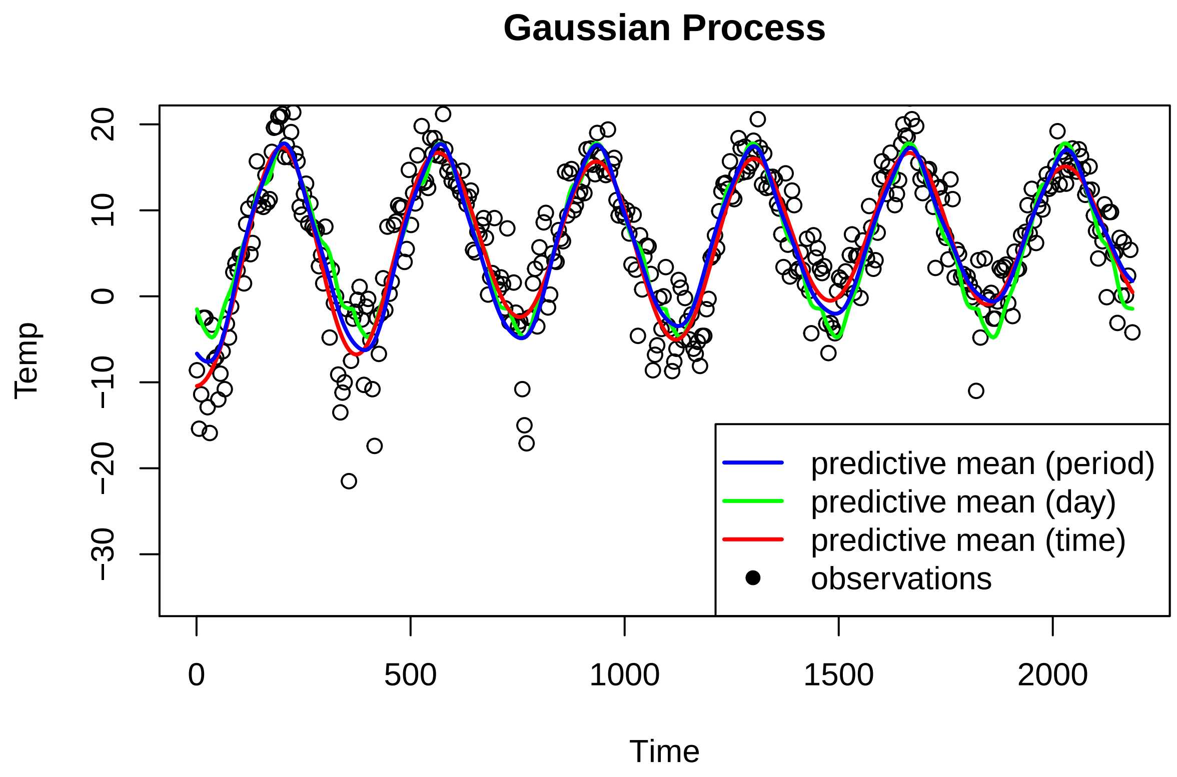
2.4 Day

```
GPDay = gausspr(temp ~ day, data = data.temp, kernel = kernel, var = sigmaN^2)

postMeanDay = predict(GPDay, data.temp)
plot(data.temp$time, data.temp$temp)
lines(data.temp$time, postMeanTime, col="red", lwd=2)
lines(data.temp$time, postMeanDay, col="green", lwd=2)
```



2.5 Peroidic kernel

```
# Period Kernel function
Pkernel = function(sigmaF = 1, l1 = 1, l2 = 1, d)
{
  PeriodicKernel <- function(x, y) {
    K = sigmaF^2 *
      exp(-2 * sin(pi * abs(y - x) / d)^2 / l1^2 ) *
      exp(-0.5 * (y - x)^2 / l2^2 )
    return(K)
  }
  class(PeriodicKernel) <- "kernel"
  return(PeriodicKernel)
}
d = 365 / sd(time)
kernel = Pkernel(sigmaF = 20, l1 = 1, l2 = 10, d)
GPPeriodic = gausspr(temp ~ time, data = data.temp, kernel = kernel, var = sigmaN^2)
postMeanPeriodic = predict(GPPeriodic, data.temp)
plot(data.temp$time, data.temp$temp, ylim=c(-35, 20),
     main="Gaussian Process", xlab="Time", ylab="Temp")
lines(data.temp$time, postMeanDay, lwd=2, col="green")
lines(data.temp$time, postMeanTime, lwd=2, col="red")
lines(data.temp$time, postMeanPeriodic, lwd=2, col="blue")
legend("bottomright",
       legend=c("predictive mean (period)", "predictive mean (day)", "predictive mean (time)", "observa
       col=c("blue", "green", "red", "black"), lty=c(1,1,1,0),
       pch=c(-1,-1,-1,19), lwd=c(2,2,2,0))
```



Here we use the general periodic kernel, which has the advantages that it captures both the seasonal pattern and the long term trend over time. After observing the plot an rough estimate can be that the periodic kernel lies between the Day and Time model.

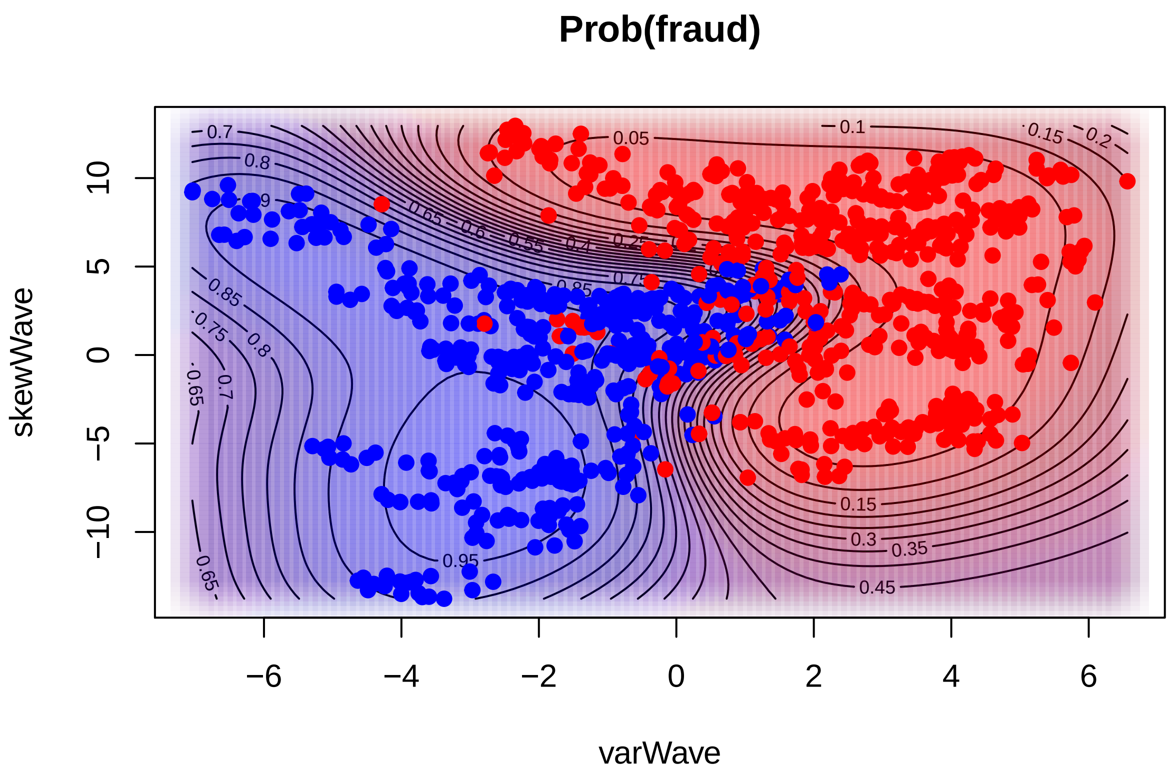
Lab 4 Task 3 - Gaussian Process for classification

Setting up data, 1000 observations for the train data and the rest for testing

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

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

3.1

```
set.seed(123)
fit = gausspr(fraud ~ varWave + skewWave, data = train)
#> Using automatic sigma estimation (sigest) for RBF or laplace kernel

# Class predictions on train data
pred = predict(fit, train)
probs = predict(fit, train, type="probabilities")
preds = ifelse(probs[,2] > 0.5, 1, 0)
CM = confusion_matrix(train$fraud, preds)
#     PRED
# TRUE   0   1
# 0    512  44
# 1     24 420
accuracy(as.integer(train$fraud)-1, preds)
#> [1] 0.932
# 0.932

# Plot Contour
plot_contour = function(fit, train) {
  # Grid setup
  x1 = seq(min(train$varWave), max(train$varWave), length=100)
  x2 = seq(min(train$skewWave), max(train$skewWave), length=100)
  grid = meshgrid(x1, x2)
  grid <- data.frame(cbind(c(grid$x), c(grid$y)))
  names(grid) = c("varWave", "skewWave")
  probsGrid = predict(fit, grid, type="probabilities")

  # Plot grid
  contour(x1, x2, matrix(probsGrid[,2], 100, byrow=TRUE), 20,
          xlab = "varWave", ylab = "skewWave", main = "Prob(fraud)")
  points(grid, col=rgb(1-probsGrid[,2], 0, probsGrid[,2], 0.0125), pch=15, cex=3)
  points(train$varWave, train$skewWave,
         col=rgb(2-as.integer(train$fraud), 0, as.integer(train$fraud)-1), pch=19)
}
plot_contour(fit, train)
```



3.2

```
# Class predictions on test data
probs = predict(fit, test, type="probabilities")
preds = ifelse(probs[,2] > 0.5, 1, 0)
CM = confusion_matrix(test$fraud, preds)
#      PRED
# TRUE    0   1
#    0 191  15
#    1   9 157
accuracy(as.integer(test$fraud)-1, preds)
#> [1] 0.9354839
# 0.9354839
# Comment: High accuracy for the test data, more or less the same as the accuracy for the training data
```

3.3

```
fit = gausspr(fraud ~ ., data = train)
#> Using automatic sigma estimation (sigest) for RBF or laplace kernel
# Class predictions on test data
probs = predict(fit, test, type="probabilities")
preds = ifelse(probs[,2] > 0.5, 1, 0)
CM = confusion_matrix(test$fraud, preds)
#      PRED
# TRUE    0   1
#    0 205   1
#    1   0 166
```

```r
kable(accuracy(as.integer(test$fraud)-1, preds), caption="Accuracy on test data")
```

Table 3: Accuracy on test data

x
0.9973118

```r
# 0.9973118
# Comment: Extremly high accuracy, performs better than the model fitted using only two covariates,
# Overfitting does not seem to be an issue
```