Lab 1 - Probabilistic Graphical Models

Arvid Edenheim - arved490

2019-07-27

1

I generated two networks with the hill-climbing algorithm using two different score functions: log likhelihood and bayesian information criterion. Due to the lack of parameter punishment in the log likelihood, it generated the complete graph as output. The two networks are clearly non equivalent.

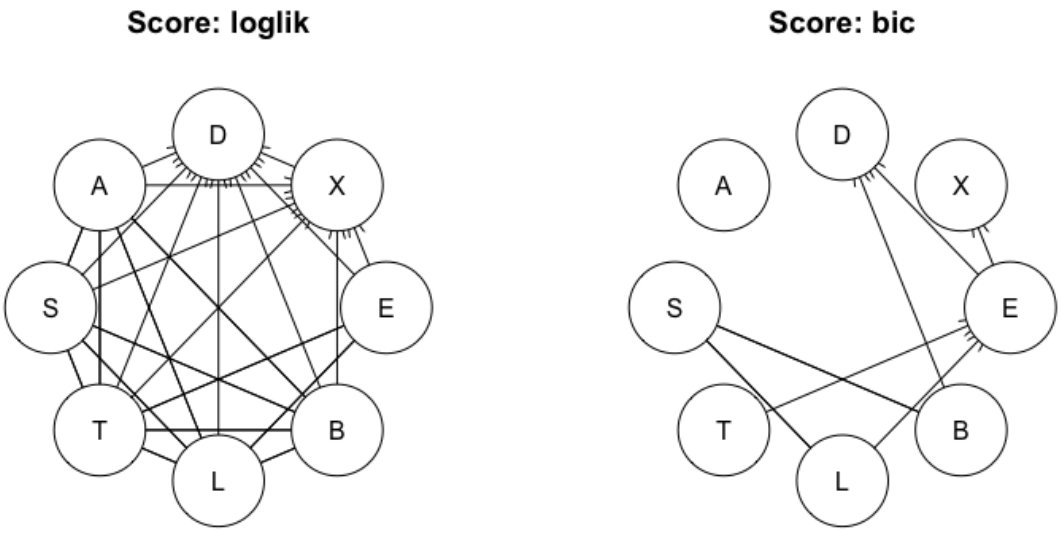


Figure 1: Generated networks from the hill-climb algorithm

```
###### Assignment 1 ######
# Show non-equivalent BN structures for multiple runs of the HC algorithm

data('asia')

par(mfrow=c(1,2))

# With log-likelihood as score function
set.seed(1)
hc_loglik <- hc(x = asia, score='loglik')
plot(hc_loglik)
hc_loglik <- cpdag(hc_loglik)
plot(hc_loglik, main='Score: loglik') # Generates complete network due to lack of parameter punishment

# With bayesian information criterion as score function
set.seed(1)
hc_bic <- hc(x = asia, score = 'bic')
hc_bic <- cpdag(hc_bic)
plot(hc_bic, main='Score: bic')

# Check equality
print(all.equal(hc_loglik, hc_bic))
# Nope
```

2

The resulting network generated an identical confusion matrix to the true network. Looking at the plots, this is most likely due to the same structure of outgoing edges from S, which was predicted on. If this were to be changed to node A, it would most likely generate conflicting results. The overall accuracy wasn't great, so if we only were interested in lowering either the false positive or false negative, the evaluated posterior probability should be compared to a more biased value than 0.5, f.i. setting it to 0.9 would make the model classify true only when it's very certain, drastically decreasing the false positives (but ofc increasing the false negatives).

Confusion Matrix - True BN

	False	True
False	341	157
True	119	383

Confusion Matrix - generated BN

	False	True
False	341	157
True	119	383

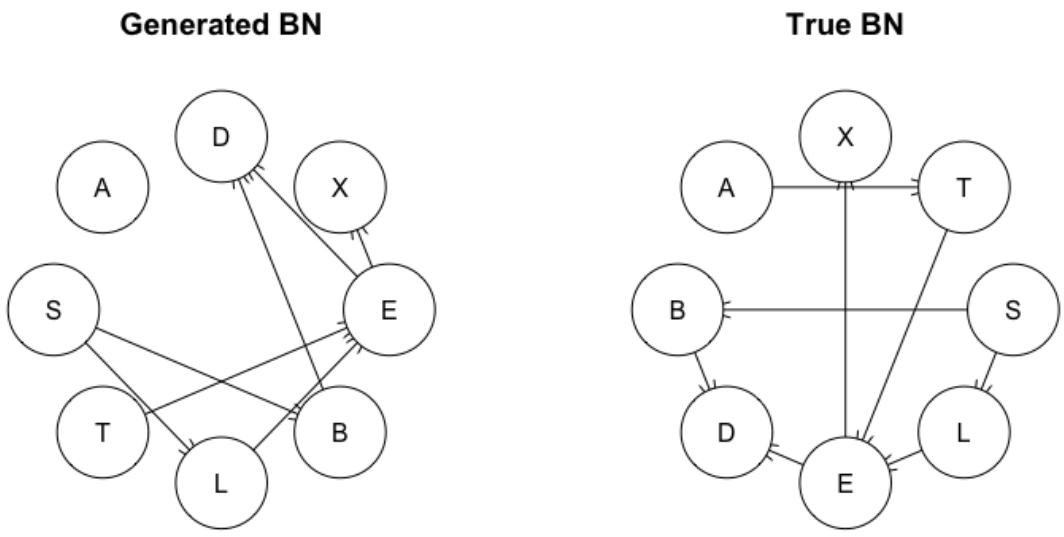


Figure 2: Network comparison

```
###### Assignment 2 ######
# Learn a BN from 80% of the data and classify on the rest, compare with true BN

confusionMatrix <- function(BN, data, obs_var, pred_var) {
  predictions <- rep(0, nrow(data))
  for(i in 1:nrow(data)) {
    X <- NULL
    for(j in obs_var) {
      X[j] <- if(data[i,j] == 'yes') 'yes' else 'no'
    }
    # X <- data[i,obs_var] didn't work, even though it returns the same as the loop above
    find <- setEvidence(object=BN, nodes=obs_var, states=X) # Recommended over setFinding
    dist <- querygrain(object=find, nodes=pred_var)[[pred_var]]
    predictions[i] <- if(dist['yes'] >= 0.5) 'yes' else 'no'
  }
  return (table(data[,pred_var], predictions, dnn=c("TRUE", "PRED")))
}

set.seed(123)
samples <- sample(1:nrow(asia), size = floor(0.8*nrow(asia)), replace = FALSE)
train <- asia[samples, ]
test  <- asia[-samples, ]

# Create networks
BN_train <- hc(train, restart=3, score = 'bic')
BN_true = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")

par(mfrow=c(1,2))
plot(BN_train, main='Generated BN')
plot(BN_true, main='True BN')

# Convert to grain and run Lauritzen-Spiegelhalter
BN_train <- compile(as.grain(bn.fit(BN_train, train)))
BN_true <- compile(as.grain(bn.fit(BN_true, train)))

# Predict and generate confusion matrices
conf_matrix <- confusionMatrix(BN_train,
                               data=test,
                               obs_var=c('A', 'D', 'X', 'E', 'B', 'L', 'T'),
                               pred_var='S')
conf_matrix_true <- confusionMatrix(BN_true,
                                    data=test,
                                    obs_var=c('A', 'D', 'X', 'E', 'B', 'L', 'T'),
                                    pred_var='S')

# How can they be identical? Am I a pro? No: P(B|S)P(L|S)
```

3

The confusion matrices were equal to those in 2), which supports the d-separation theorem for markov blankets. As described in 2), the markov blanket for both graphs are identical, yielding the similar results.

$MB_{hc}(S) = MB_{true}(S) = [B, L]$

Confusion Matrix - True BN

	False	True
False	341	157
True	119	383

Confusion Matrix - generated BN

	False	True
False	341	157
True	119	383

4

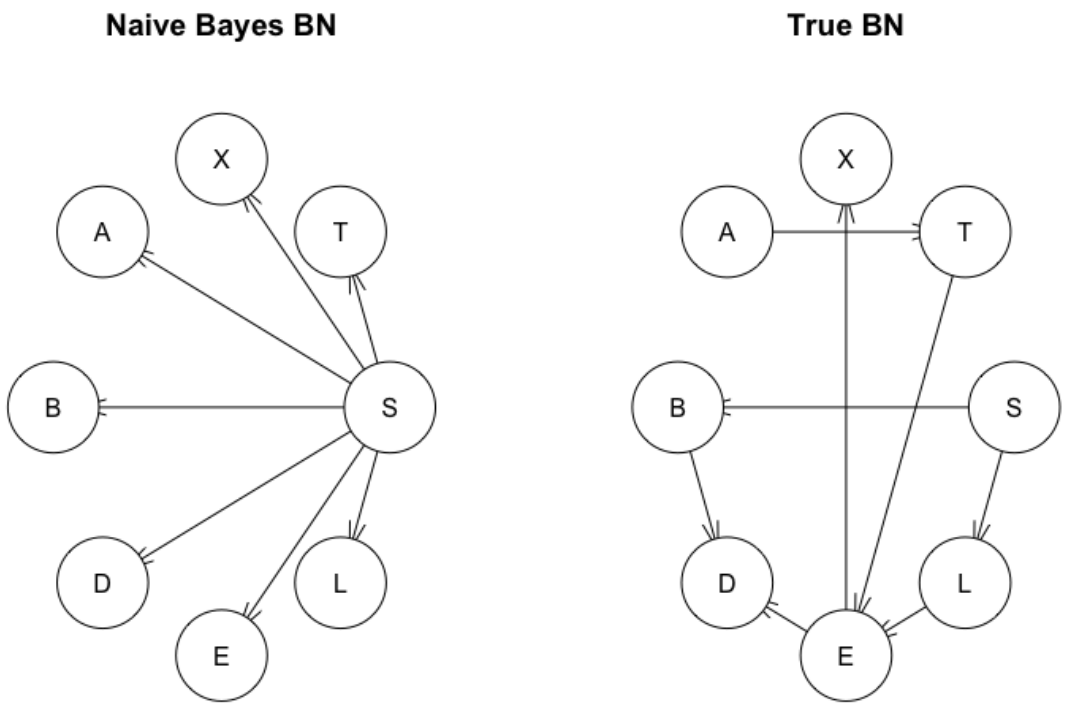


Figure 3: Network comparison

Confusion Matrix - Naive Bayes

	False	True
False	374	124
True	181	321

Confusion Matrix - generated BN

	False	True
False	341	157
True	119	383

```
###### Assignment 4 ######

# Create a naive bayes classifier on S
# Naive bayes, wiki: "The value of a particular feature is independent
# of the value of any other feature, given the class variable. For example,
```

```r
# a fruit may be considered to be an apple if it is red, round,
# and about 10 cm in diameter."

set.seed(123)
samples <- sample(1:nrow(asia), size = floor(0.8*nrow(asia)), replace = FALSE)
train <- asia[samples, ]
test  <- asia[-samples, ]

# Nodes: S, c('A', 'D', 'X', 'E', 'B', 'L', 'T')

BN_train = model2network("[S][A|S][D|S][X|S][E|S][B|S][L|S][T|S]")
BN_true = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")

par(mfrow=c(1,2))
plot(BN_train, main='Naive Bayes BN')
plot(BN_true, main='True BN')

# Convert to grain and run Lauritzen-Spiegelhalter
BN_train <- compile(as.grain(bn.fit(BN_train, train)))
BN_true <- compile(as.grain(bn.fit(BN_true, train)))

# Predict and generate confusion matrices
conf_matrix <- confusionMatrix(BN_train,
                               data=test,
                               obs_var=c('A', 'D', 'X', 'E', 'B', 'L', 'T'),
                               pred_var='S')
conf_matrix_true <- confusionMatrix(BN_true,
                                    data=test,
                                    obs_var=c('A', 'D', 'X', 'E', 'B', 'L', 'T'),
                                    pred_var='S')
```

5

A short comparison between 2) & 3) can be found in 3).

The reason the results differ is reasonably due to the vastly different structure, resulting in a different markov blanket, with all other nodes appearing as conditionally independent children given S.

Lab 2 - Hidden Markov Models

Arvid Edenheim - arved490

2019-08-18

1 - Model

The transition probability $p(x_{t+1}|x_t) = 0.5$ & $p(x_t|x_t) = 0.5$.

The emission/observation probability $p(y_{t-2}|x_t) = p(y_{t-1}|x_t) = p(y_t|x_t) = p(y_{t+1}|x_t) = p(y_{t+2}|x_t) = 0.2$.

This translates into the following matrices:

Transition matrix

	1	2	3	4	5	6	7	8	9	10
1	0.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.5	0.5	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.5	0.5	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	0.0	0.0	0.5	0.5	0.0	0.0	0.0
7	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.5	0.0	0.0
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.5	0.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.5
10	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5

Emission matrix

	1	2	3	4	5	6	7	8	9	10
1	0.2	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.2	0.2
2	0.2	0.2	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.2
3	0.2	0.2	0.2	0.2	0.2	0.0	0.0	0.0	0.0	0.0
4	0.0	0.2	0.2	0.2	0.2	0.2	0.0	0.0	0.0	0.0
5	0.0	0.0	0.2	0.2	0.2	0.2	0.2	0.0	0.0	0.0
6	0.0	0.0	0.0	0.2	0.2	0.2	0.2	0.2	0.0	0.0
7	0.0	0.0	0.0	0.0	0.2	0.2	0.2	0.2	0.2	0.0
8	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.2	0.2	0.2
9	0.2	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.2	0.2
10	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.2

```
############### 1 ###############

set.seed(1234)
nStates <- 10
states = symbols <- 1:10

getNoiseInterval <- function(index, nStates, noise) {
  start <- index - noise
  end <- index + noise
  interval <- start:end %% nStates
  interval[interval==0] <- nStates
  return(interval)
}

# Initialize matrix with transition probabilities.
# P(X_1|X) = 0.5, P(X|X) = 0.5
tProb <- 0.5
transProbs <- tProb * diag(nStates)
for(i in 1:nStates) {
  transProbs[i,(i %% nStates + 1)] <- tProb
}

# Initialize emission probabilities.
# P(Y-2|X) = P(Y-1|X) = P(Y|X) = P(Y+1|X) = P(Y+2|X) = 0.2
eProb <- 0.2
emissionProbs <- matrix(0, nrow = nStates, ncol = nStates)
noise <- 2
for(i in 1:nStates) {
  emissionProbs[i,getNoiseInterval(i, nStates, noise)] <- eProb
}

HMM <- initHMM(States = states,
               Symbols = symbols,
               transProbs = transProbs,
               emissionProbs = emissionProbs)
```

2 - Simulation

Simulation of 100 timesteps was trivial using the HMM library.

```
############### 2 ###############

nSteps <- 100
sims <- simHMM(HMM, nSteps)
```

3 - Forward/Backward and Path Calculation

```
############### 3 ###############

# Exponential due to forward/backward returning log values
alpha <- exp(forward(HMM, sims$observation))
beta <- exp(backward(HMM, sims$observation))

# Normalizing at each time step to prevent underflow (?)
# and to, you know, normalize to obtain a prob. distr.
# Basically performs col[,i] / sum(col[,i]) to get the normalized distr.
filter <- prop.table(alpha, margin = 2)
smoother <- prop.table(alpha*beta, margin = 2)

# Evaluate most probable path using viterbi.
viterbiPath = viterbi(HMM, sims$observation)

# Generate path by maximizing the marginal probability for each individual timestep
filterPath <- apply(filter, MARGIN = 2, which.max)
smootherPath <- apply(smoother, MARGIN = 2, which.max)
```

4 - Accuracy

```
############### 4 ###############

accuracy <- function(path, states) {
  return(sum(diag(table(path, states)) / length(path)))
}

# 0.49
viterbiAccuracy = accuracy(viterbiPath, sims$states)

# 0.63
filterAccuracy =  accuracy(filterPath, sims$states)

# 0.75
smootherAccuracy =  accuracy(smootherPath, sims$states)
```

5 - Verifying

Repeated trials yields the same result as in 4). The smoothed distribution appears to have a higher accuracy than both the most probable path and the filtered distribution. This is natural since the smoother is the joint probability of both future and past observations, thus containing more information than the filtered distribution. The viterbi algorithm only calculates the most probable sequence, which intuitively always will be less accurate than the path obtained by maximizing the marginal probability for each individual timestep, regardless of neighboring states.

```
############### 5 ###############

sims <- simHMM(HMM, nSteps)

alpha <- exp(forward(HMM, sims$observation))
beta <- exp(backward(HMM, sims$observation))

filter <- prop.table(alpha, margin = 2)
smoother <- prop.table(alpha*beta, margin = 2)

viterbiPath = viterbi(HMM, sims$observation)
filterPath <- apply(filter, MARGIN = 2, which.max)
smootherPath <- apply(smoother, MARGIN = 2, which.max)

# 0.55
viterbiAccuracy = accuracy(viterbiPath, sims$states)

# 0.58
filterAccuracy =  accuracy(filterPath, sims$states)

# 0.73
smootherAccuracy =  accuracy(smootherPath, sims$states)

# Smoothing appears to perform better at evaluating the hidden states
```

6 - Additional observations

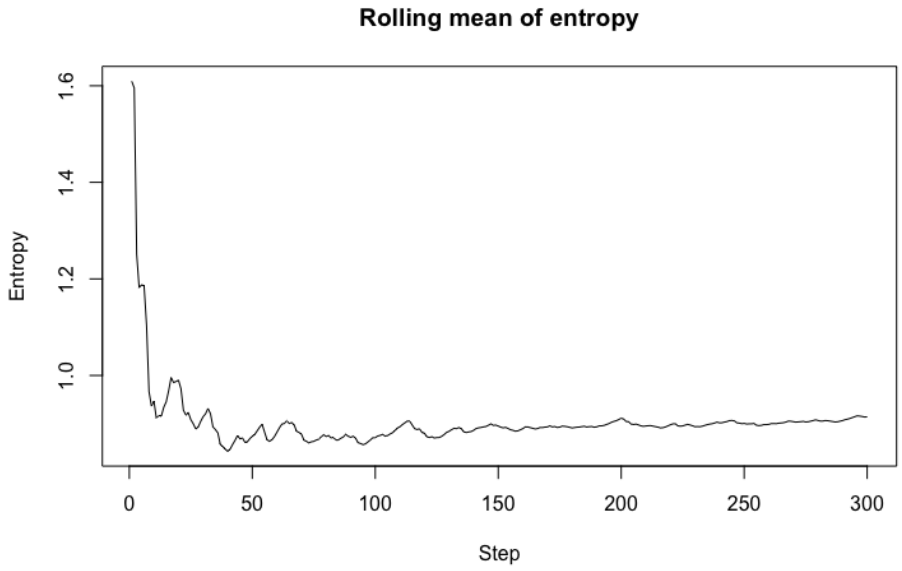


Figure 1: Entropy

I calculated the entropy of the filtered distributions over 300 timesteps and evaluated a rolling mean from timestep 1 to $i \in [1:300]$. After repeated trials, the mean appeared to converge after 70-100 iterations, which would imply that further observations don't necessarily provide additional information.

```
############### 6 ###############

# With 300 time steps
nSteps <- 300
sims300 <- simHMM(HMM, nSteps)
alpha300 <- exp(forward(HMM, sims300$observation))
filter300 <- prop.table(alpha300, margin = 2)

entropy300 <- apply(filter300, MARGIN = 2, entropy.empirical)

# Evaluate rolling mean from step 1 to current step i
rollingMeanEntropy <- rep(0, nSteps)
for(i in 1:nSteps) {
  rollingMeanEntropy[i] <- mean(entropy300[1:i])
}

plot(rollingMeanEntropy,
     type='l',
     main="Rolling mean of entropy",
     ylab = 'Entropy',
     xlab='Step')

# Comment: Converges after approx 70-100 steps.
# More observations does not appear to provide further information
```

7 - Step 101

The state vector for the filtered distribution at timestep 100

$X_{100} =$

$$\begin{bmatrix} 0.654 & 0.345 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \end{bmatrix}$$

Which, when multiplied with the transition matrix from 1) yields the probability distribution for the hidden state at timestep 101,

$X_{101} =$

$$\begin{bmatrix} 0.327 & 0.500 & 0.173 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \end{bmatrix}$$

```
############### 7 ###############

hiddenStates101 <- colSums(filter[,100] * transProbs)

# P(X_101 = 1) = 0.33, P(X_101 = 2) = 0.50, P(X_101 = 3) = 0.17
# given filter[,100] = [0.65, 0.35, 0, ...]
```

Lab 3 - State Space Models

Arvid Edenheim - arved490

2019-08-18

Setup

```
############## Function setup ###############

# Returns emission probabilities
# Returns density if density = TRUE
emissionModel <- function(mean,
                          sigma=1,
                          nSamples=1,
                          density=FALSE,
                          X,
                          noise=-1:1) {

  return(sum(sapply(noise,
                    function(n)
                      if(density)
                        dnorm(X, mean + n, sigma)
                      else
                        rnorm(nSamples, mean + n, sigma))) / 3)
}

# Returns transition probabilities
transitionModel <- function(mean,
                            transition=TRUE,
                            sigma=1,
                            nSamples=1,
                            noise=0:2) {

  return(sum(sapply(noise,
                    function(n) rnorm(nSamples,
                                      mean + n,
                                      sigma))) / 3)
}

# Yeah, this was super necessary
sampleUnif <- function(N, start=0, end=100) {
  return(runif(N, start, end))
}

# M = number of particles to sample
# T = number of timesteps, i.e. observations
# X = Observations from emission model
# Z = latent states
# zPred = belief for Z step t after step t-1
# If constant, the emission probability will be fixed to 1
particleFilter <- function(M, T, X, emissionSigma=1, constant=FALSE) {
  zPred <- matrix(0, nrow = T, ncol = M)
  Z <- matrix(0, nrow = T, ncol = M)
  w <- rep(0, M)

  # Initial belief
  Z[1,] <- sampleUnif(N=M)
  zPred[1,] <- sapply(Z[1,], function(x) transitionModel(mean=x))
```

```
  # The following loop is implemented according to the PF algorithm from lecture 8
  for(t in 2:T) { # Row

    # Prediction
    for(m in 1:M) { # Col
      # Sample Z_t^m as p(z_t|z_t-1^m) from transition model & add to zPred
      # calculate weight w_t as p(x|z_t^m) from emission model (density)
      zPred[t, m] = transitionModel(mean=Z[t-1, m])
      w[m] <- if(constant) 1 else emissionModel(mean = zPred[t, m],
                                                density = TRUE,
                                                X=X[t],
                                                sigma = emissionSigma)
    }

    # Correction
    # Sample with replacement from zPred with prob. prop. to w_t & add to Z
    Z[t,] = sample(x = zPred[t,], replace = TRUE, size = M, prob = w)
  }
  return(Z)
}

# Simulate observations
simulate <- function(T, emissionSigma=1) {
  states <- rep(0, T)
  observations <- rep(0, T)

  # First iteration
  states[1] <- sampleUnif(1)
  observations[1] <- emissionModel(mean = states[1], sigma = emissionSigma)

  for(t in 2:T) {
    states[t] <- transitionModel(mean = states[t-1])
    observations[t] <- emissionModel(mean = states[t], sigma = emissionSigma)
  }

  return(list(states=states, observations=observations))
}

# Plots histogram of particles &
# a comparison between the true state and particles, for timestep t
# Also calculates the Mean Absolute Error
plotParticles <- function(particles, simulations, step) {
  Z <- simulations$states[step]
  par(mfrow=c(2,1), mar=c(2,2,2,2))
  hist(particles[step,], 25, xlab = '', main = paste('Step', step))
  legend('topright',
         legend = c('True state', 'Particle mean'),
         fill = c('red', 'green'))
  abline(v=Z, col='red')
  abline(v=mean(particles[step,]), col='green')
  plot(particles[step,],
       xlab = '',
       ylim = c(min(particles[step,])*0.95, max(particles[step,])*1.05))
```

```
  abline(h=Z, col='red')
  abline(h=mean(particles[step,]), col='green')
  legend('bottomright', legend = c(
    paste('MAPE',eval(round(mean(abs(Z - particles[step,])/Z), 5)), '%'),
    paste('MAE',eval(round(mean(abs(Z - particles[step,])), 5)))))
}

# Plot the residual between the particles and the true values
plotResidual <- function(particles, simulations, sigma) {
  res <- rowMeans(particles) - simulations$states
  plot(res[2:length(res)], type='l', ylim=c(-20, 20), main=paste('sd = ', eval(sigma)))
}
```

Model

Transition model

$p(z_t|z_{t-1}) = \frac{\mathcal{N}(z_t|z_{t-1},1)+\mathcal{N}(z_t|z_{t-1}+1,1)+\mathcal{N}(z_t|z_{t-1}+2,1)}{3}$

Emission model

$p(x_t|z_t) = \frac{\mathcal{N}(x_t|z_t,1)+\mathcal{N}(x_t|z_t-1,1)+\mathcal{N}(x_t|z_t+1,1)}{3}$

Initial model

$p(z_1) = Uniform(0, 100)$

1

I plotted the particles at timestep 1, 25, 75 and 100 and evaluated the MAE as well as MAPE score in order to track the progress.

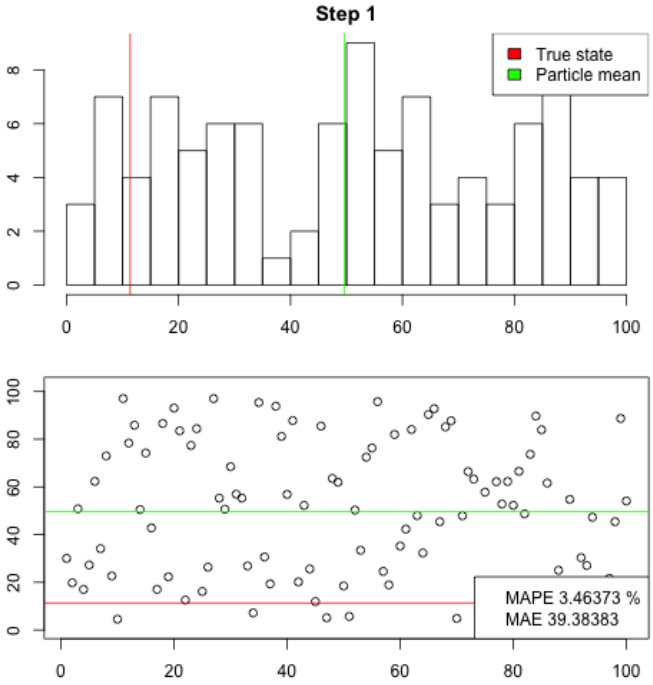


Figure 1: Particles at step 1

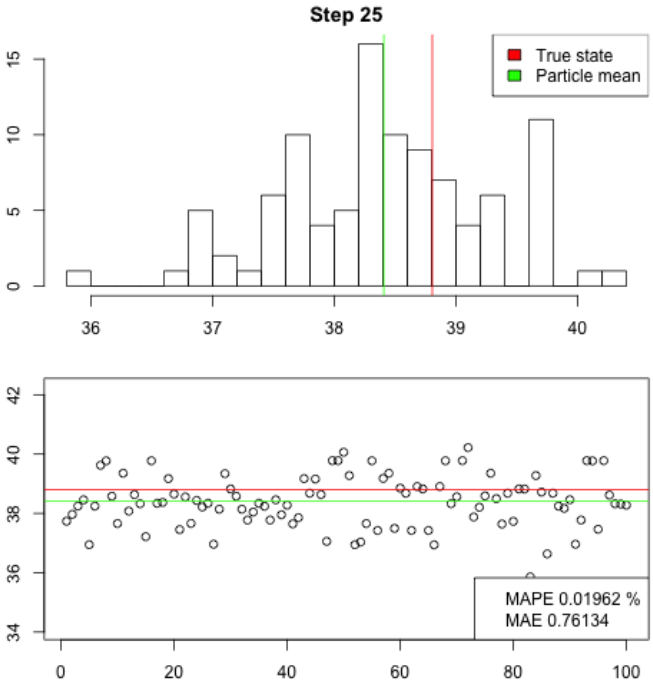


Figure 2: Particles at step 25

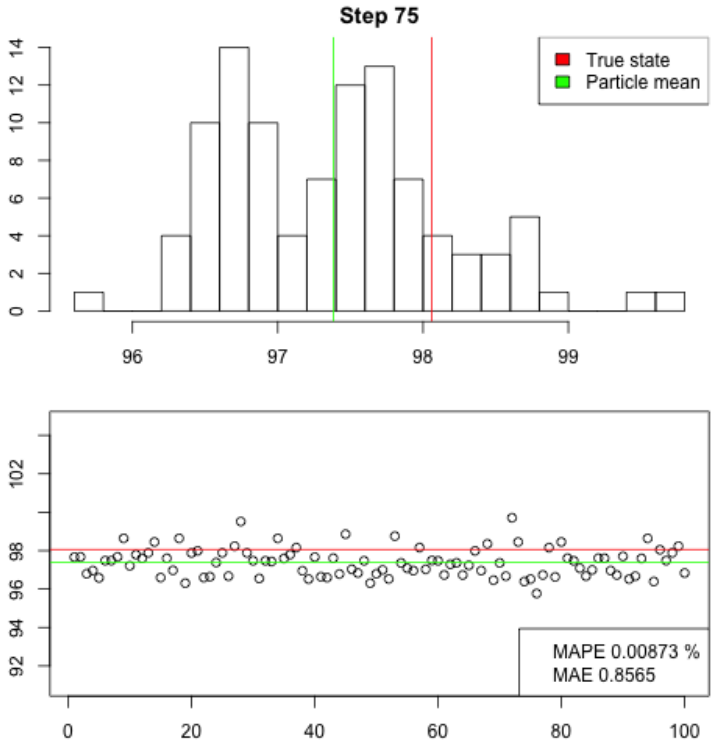


Figure 3: Particles at step 75

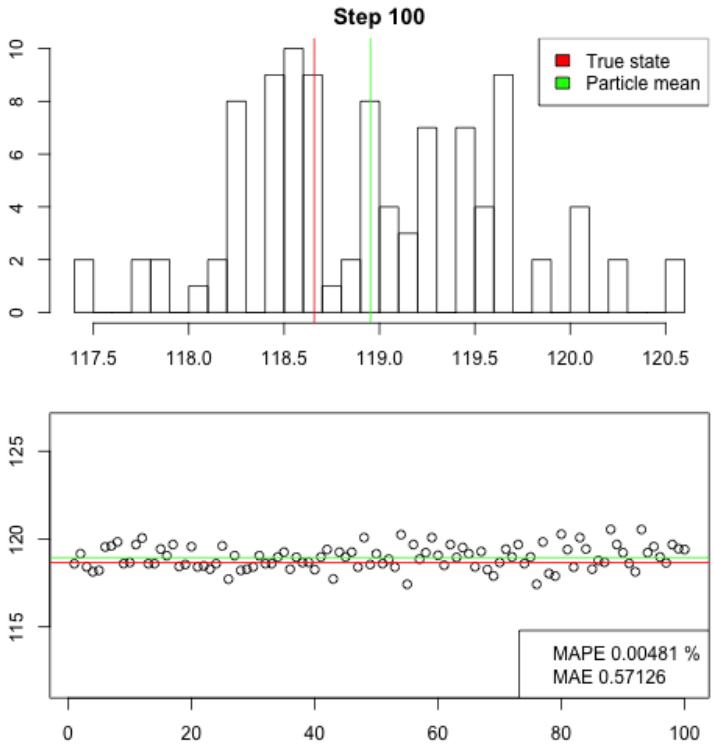


Figure 4: Particles at step 100

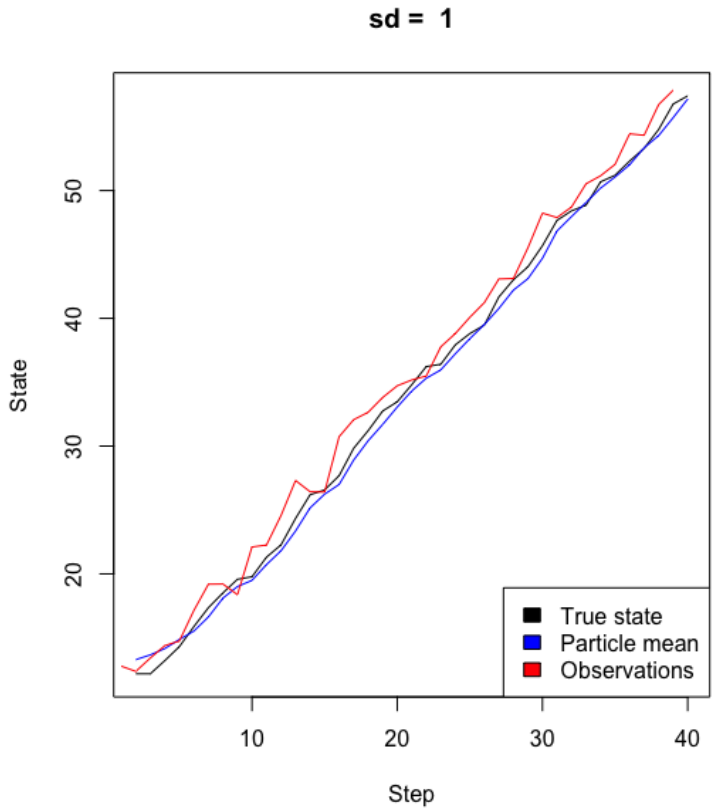


Figure 5: Traceplot comparision

```
################ 1 ################
set.seed(1234)
T = M <- 100
simulations <- simulate(T)
particles <- particleFilter(T=T, M=M, X=simulations$observations)

plotParticles(particles = particles, simulations = simulations, step=1)
plotParticles(particles = particles, simulations = simulations, step=25)
plotParticles(particles = particles, simulations = simulations, step=75)
plotParticles(particles = particles, simulations = simulations, step=100)

plotCurve(particles = particles, simulations = simulations)
```

2

I started with plotting traceplots with sd = 5 and sd = 50, as in 1), which turned out to be a bit messy when the variance of the observations got higher. In order to evaluate and compare the different models, I plotted the residual of the true state and particle mean for the different standard deviations. It's surprising how well the filter performs even for higher sigmas, although it struggles when the variance gets absurd. This shows that although the variance of the particles might be high, evaluating the mean will still generate a reasonably good prediction. Thus, more noise (higher emission variance) probably requires a larger number of particles in order to give accurate predictions.

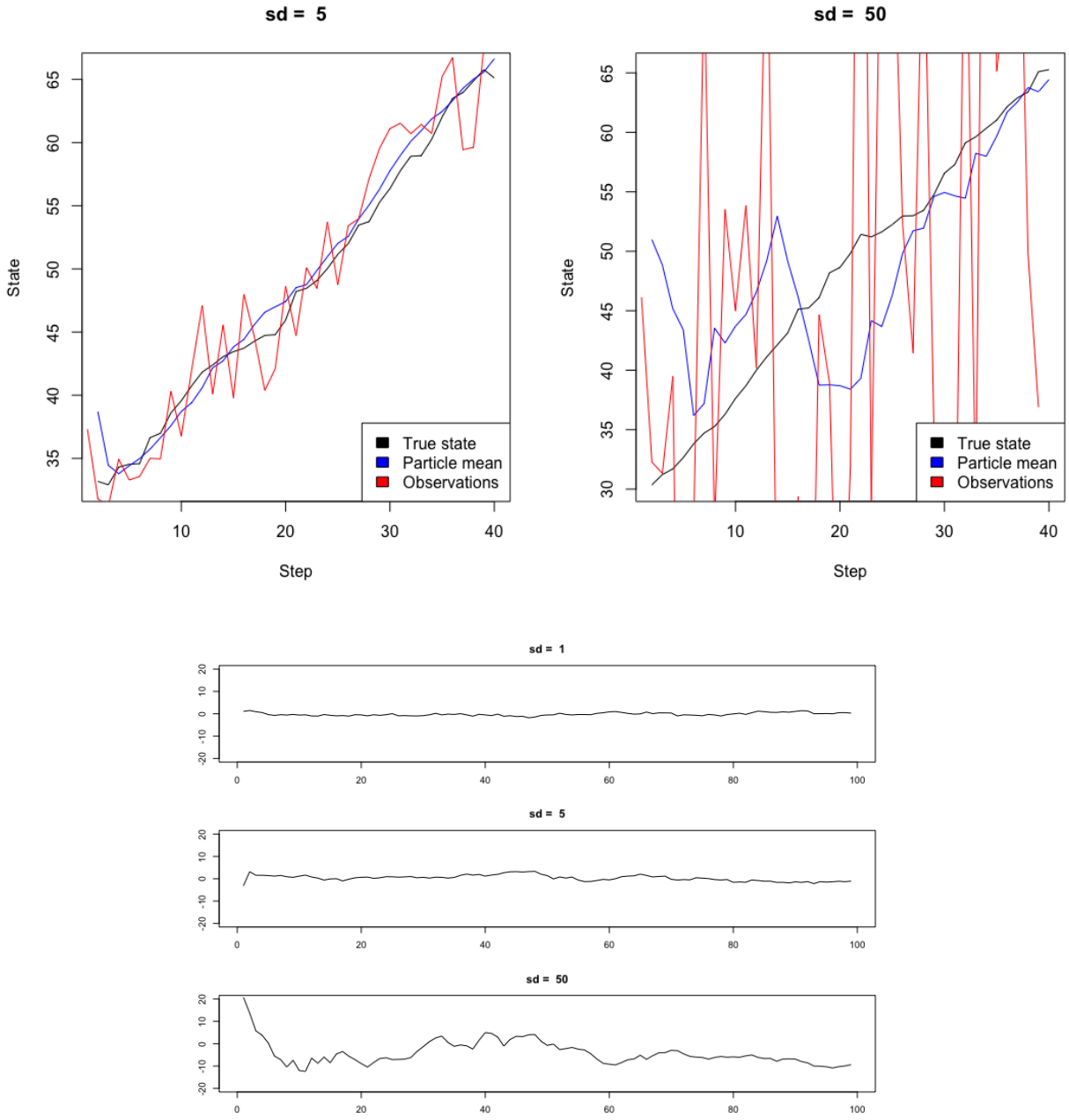
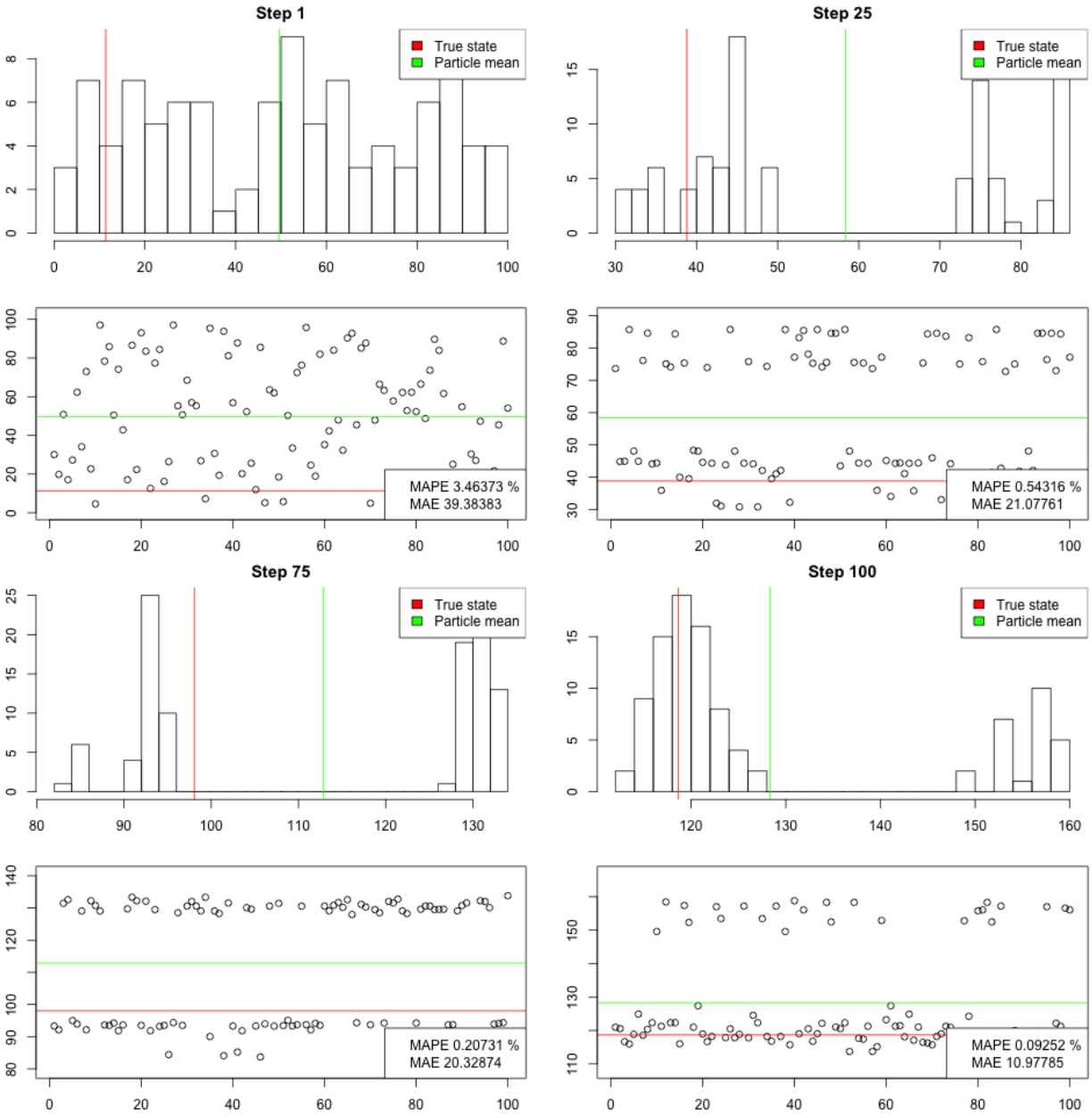


Figure 6: Particle residual for different standard deviations

```
################ 2 ################
set.seed(1234)
T = M <- 100

par(mfrow=c(3,1), mar=c(3,3,3,3))

# Standard deviation = 1
simulations <- simulate(T)
particles <- particleFilter(T=T, M=M, X=simulations$observations)
plotResidual(particles, simulations,1)

# Standard deviation = 5
simulations <- simulate(T, emissionSigma = 5)
particles <- particleFilter(T=T, M=M, X=simulations$observations, emissionSigma = 5)
plotResidual(particles, simulations, 5)

# Standard deviation = 50
simulations <- simulate(T, emissionSigma = 50)
particles <- particleFilter(T=T, M=M, X=simulations$observations, emissionSigma = 50)
plotResidual(particles, simulations, 50)
```

3

Without correction, the particles seem to converge towards two different values. I would have expected the particles to be distributed almost according to the initial uniform model, since we sample from the particles with unweighted probability.




```
################ 3 ################
set.seed(1234)
T = M <- 100
simulations <- simulate(T)
particles <- particleFilter(T=T, M=M, X=simulations$observations, constant = TRUE)

plotParticles(particles = particles, simulations = simulations, step=1)
plotParticles(particles = particles, simulations = simulations, step=25)
```

```
plotParticles(particles = particles, simulations = simulations, step=75)
plotParticles(particles = particles, simulations = simulations, step=100)
```

Lab 4 - Gaussian Processes

Arvid Edenheim - arved490

2019-08-18

Function Setup

```
####### Functions #######

# Squared exponential kernel from lecture examples
# Courtesy of Jose M. Pena, LiU
squaredExponential <- function(x1,x2,sigmaF,l){
  n1 <- length(x1)
  n2 <- length(x2)
  K <- matrix(NA,n1,n2)
  for (i in 1:n2){
    K[,i] <- sigmaF^2 * exp( -0.5 * ( (x1 - x2[i]) / l)^2 )
  }
  return(K)
}

# Prediction function from Rasmussen & Williams
# X = Training inputs
# y = Training targets/outputs
# XStar = Vector of inputs where the posterior distr. is evaluated
# hyperParam = kernel parameters sigma_f and l
# sigmaNoise = Noise sigma_n
# K = Kernel function
posteriorGP <- function(X, y, XStar, sigmaF, l, sigmaNoise, K) {

  # Need to transpose cholesky matrix to attain lower triangular matrix
  L <- t(chol(K(X, X, sigmaF, l) + sigmaNoise^2 * diag(length(X))))
  alpha <- solve(t(L), solve(L, y))

  # Predictive mean
  kStar <- K(X, XStar, sigmaF, l)
  fBar <- t(kStar) %*% alpha

  # Predictive variance
  v <- solve(L, kStar)
  vf <- K(XStar, XStar, sigmaF, l) - t(v) %*% v

  return(list(mean=fBar, variance=diag(vf)))
}


plotGP <- function(XStar, X, y, mean, sd) {
  plot(XStar,
       posterior$mean,
       type='l',
       ylim=c(min(posterior$mean) - 3,max(posterior$mean) + 3))
  points(X, y)
  quantile <- qnorm(0.975)
  lines(XStar, mean + quantile * sd, col='grey')
  lines(XStar, mean - quantile * sd, col='grey')
  legend('topleft', legend = c('Mean', 'Probability band'), fill = c('black', 'grey'))
}
```

2.1 - GP Regression implementation

GP model to implement: $y = f(x) + \epsilon$ with $\epsilon \sim N(0, \sigma_n^2)$ and $f \sim GP(0, k(x, x'))$

2.1.1

The implementation for the posterior of the GP above was implemented using the algorithm described by Rasmussen & Williams.

```
####### 2.1.1 #######
# See posteriorGP and squaredExponential above
```

2.1.2

The prior was updated with the point $(x, y) = (0.4, 0.719)$, using the the parameters $\sigma_f = 1$, $l = 0.3$ and $\sigma_n = 0.1$, after which the posterior mean was plotted with 95% probability bands. The generated plot clearly indicates a smaller variance around the observation.

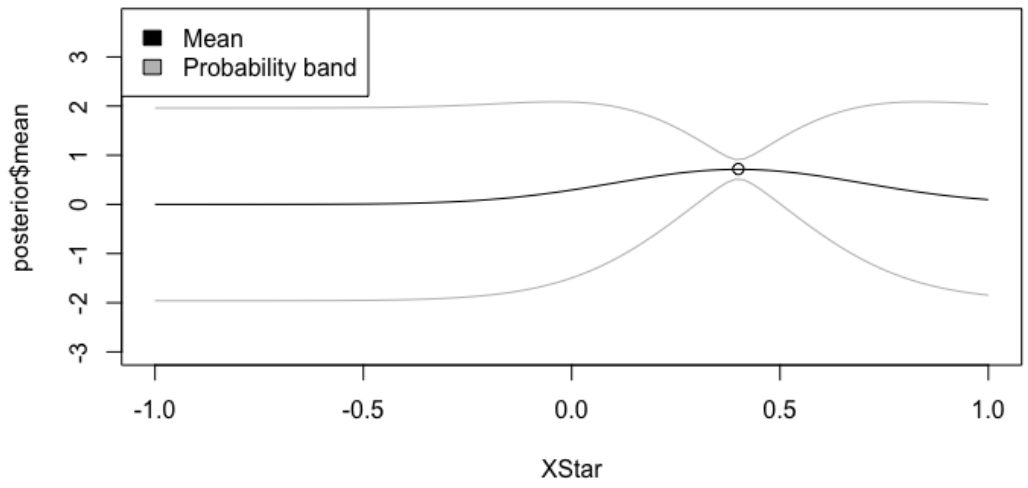


Figure 1: Posterior mean with $\sigma = 1$ and $l = 0.3$

```
####### 2.1.2 #######
sigmaF <- 1
l <- 0.3
XStar <- seq(-1, 1, length.out = 100)
y <- c(0.719)
X <- c(0.4)
sigmaN <- 0.1

posterior <- posteriorGP(X, y, XStar, sigmaF, l, sigmaN, squaredExponential)
plotGP(XStar, X, y, posterior$mean, sqrt(posterior$variance))
```

2.1.3

The posterior was updated with yet another observation and plotted. Same conclusion as 2.1.2.

JMP: Updating the posterior after one observation with a new observation gives the same result as updating the prior directly with the two observations.

But that must be less efficient, right? Not even sure how to update the posterior.

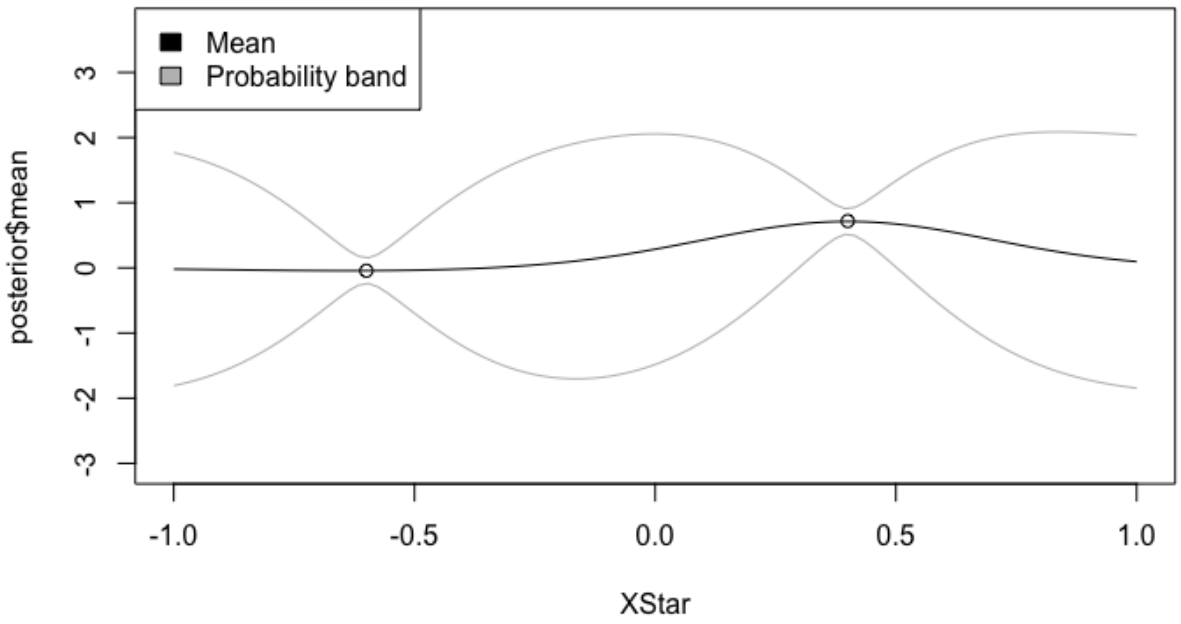


Figure 2: Posterior mean with two observations

```
####### 2.1.3 #######
y <- c(0.719, -0.044)
X <- c(0.4, -0.6)
posterior <- posteriorGP(X, y, XStar, sigmaF, l, sigmaN, squaredExponential)
plotGP(XStar, X, y, posterior$mean, sqrt(posterior$variance))
```

2.1.4

Now with five observations. The posterior starts to resemble a decent regression curve, despite the small number of obsvervations.

x	-1.0	-0.6	-0.2	0.4	0.8
y	0.768	-0.044	-0.940	0.719	-0.664

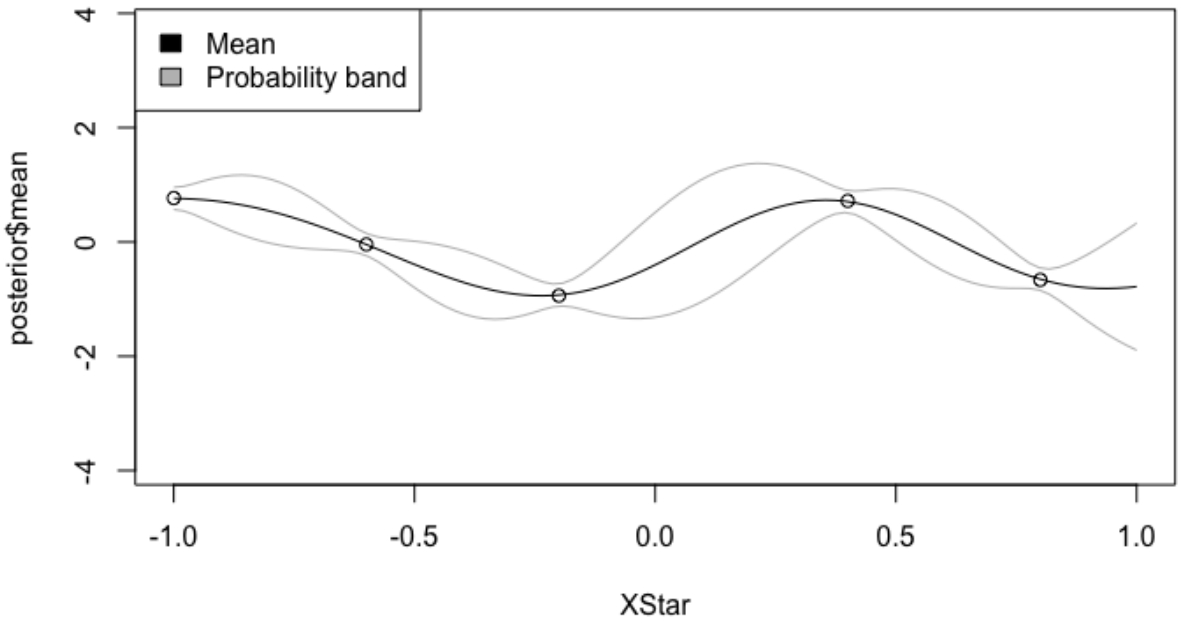


Figure 3: Posterior mean with five observations

```
####### 2.1.4 #######
y <- c(0.719, -0.044, 0.768, -0.940, -0.664)
X <- c(0.4, -0.6, -1, -0.2, 0.8)
posterior <- posteriorGP(X, y, XStar, sigmaF, l, sigmaN, squaredExponential)
plotGP(XStar, X, y, posterior$mean, sqrt(posterior$variance))
```

2.1.5

Same as 2.1.4, but with $\sigma_f = 1$ and $l = 1$.

A higher value for l gives a wider span in which the observations will affect the posterior (higher covariance), thus decreasing the overall uncertainty/variance in-between observations, making the model smoother, and possibly makes the model more prone to underfit. Looking at the plot, $l = 1$ seems to result in underfitting the data.

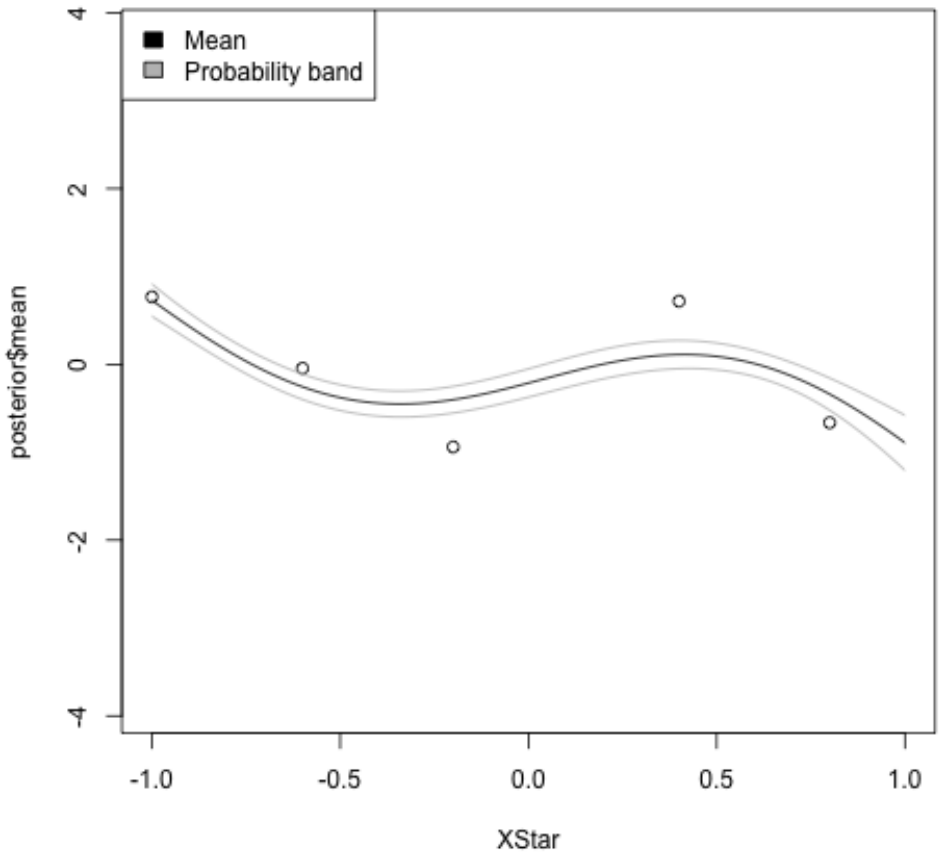


Figure 4: Posterior mean with changed kernel parameters

```
####### 2.1.5 #######
sigmaF <- 1
l <- 1
```

2.2 - GP Regression with kernlab

Setup

```
# Setup
temp <- read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/TempTullinge.
         header=TRUE,
         sep=";")
time <- 1:2190
day <- rep(1:365, 6)

# Sample (so what I just specified was pointless)
samples <- seq(1, 2190, by=5)
time <- time[samples]
day <- day[samples]
tempSamples <- temp$temp[samples]
```

2.2.1

$K(1,2) = 0.6065307$

$X = (1,3,4)^T$ & $X_* = (2,3,4)^T$

$K(X, X_*) =$

$$\begin{pmatrix} 0.6065307 & 0.1353353 & 0.0111090 \\ 0.6065307 & 1.0000000 & 0.6065307 \\ 0.1353353 & 0.6065307 & 1.0000000 \end{pmatrix}$$

```
####### 2.2.1 #######
squaredExponentialKernel <- function(ell, sigmaf) {
  squaredExponential <- function(x, y = NULL) {
    n1 <- length(x)
    n2 <- length(y)
    K <- matrix(NA,n1,n2)
    for (i in 1:n2){
      K[,i] <- sigmaf^2 * exp( -0.5 * ( (x - y[i]) / ell)^2 )
    }
    return(K)
  }
  class(squaredExponential) <- 'kernel'
  return(squaredExponential)
}

SEkernel <- squaredExponentialKernel(ell = 1, sigmaf = 1)

SEkernel(1,2)

X <- c(1,3,4)
XStar <- c(2,3,4)

K <- kernelMatrix(kernel = SEkernel,
             x = X,
             y = XStar)
```

2.2.2

Model using time as input

$temp = f(time) + \epsilon$ with $\epsilon \sim N(0, \sigma_n^2)$ and $f \sim GP(0, k(time, time'))$

Fitted a second degree polynomial due to the characteristics of the data. The generated GP seems to fit the data fairly well.

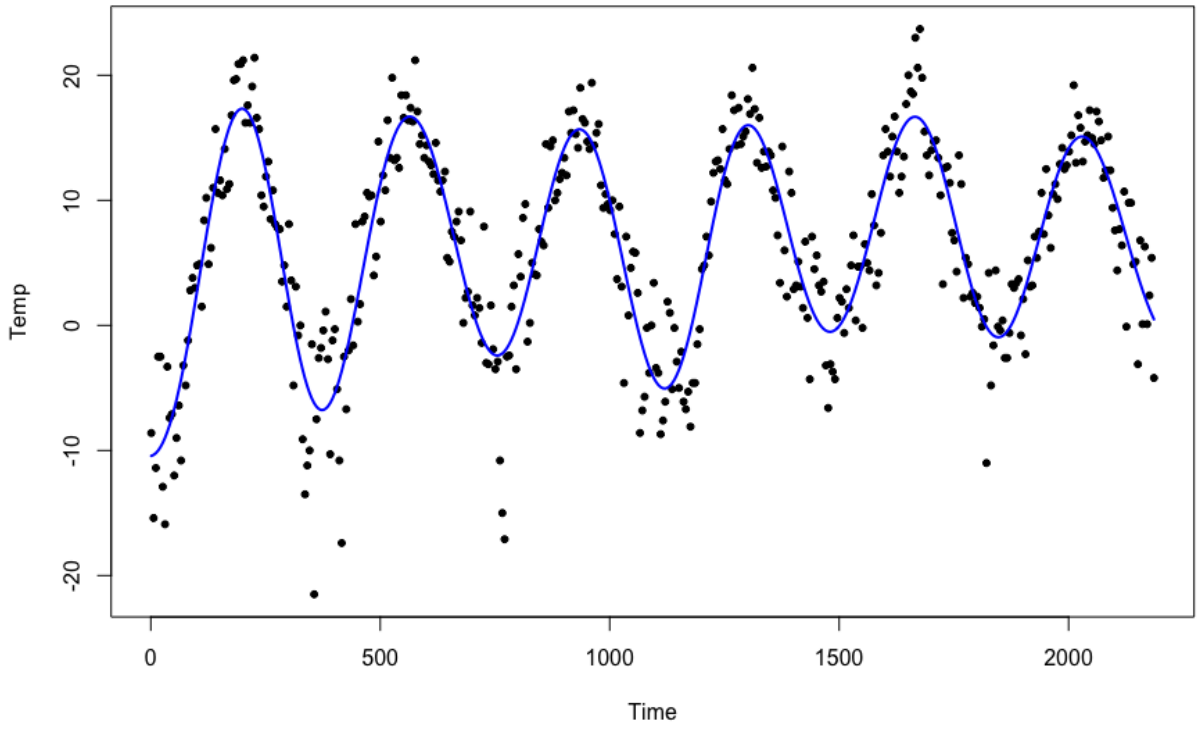


Figure 5: GP regression with time as input

```
####### 2.2.2 #######
plot(temp$temp)

# Fitting a second degree polynomial due to characteristics of the data
fit <- lm(formula=tempSamples ~ time + I(time^2))
sigmaNoise <- sd(fit$residuals)

sigmaF <- 20
l <- 0.2
SEkernel <- squaredExponentialKernel(ell = l, sigmaf = sigmaF)

GPfit <- gausspr(time, tempSamples, kernel = SEkernel, var = sigmaNoise^2)
meanPred <- predict(GPfit, time) # Predicting the training data. To plot the fit.
```

```r
plot(time, tempSamples, pch=20, ylab='Temp', xlab='Time')
lines(time, meanPred, col="blue", lwd = 2)

# As before, increasing l results in underfitting
# Setting a small sigmaF also results in underfitting, otherwise minor effect
```

2.2.3

I consider the fitted model to be quite good, however underfitting somewhat when considering the extremes.

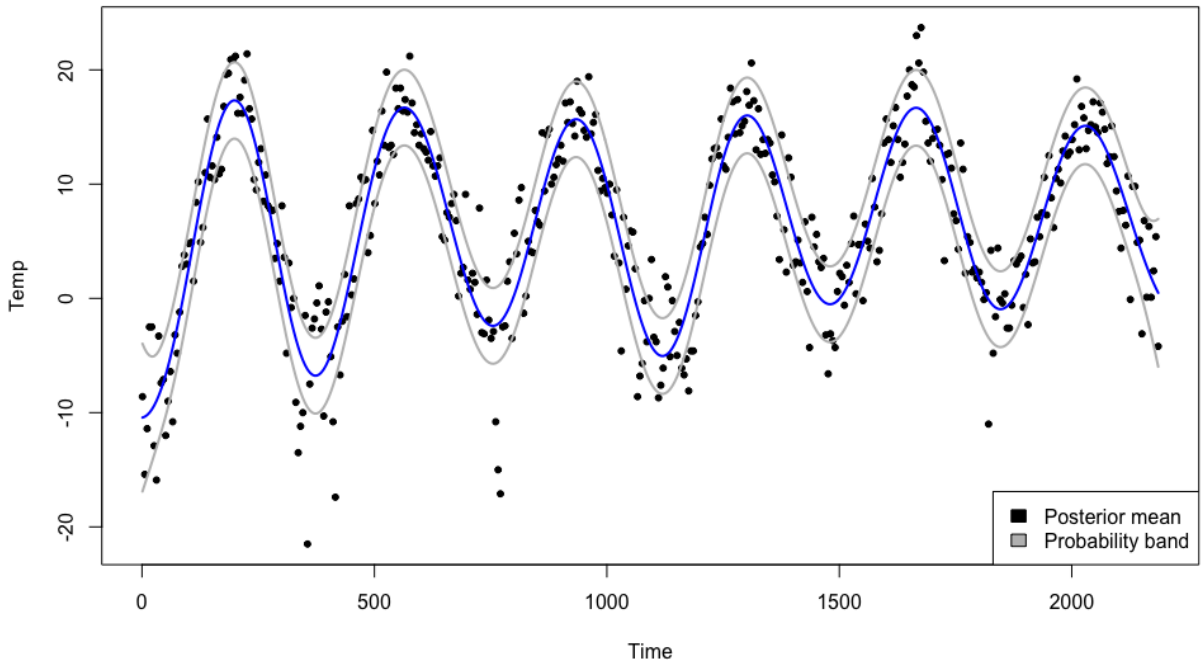


Figure 6: GP with 95% probability bands

```r
####### 2.2.3 #######
posterior <- posteriorGP(X = scale(time),
                         y = scale(tempSamples),
                         XStar = scale(time),
                         sigmaF = sigmaF,
                         l = l,
                         sigmaNoise = sigmaNoise,
                         K = squaredExponential)

quantile <- qnorm(0.025)
lines(time, meanPred - quantile * sqrt(posterior$variance),
      col = "grey", lwd = 2)
lines(time, meanPred + quantile * sqrt(posterior$variance),
```

```
      col = "grey", lwd = 2)
legend('bottomright',
       legend = c('Posterior mean', 'Probability band'),
       fill = c('black', 'grey'))
```

2.2.4

Model using day as input

$temp = f(day) + \epsilon$ with $\epsilon \sim N(0, \sigma_n^2)$ and $f \sim GP(0, k(day, day'))$

This model also generates a reasonably good model, although not as smooth as in 2.2.2. One possible reason for using this instead would be that it appears to be slightly better at capturing the extremes.

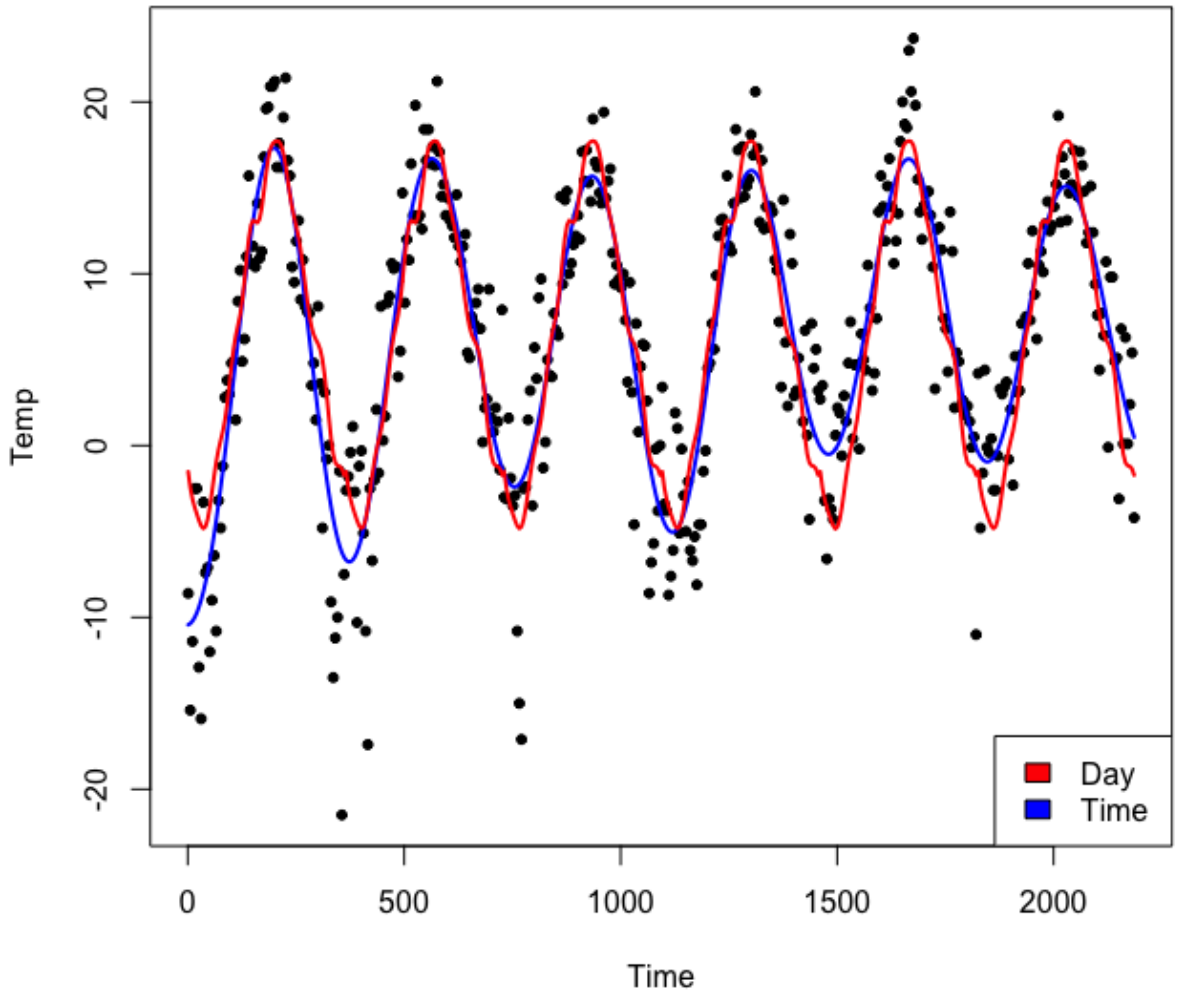


Figure 7: GP regression with day as input

```
####### 2.2.4 #######
fit <- lm(formula=tempSamples ~ day + I(day^2))
sigmaNoise <- sd(fit$residuals)

GPfit <- gausspr(day, tempSamples, kernel = SEkernel, var = sigmaNoise^2)
meanPredDay <- predict(GPfit, day) # Predicting the training data. To plot the fit.

plot(time, tempSamples, pch=20, ylab='Temp', xlab='Time')
lines(time, meanPred, col="blue", lwd = 2)
lines(time, meanPredDay, col="red", lwd = 2)
legend('bottomright', legend = c('Day', 'Time'), fill = c('red', 'blue'))
```

2.2.5

The GP with a periodic kernel also appeared to be a good fit. I would prefer this model over the others since it allows more configurability, although increasing the need to select reasonable hyperparameters in order to avoid under/overfitting.

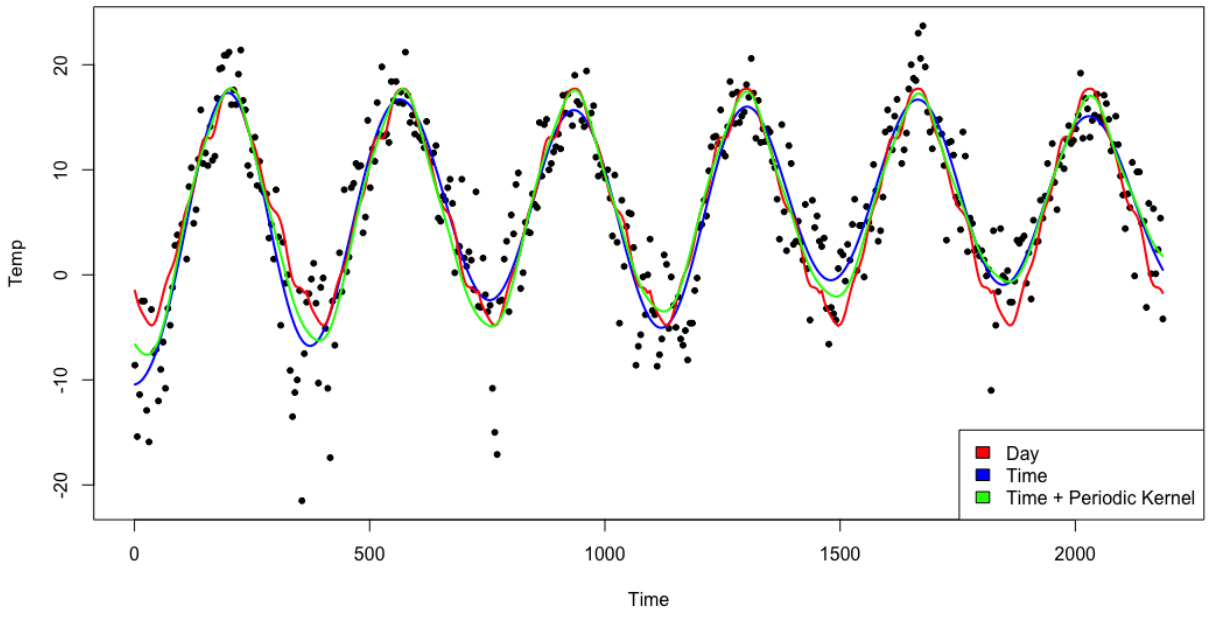


Figure 8: GP with periodic kernel

```
####### 2.2.5 #######

sigmaF <- 20
l1 <- 1
l2 <- 10
d = 365 / sd(time) # Using the sampled interval, nearly the same as sd(1:2190)

PKernel <- function(l1, l2, d, sigmaF) {
  periodicKernel <- function(x, y = NULL) {
    K <- sigmaF^2*exp(-2*sin(pi*abs(x-y)/d)^2/l1^2)*exp(-0.5*abs(x-y)^2/l2^2)
    return(K)
  }
  class(periodicKernel) <- 'kernel'
  return(periodicKernel)
}

pk <- PKernel(l1, l2, d, sigmaF)

fit <- lm(formula=tempSamples ~ time + I(time^2))
sigmaNoise <- sd(fit$residuals)
```

```
GPfit <- gausspr(time, tempSamples, kernel = pk, var = sigmaNoise^2)
meanPredPeriodic <- predict(GPfit, time)

plot(time, tempSamples, pch=20, ylab='Temp', xlab='Time')
lines(time, meanPred, col="blue", lwd = 2)
lines(time, meanPredDay, col="red", lwd = 2)
lines(time, meanPredPeriodic, col="green", lwd = 2)
legend('bottomright',
       legend = c('Day', 'Time', 'Time + Periodic Kernel'),
       fill = c('red', 'blue', 'green'))
```

2.3 - Classification with kernlab

Setup

```
# Setup
accuracy <- function(confMatrix) {
  return(sum(diag(confMatrix))/sum(confMatrix))
}

data <- read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/banknoteFraud
                 header=FALSE,
                 sep=",")

names(data) <- c("varWave","skewWave","kurtWave","entropyWave","fraud")
data[,5] <- as.factor(data[,5])

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

2.3.1 - Using varWave and skewWave to classify

Accuracy = 0.932

Confusion Matrix

	False	True
False	512	24
True	44	420

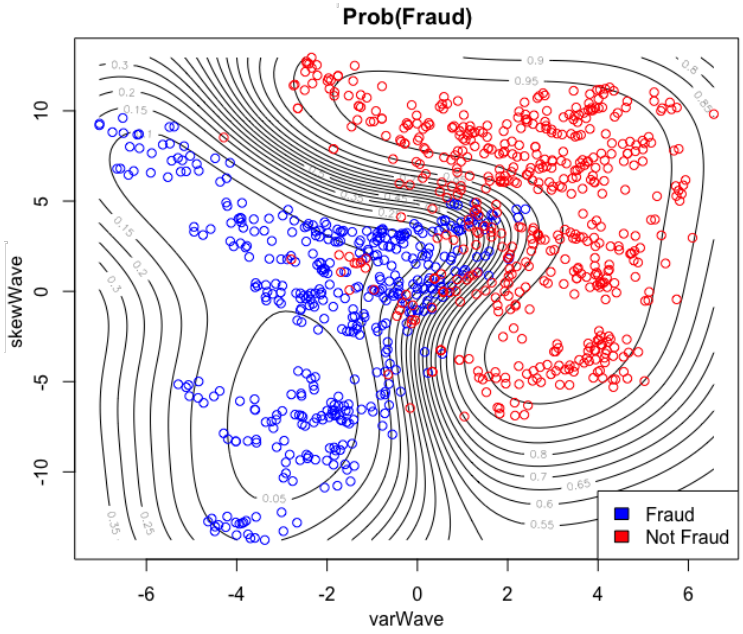


Figure 9: Contour plot of fraud data

```
# Using varWave and skewWave to classify
GPfit <- gausspr(fraud ~  varWave + skewWave, data=train)

# Predict on the training set
GPpred <- predict(GPfit, train[, c(T, T, F, F, F)]) # Doesn't seem to be necessary
confM <- table(GPpred, train[,5]) # confusion matrix
accuracy(confM)

# Class probabilities
probPreds <- predict(GPfit, train[, c(T, T, F, F, F)], type="probabilities")
x1 = seq(min(train$varWave), max(train$varWave), length=100)
x2 = seq(min(train$skewWave), max(train$skewWave), length=100)
gridPoints <- meshgrid(x1, x2)
gridPoints <- cbind(c(gridPoints$x), c(gridPoints$y))
gridPoints <- data.frame(gridPoints)
names(gridPoints) <- names(train)[c(1,2)]
probPreds <- predict(GPfit, gridPoints, type="probabilities")

# Contour plot
contour(x1,
        x2,
        matrix(probPreds[,1], 100, byrow = TRUE),
        20,
        xlab = "varWave",
        ylab = "skewWave",
        main = 'Prob(Fraud)')
points(train$varWave[train$fraud == 1],
       train$skewWave[train$fraud == 1],
       col = 'blue')
points(train$varWave[train$fraud == 0],
       train$skewWave[train$fraud == 0],
       col = 'red')

legend('bottomright', legend = c('Fraud', 'Not Fraud'), fill = c('Blue', 'Red'))
```

2.3.2

Accuracy = 0.9354839, which is pretty much the same as the accuracy on the training data. Not great, not terrible.

Confusion Matrix

	False	True
False	191	9
True	15	157

```
####### 2.3.2 #######
# Predict on the test data
GPtest <- predict(GPfit, test[, c(T, T, F, F, F)]) # Doesn't seem to be necessary
confusionMatrix <- table(GPtest, test[,5])
accuracy(confusionMatrix)
```

2.3.3

Accuracy on test data = 0.9973118. Great! Just one wrongly predicted out of 372 samples. Overfitting not likely since test data.

Confusion Matrix

	False	True
False	205	0
True	1	166

```
####### 2.3.3 #######
# Use all available covariates
GPfit <- gausspr(fraud ~  varWave + skewWave + kurtWave + entropyWave, data=train)

# Predict on test data
GPtest <- predict(GPfit, test[, c(T, T, T, T, F)]) # Doesn't seem to be necessary
confusionMatrix <- table(GPtest, test[,5])
accuracy(confusionMatrix)
# WOW! High!
```