AdvML Lab4 Group Report

Andreas Stasinakis(andst745) & Mim Kemal Tekin(mimte666) & Stefano Toffol(steto820) & Bruno Barakat(bruba569)

16 ottobre, 2019

Contents

Question 1

GP regression

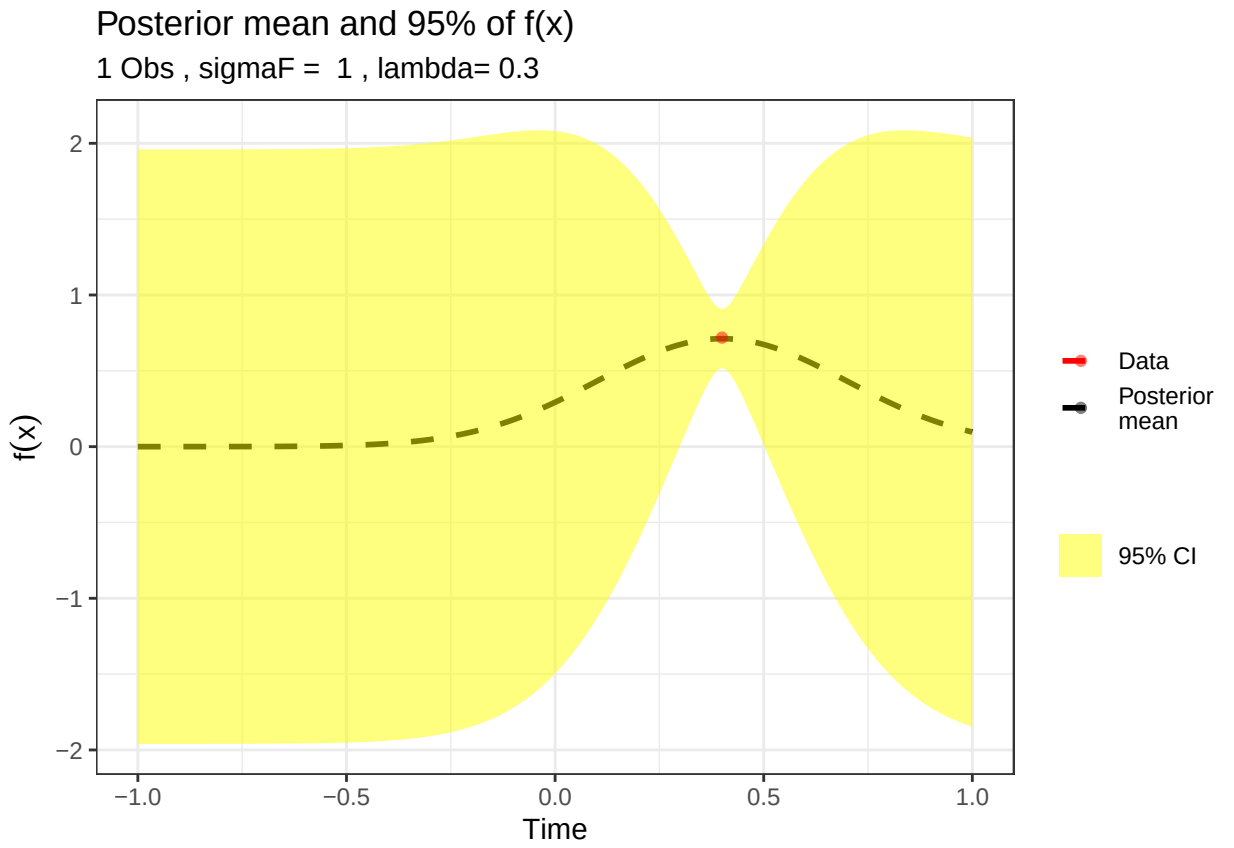
```
#All the function we will use for the 1st question
# Covariance function
SquaredExpKernel <- function(x1,#vector of traing data
                             x2,#vector of training data
                             par)#vector of 2 parameters:SigmaF and Lambda(l)
  {
  sigmaF = par[1]
  l = par[2]
  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)
}


#function for Cholesky decompotion
#The formula for the covariance of the joint Gaussian is not accurate and stable
#If we have many dimensions. Therefore we will use Cholesky decompotion in order
#to make more stable.

Decompotion = function(X,#Training data
                       y,#target training data
                       Kernel_function,#function for compute kernel covariance
                       sigmaNoise,#sd of the noise of the data
                       X_star, #data where the post is evaluated
                       hyperPar)#vector of 2 param: SigmaF and lambda
  {
  n = length(X)
  Kxx = Kernel_function(X,X,hyperPar)
  Kxs = Kernel_function(X,X_star,hyperPar)
  Ksx = Kernel_function(X_star,X,hyperPar)
  Kss = Kernel_function(X_star,X_star,hyperPar)

  #first we do the cholesky decompotion
  L = t(chol(Kxx + (sigmaNoise^2)*diag(n)))


  alpha  = solve(t(L),solve(L,y))

  #Compute the posterior mean
  meanPost = t(Kxs)%*%alpha

  #compute the variance of the posterior
  v = solve(L,Kxs)

  VarPost = Kss - t(v)%*%v
```

```
  #finally compute the log marginal likelihood for model comparison
  #logLike = (-1/2)*(t(y)*alpha) - sum(log(diag(L))) -(n/2)*log(2*pi)

  return(list(meanPost,VarPost))

}

# Mean function. In that example we have the mean vector 0
MeanFunc = function(x){
  m = rep(0,length(x))
  return(m)
}



#Function to simulate from the posterior
posteriorGP = function(X,#Vector of input training data,
                       y,#Vector of training targets
                       Xstar,#Vector of inputs where the post is evaluated
                       hyperParam,#Vector with two elements sigmaF and lambda
                       Kernel_function, #Kernel function
                       sigmaNoise) #Noise's standard deviation

  {
  n = length(Xstar)
  sigmaF = hyperParam[1]
  lambda = hyperParam[2]

  #We send our input to the Decompotion function
  postMean = Decompotion(X = X,y = y,Kernel_function = Kernel_function,
                         sigmaNoise = sigmaNoise,X_star = Xstar,
                         hyperPar = hyperParam)[[1]]

  varPost = Decompotion(X = X,y = y,Kernel_function = Kernel_function,
                        sigmaNoise = sigmaNoise,X_star = Xstar,
                        hyperParam)[[2]]


  # logMarginal = Decompotion(X = X,y = y,Kernel_function = SquaredExpKernel,
  #                                     sigmaNoise = sigmaNoise,X_star = Xstar,
  #                                     hyperParam)[[3]]
  #simulate from the posterior
  #res = rmvnorm(n, mean = postMean, sigma = varPost)
  return(list(postMean,varPost))


  }


plot_FUN = function(df,X,y,allColors,pars,...){
  p = ggplot()+
    geom_line(mapping = aes(x = df[,1],y = df[,2], col = "Posterior\nmean"),
              lty = 2, size = 1)+
```

```
    geom_ribbon(aes(x = df[,1],ymin=df[,3],
                    ymax=df[,4],fill = "95% CI"),linetype=2, alpha=0.5)+
    geom_point(mapping = aes(x = X, y = y, col = "Data"), alpha = 0.5)+
    labs(title = paste("Posterior mean and 95% of",expression(f(x))),
         subtitle = paste(...,"Obs",", sigmaF = ",
                          pars[1],", lambda=",pars[2]),
         x = "Time", y = expression(f(x)))+
    scale_color_manual(values = c("Posterior\nmean" = allColors[1],
                                  "Data" = allColors[2]), name = "")+
    scale_fill_manual(values = c("95% CI" = allColors[3]),name = "" )
    #theme(legend.key = element_rect(fill = "lightblue", color   = "blue"))

  return(p)

}
```

1.2 Posterior mean with one observation(Scaterplot with 95% CI)

```
#User's input
X = 0.4 #here we only have only observation
y = 0.719
Xstar = seq(-1,1,0.01) #We evaluate the function in a grid vector of Xvalues
hyperPar = c(1,0.3) # Sigmaf and lambda
sigmaN = 0.1 #the sd of the model
allColors = c("black","red", "yellow") #colors for the plots

#First run the alogirhm
post1 = posteriorGP(X = X,y = y,Xstar = Xstar,hyperParam = hyperPar,
                    Kernel_function = SquaredExpKernel,
                    sigmaNoise = sigmaN)


postMean1 = post1[[1]] #store the posterior mean
varPost1= post1[[2]] #store the variances

low_CI1 = postMean1 -1.96*sqrt(diag(varPost1))
upper_CI1 = postMean1 +1.96*sqrt(diag(varPost1))

dfPost1 = data.frame(Xstar,postMean1,low_CI1,upper_CI1)
colnames(dfPost1) = c("Xgrid","PostMean", "Lower CI",  "Upper CI")

plt1 = plot_FUN(dfPost1,X,y,allColors = allColors,hyperPar,1)
plt1
```

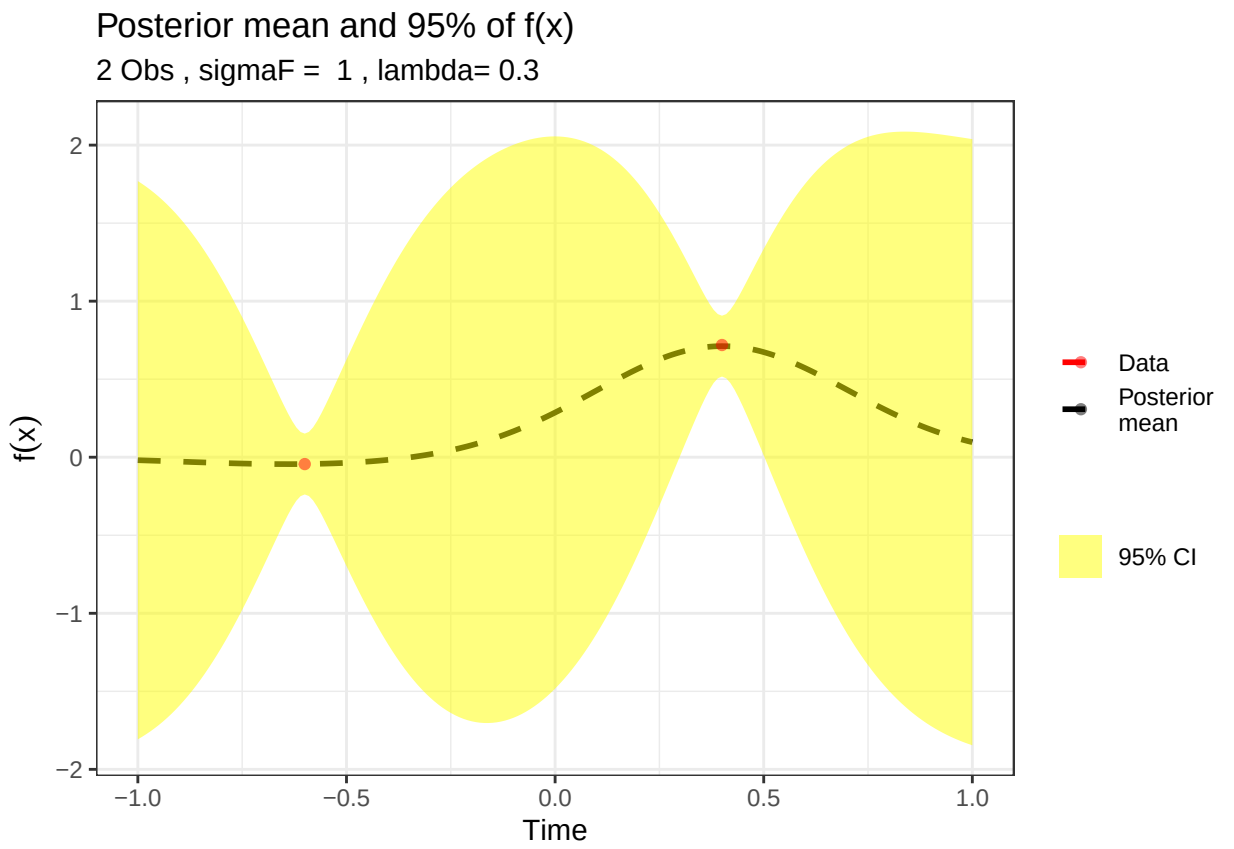


In this Question, we want to implement a GP regression using our own functions. Our target is to calculate the posterior mean and the posterior covariance matrix. After that we can make predictions or generate from the posterior distribution. We first update our prior knowledge using only one observations. Of course, we expected that the output wil be really wide. We need more observations in order to have a more clear picture of the posterior.

1.3 Update the Posterior(scaterplot and 95% CI)

```
X = c(0.4,-0.6)
y = c(0.719,-0.044)

#Second run the alogirhm
post2 = posteriorGP(X = X,y = y,Xstar = Xstar,hyperParam = hyperPar,
                    Kernel_function = SquaredExpKernel,
                    sigmaNoise = sigmaN)

postMean2 = post2[[1]]
varPost2= post2[[2]]

low_CI2 = postMean2 -1.96*sqrt(diag(varPost2))
upper_CI2 = postMean2 +1.96*sqrt(diag(varPost2))



dfPost2 = data.frame(Xstar,postMean2,low_CI2,upper_CI2)
colnames(dfPost2) = c("Xgrid","PostMean", "Lower CI",  "Upper CI")
```

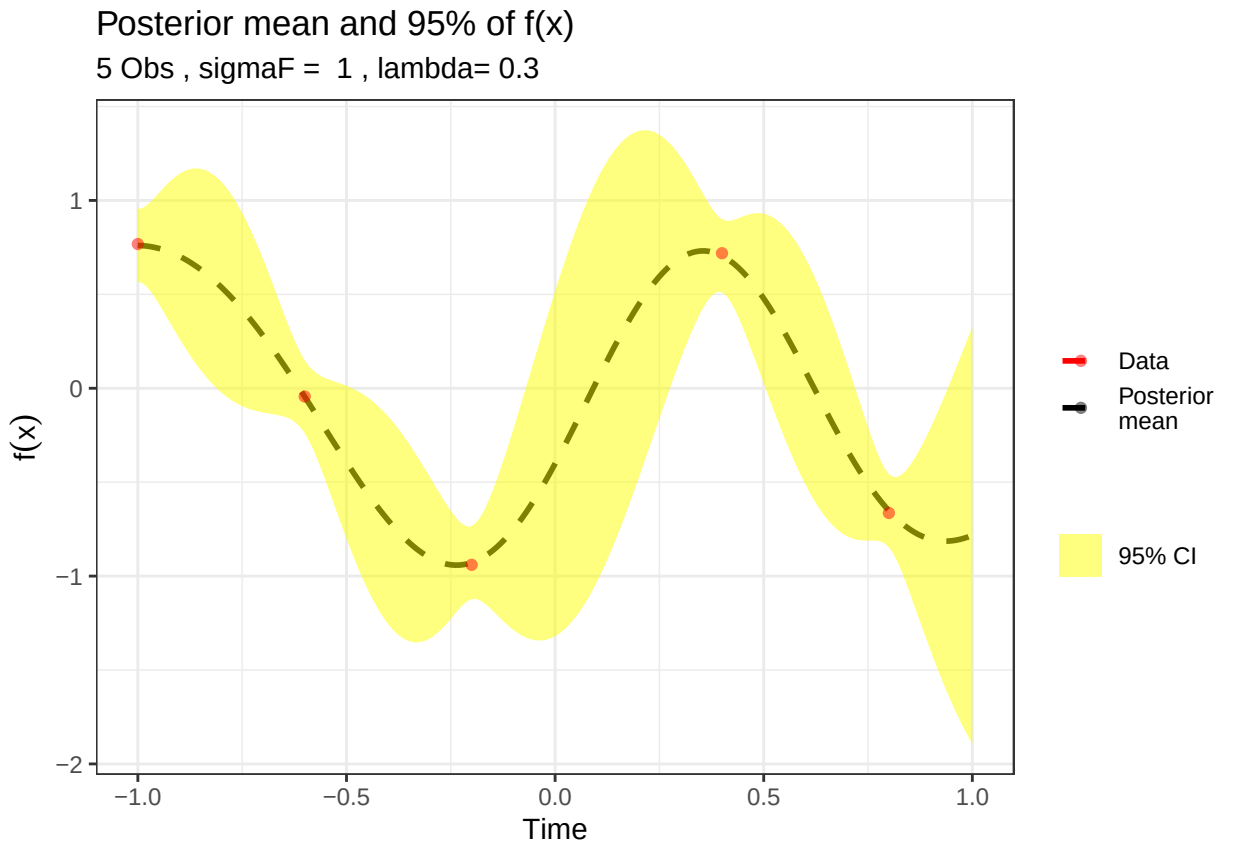
```
plt2 = plot_FUN(dfPost2,X,y,allColors,hyperPar,length(X))
plt2
```



We use one more observation of your data, 2 in total. The output is more precise than with only one observation, the boundaries start to became more tight, but we need still more observations.

1.4 Update the Posterior using 5 observations(scaterplot and 95% CI)

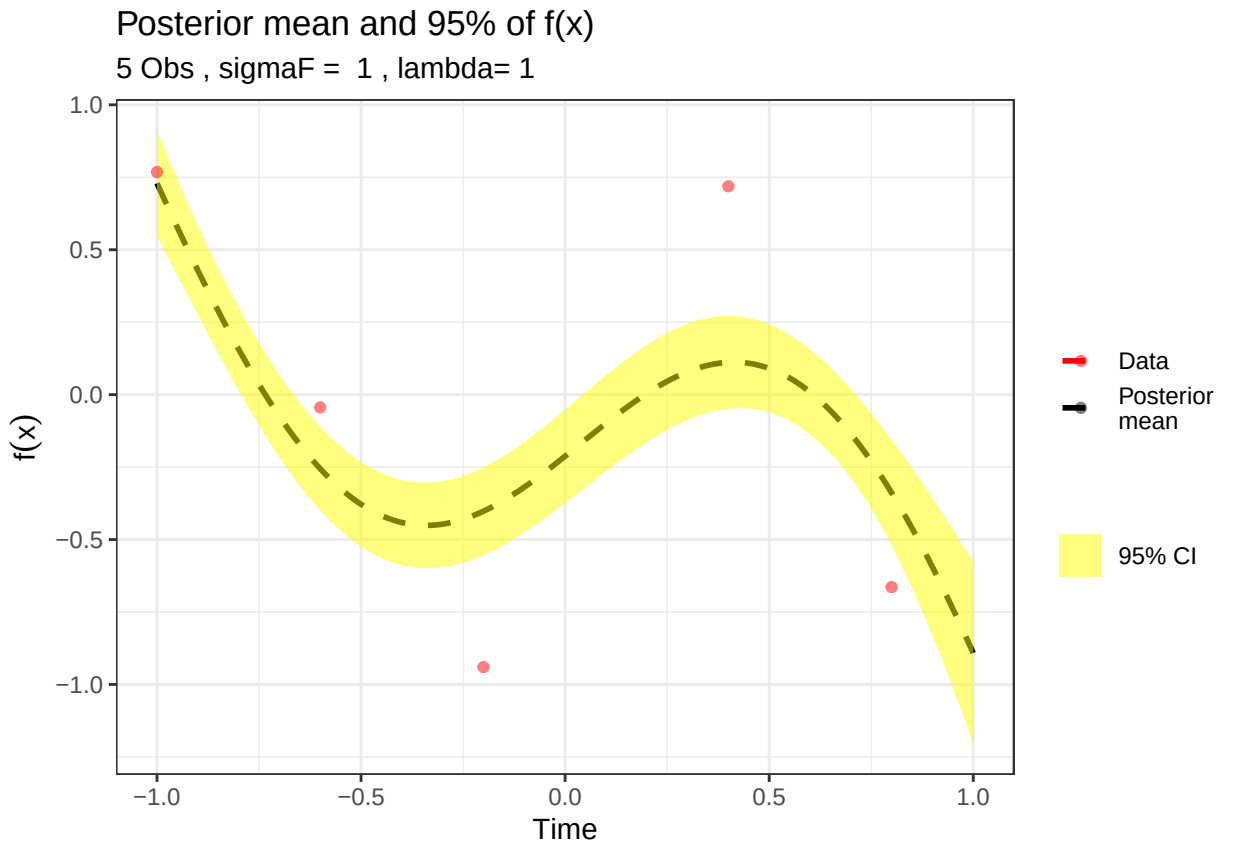
```
#Third run
X = c(-1,-0.6,-0.2,0.4,0.8)
y = c(0.768,-0.044,-0.94,0.719,-0.664)

post3 = posteriorGP(X = X,y = y,Xstar = Xstar,hyperParam = hyperPar,
                    Kernel_function = SquaredExpKernel,
                    sigmaNoise = sigmaN)

postMean3 = post3[[1]]
varPost3 = post3[[2]]

low_CI3 = postMean3 -1.96*sqrt(diag(varPost3))
upper_CI3 = postMean3 +1.96*sqrt(diag(varPost3))

dfPost3 = data.frame(Xstar,postMean3,low_CI3,upper_CI3)
colnames(dfPost3) = c("Xgrid","PostMean", "Lower CI",  "Upper CI")

plt3 = plot_FUN(dfPost3,X,y,allColors = allColors,pars = hyperPar,length(X))
plt3
```



Using 5 observations, we get a smooth function which starts to have a specific shape. Moreover, we can now see a patern of the function, and we can use that posterior means and variances in order to generate from that posterior and make predictions. Of course, the more observations, the more accurate our posterior will be. But with just 5 observations, we have a pretty decent output.

1.5 Different Hyperparameter and final comparison

```
#Change the Hyperparameters
hyperPar = c(1,1)

post3b = posteriorGP(X = X,y = y,Xstar = Xstar,hyperParam = hyperPar,
                     Kernel_function = SquaredExpKernel,
                     sigmaNoise = sigmaN)

postMean3b = post3b[[1]]
varPost3b = post3b[[2]]

low_CI3b = postMean3b -1.96*sqrt(diag(varPost3b))
upper_CI3b = postMean3b +1.96*sqrt(diag(varPost3b))

dfPost3b = data.frame(Xstar,postMean3b,low_CI3b,upper_CI3b)
colnames(dfPost3b) = c("Xgrid","PostMean", "Lower CI",  "Upper CI")

plt3b = plot_FUN(dfPost3b,X,y,allColors,hyperPar,length(X))
plt3b
```



Finally, we run the same procedure with different hyperparameters in order to make comments in the role of each hyperameter. The first one, σ_f, controls the variance of the function. That means that the higher σ_f, the wider the credible intervals will be. Moreover, the parameter l controls the smoothness of the function. More specific, values close to 1, "reduce" the smoothness of the entire function, while values close to 0, are increasing that smoothness. The best choice of l is problem dependace. That means, that there is no any rule about better or worse values of l.

For that specific problem now, we can see that when we increase l, the function can not capture the trend of the data. Therefore, for that specific problem, we need a more smooth function. For that reason, a value of l close to 0.3 should be choosen.

Question 2: Regression using package kernlab

In this question we are asked to train again different GP model, but this time fitting them into a real dataset. Moreover, we are asked to use functions from the `R` package `kernlab` and experiment with different kinds of kernels as well.

The dataset contains he daily mean temperature in Stockholm (Tullinge) during the period January 1, 2010 - December 31, 2015. We will add two variables, `time` and `day` (number of days from the first record - number of days from the 1^{st} of January of the same year). We will moreover subset the dataset, taking one observation every 5th.

1 - Kernel experiment

In this section we will process the data as requested, implement a square exponential kernel function and evaluate it in the point $x = 1, x' = 2$. The matrix output is reported at the end of the code chunk.

```
# Read the new data and prepare necessary variables
data <- read.csv(
  "https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/TempTullinge.csv",
  header=TRUE, sep=";")
n <- nrow(data)
data$time = 1:n
data$day = rep(1:365, floor(n/365)+1)[1:n]
sub_index = seq(1,2186, 5)
data = data[sub_index, c(2,3,4)]


# Redefine kernel function to have two inputs in the parameters
# We need to specify nested functions with only hyperparameters because of
# how kernelMatrix(.) works
gkernel <- function(sigmaF, ell){
  kernel <- function(x, y){
    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(kernel) <- "kernel"
  return(kernel)
}

# Evaluate kernel at point (1,2)
SEkern <- gkernel(sigmaF = 1, ell = 0.3)
SEkern(x = 1, y = 2)
```

```
##             [,1]
## [1,] 0.00386592
```

```r
# Compute cov using kernelMatrix and our kernel SEkern
X <- matrix(c(1,3,4))
XStar <- matrix(c(2,3,4))
K_matrix <- kernelMatrix(kernel = SEkern, X, XStar)
```

0.0038659	0.0000000	0.0000000
0.0038659	1.0000000	0.0038659
0.0000000	0.0038659	1.0000000

2 - GP with squared exponential function on time

In this section we will implement the following GP model:

$$temp = f(time) + \epsilon \begin{cases} \epsilon \sim \mathcal{N}(0, \sigma_n^2) \\ f \sim \mathcal{GP}(0, k(time, time')) \end{cases}$$

In order to have a realistic estimate of the variance of the noise, σ_n^2, we will run a simple quadratic regression and use the residual variance derived. The other two hyperparameter are instead set to $\sigma_f = 20$ and $\ell = 0.2$. We will then plot the result of the model and the observations used to estimate it.

```r
# Linear regression temperature vs time
fitted <- lm(temp ~ time + I(time^2), data = data)
# Extract the residual variance
sigma=sd(residuals(fitted))

# Compute the actual regression with the help of the package
gpfit <- gausspr(x=data$time, y=data$temp,
                 kernel=gkernel(sigmaF = 20, ell = 0.2),
                 kpar=list(sigmaF=20, ell=0.2),
                 var=sigma^2)

meanPredTime <- predict(gpfit, data$time)
```

As we can observe from the plot, the estimate is quite accurate, managing to capture the seasonal trend of the data.

3 - Compute the posterior variance

With the package used is not (currently) possible to compute the posterior variance of the model. We therefore used our previously implemented version on the Algorithm 2.1 on page 19 of Rasmussen and Willams' book. Side note: tha scaling of the data is necessary to adjust the output of the algorithm to what the library does by default.

```r
# Extract the variance of the model
var = Decompotion(X = scale(data$time), y = data$temp,
                  Kernel_function = SquaredExpKernel,
                  sigmaNoise = sigma, X_star = scale(data$time),
```

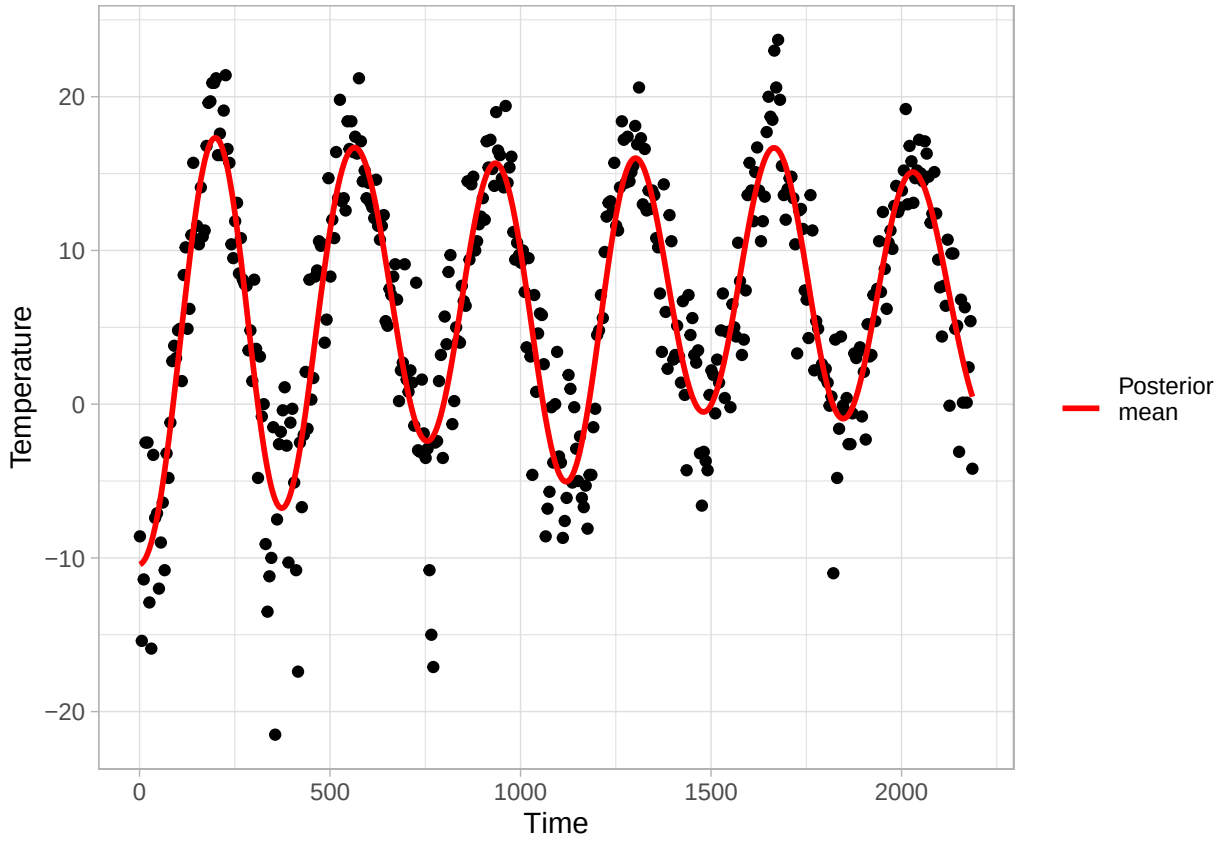


Figure 1: Fit of the GP process over time (without the probability band) usign the squared exponential kernel.

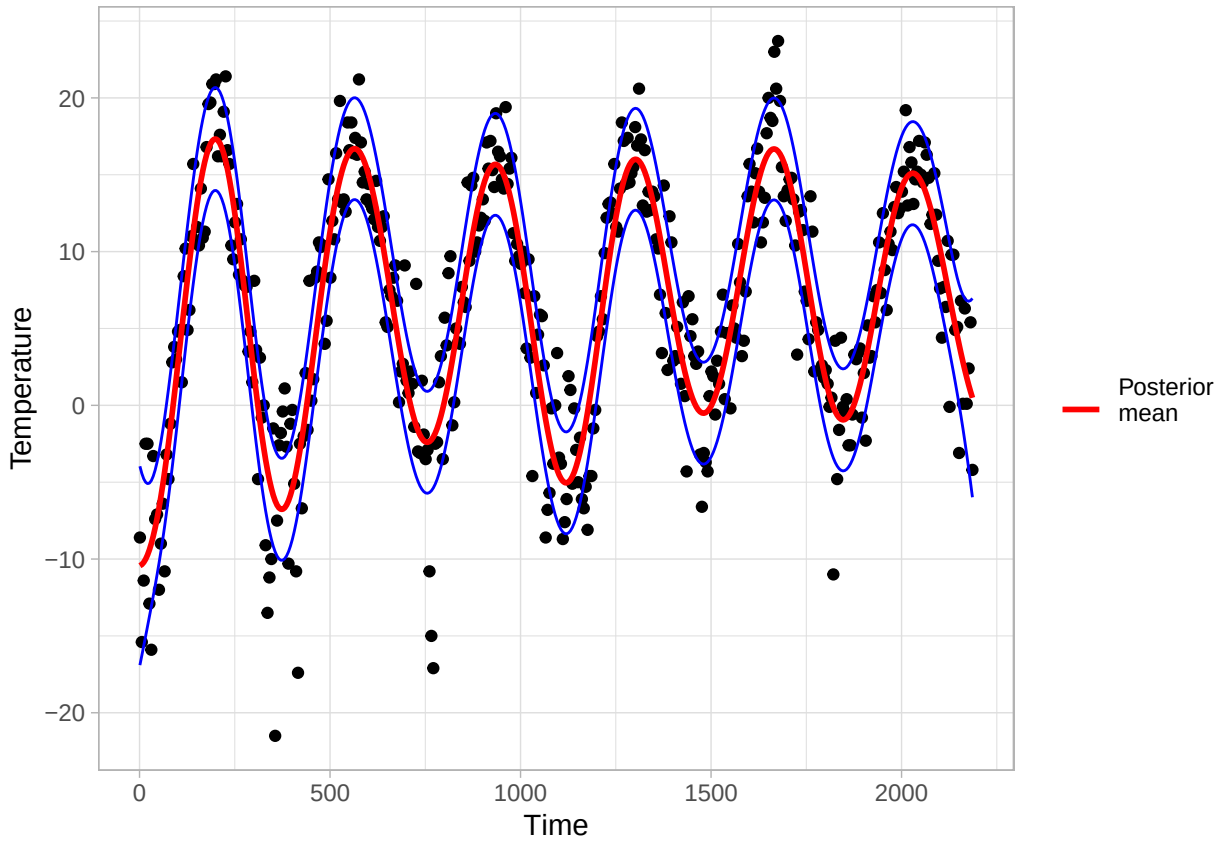


Figure 2: Fit of the GP process over time (*with* the probability band) usign the squared exponential kernel.

```
                  hyperPar = c(20, 0.2))[[2]]

# Compute the probability bands
B1 <- meanPredTime - 1.96*sqrt(diag(var))
B2 <- meanPredTime + 1.96*sqrt(diag(var))

p0 +
  geom_line(aes(x=data$time, y=B1), color = col[2]) +
  geom_line(aes(x=data$time, y=B2), color = col[2])
```

The probability band is really thin in the increasing/decresing sections of the posterior mean, however it gets quite wide in the points where the mean changes direction, corresponding to the end of summer or winter. In the changing sections the model is much more unsure and fails to clearly identify the position of the real mean.

4 - GP with squared exponential function on day

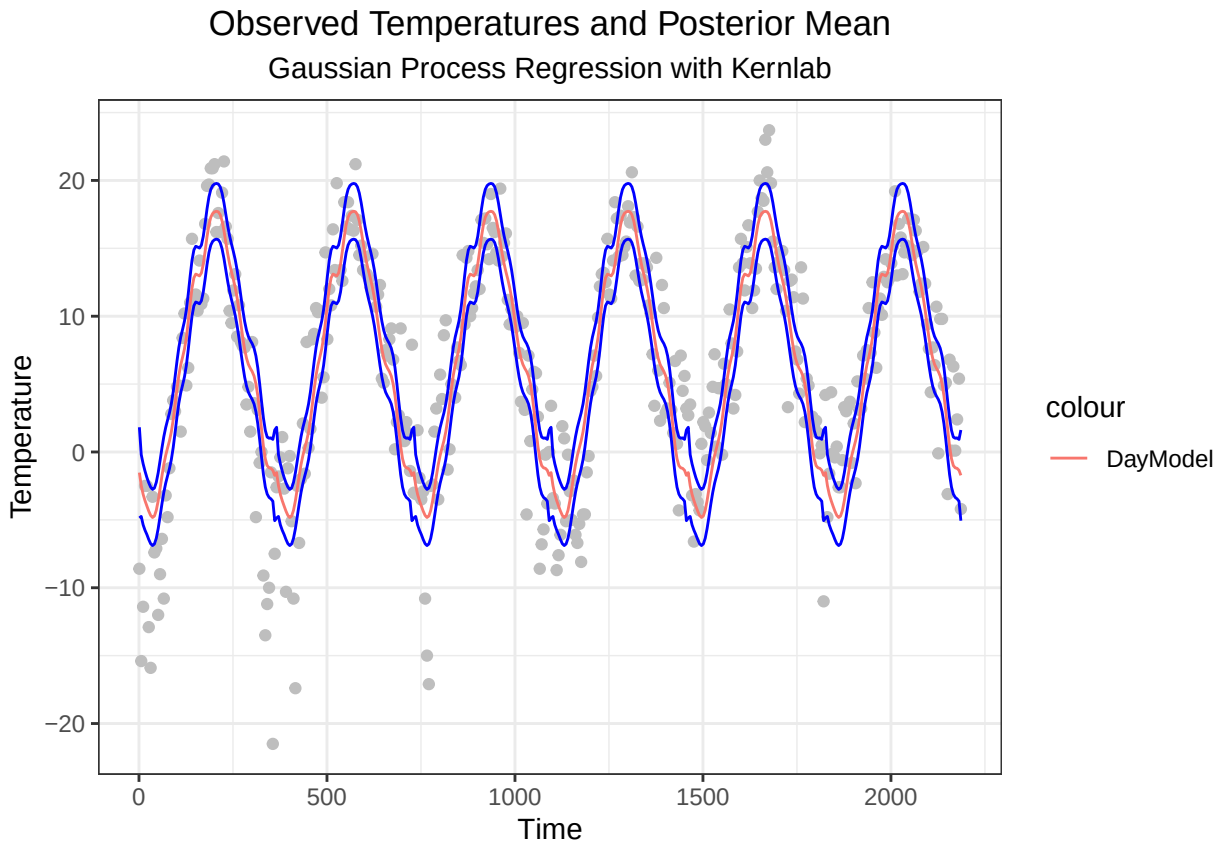
We will now redo task 2 and 3 of this question but this time implementing a model based on the variable *day*:

$$temp = f(day) + \epsilon \begin{cases} \epsilon \sim \mathcal{N}(0, \sigma_n^2) \\ \\ f \sim \mathcal{GP}(0, k(day, day')) \end{cases}$$

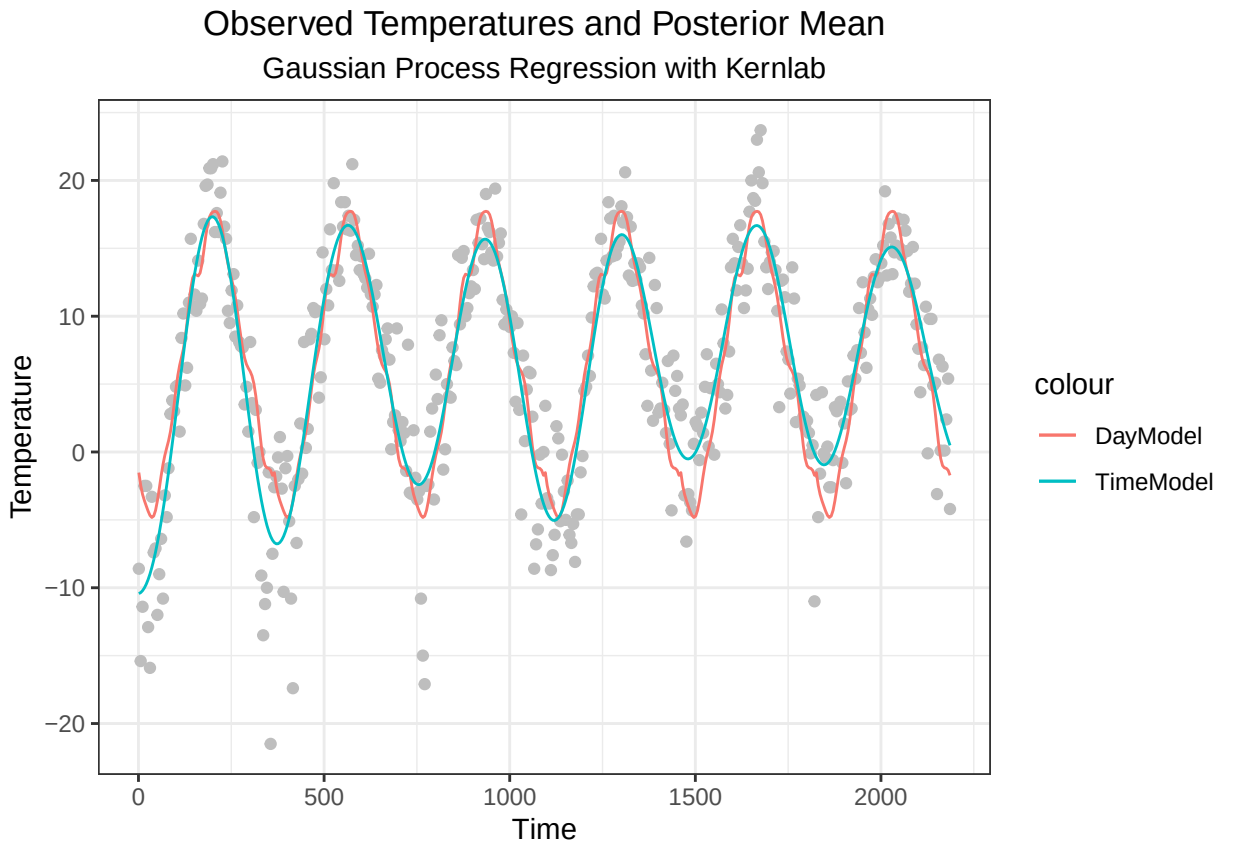
In the plot we will also superimpose the posterior mean computed with the model based on *time*, in order to have a comparison between the two.

```
# Recompute the sigma of the noise
fitted <- lm(temp ~ day + I(day^2), data = data)
sigma = sd(residuals(fitted))

# Compute the actual regression with the help of the package
gpfit<-gausspr(x=data$day, y=data$temp,
               kernel=gkernel(sigmaF = 20, ell = 0.2),
               kpar=list(sigmaF=20, ell=0.2),
               var=sigma^2)
meanPredDay<-predict(gpfit, data$day)

# Fit the GP to the day and plot it
```

```
p0 <- ggplot(data.frame(x = data$time, y = data$temp)) +
  geom_point(aes(x,y), color ="grey") +
  theme(plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5)) +
  labs(title = "Observed Temperatures and Posterior Mean",
       subtitle = "Gaussian Process Regression with Kernlab",
       x = "Time",
       y = "Temperature")+
  geom_line(aes(x,meanPredDay, color = "DayModel"))
  # scale_color_manual(name='')

# Extract the variance of the model
var = Decompotion(X = scale(data$day), y = data$temp,
                  Kernel_function = SquaredExpKernel,
                  sigmaNoise = sigma, X_star = scale(data$day),
                  hyperPar = c(20, 0.2))[[2]]

# Compute the probability bands
B1 <- meanPredDay - 1.96*sqrt(diag(var))
B2 <- meanPredDay + 1.96*sqrt(diag(var))

p0 +
  geom_line(aes(x=data$time, y=B1), color = col[2]) +
  geom_line(aes(x=data$time, y=B2), color = col[2])
```




```
p0+
  geom_line(aes(x, y=meanPredTime, color="TimeModel"))
```



This model shows a cler difference with the previous one: the resulting posterior mean is much more bumpy and perfectly repeats itself year after year. This irregular output is due to the fact that the model tries to average the temperature over different years. For this reason a single outlier may bring the trend of the posterior mean to follow that value, resulting in the irregularities we observed. However, the probability band is much narrower, even at the change of the seasons. Nonetheless, the mean is really different between the two models during the winters of the last three years: they were generally warmer than the average, so while the GP based on the *time* followed the trend of that moment, the GP based on the *day* remained constant and underestimated the actual temperatures.

5 - GP with periodic kernel

We will now try to implement a GP with a periodic kernel. The kernel used has the following form:

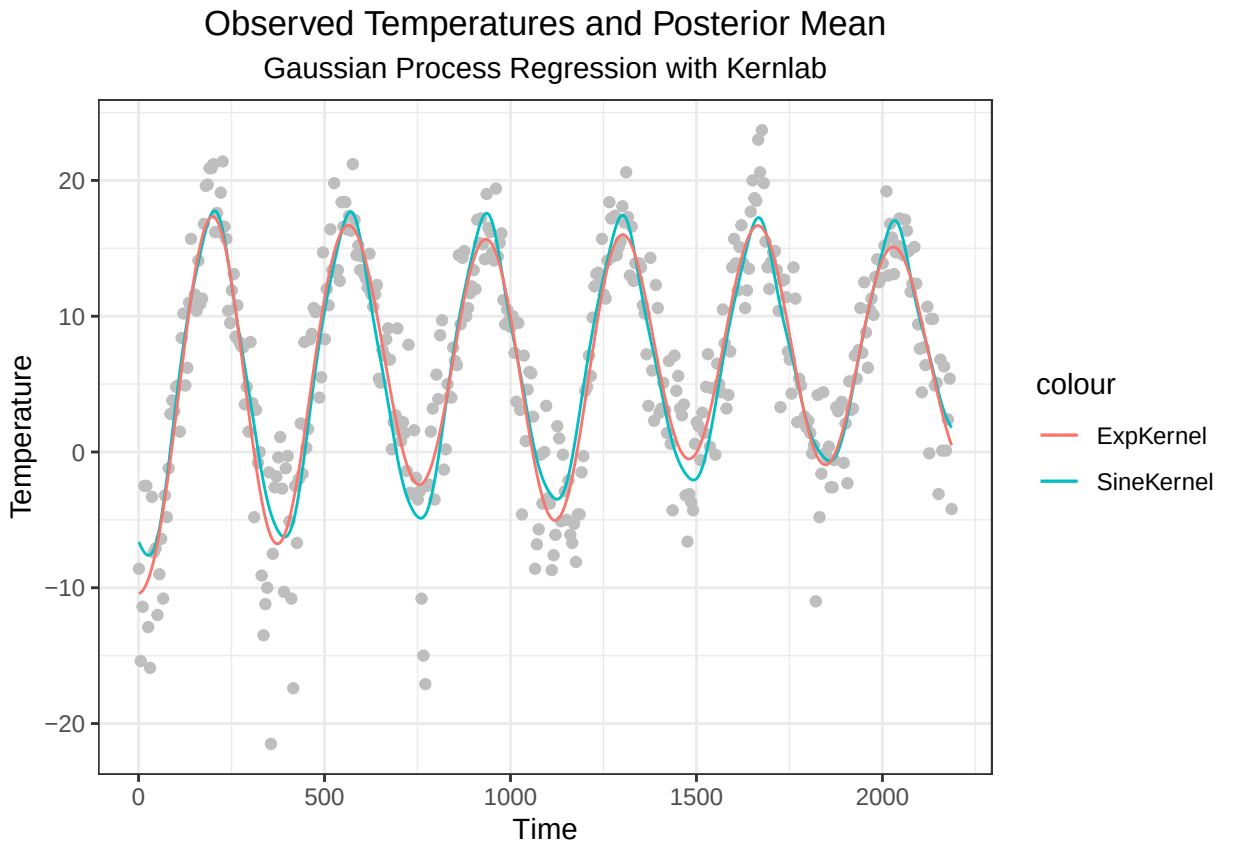
$$k(x, x') = \sigma_f^2 \exp\left\{-\frac{2\sin^2(\pi \mid x - x' \mid /d)}{\ell_1^2}\right\} \exp\left\{-\frac{1}{2}\frac{\mid x - x' \mid^2}{\ell_2^2}\right\}$$

This kernel allows to put two different smoothing parameters, one for the general correlation of the data (ℓ_1) and another for the correlation between the days (ℓ_2). We will set the hyperparameters as $\ell_1 = 1; \ell_2 = 10; d = 365/\texttt{sd(time)}$. The reason for the rather strange period here is that kernlab standardizes the inputs to have standard deviation of 1.

```
# Create a wrapper function for the periodic_kernel
sinkernel<-function(sigmaF, ell1, ell2, d){
```

```
  kernel<-function(x, y){
    n1 <- length(x)
    n2 <- length(y)
    K <- matrix(NA,n1,n2)
    for (i in 1:n2){
      K[, i] = sigmaF^2*
        exp(-2*((sin(pi*(x-y[i])/d))^2)/(ell1^2))*
        exp(-0.5*( (x-y[i])/ell2)^2)
    }
    return(K)
  }
  class(kernel)<-"kernel"
  return(kernel)
}

fitted<- lm(temp ~ time + I(time^2), data = data)
sigma=sd(residuals(fitted))


gpfit<-gausspr(x=data$time, y=data$temp,
               kernel=sinkernel(sigmaF = 20, ell = 1, ell2=10, d=365/sd(data$time)),
               kpar=list(sigmaF=20, ell=0.2),
               var=sigma^2)

meanPredTimeSine<-predict(gpfit, data$time)
```

```
p0 <- ggplot(data.frame(x = data$time, y = data$temp)) +
  geom_point(aes(x,y), color ="grey") +
  theme(plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5)) +
  labs(title = "Observed Temperatures and Posterior Mean",
       subtitle = "Gaussian Process Regression with Kernlab",
       x = "Time",
       y = "Temperature")+
  geom_line(aes(x,meanPredTimeSine, color='SineKernel'))

p0+
  geom_line(aes(x=data$time, y=meanPredTime, color='ExpKernel'))
```



The GP periodic kernel model seems somehow a mix between the GP based on *time* and the GP based on *day*: the posterior mean is much smoother in general, with a generally thing probability band. The trend has a periodic structure but is flexible and adapts to the rising of the temperature for each year, differently from the GP based on *day*. Having said that, the model does not perform well in case of extreme events: the change of the season is sometimes either overestimated (winter) or underestimated (summer). The days of frost or heat waves are not capture by this model, meaning that is not capable to capture extreme (but still periodic and somehow predictable) events. A smaller ℓ_2 may compensate this lack of adaptability, giving to the model more importance to the records of the days immediately before/after.

Question 3 - Gaussian Process Classification with `kernlab`

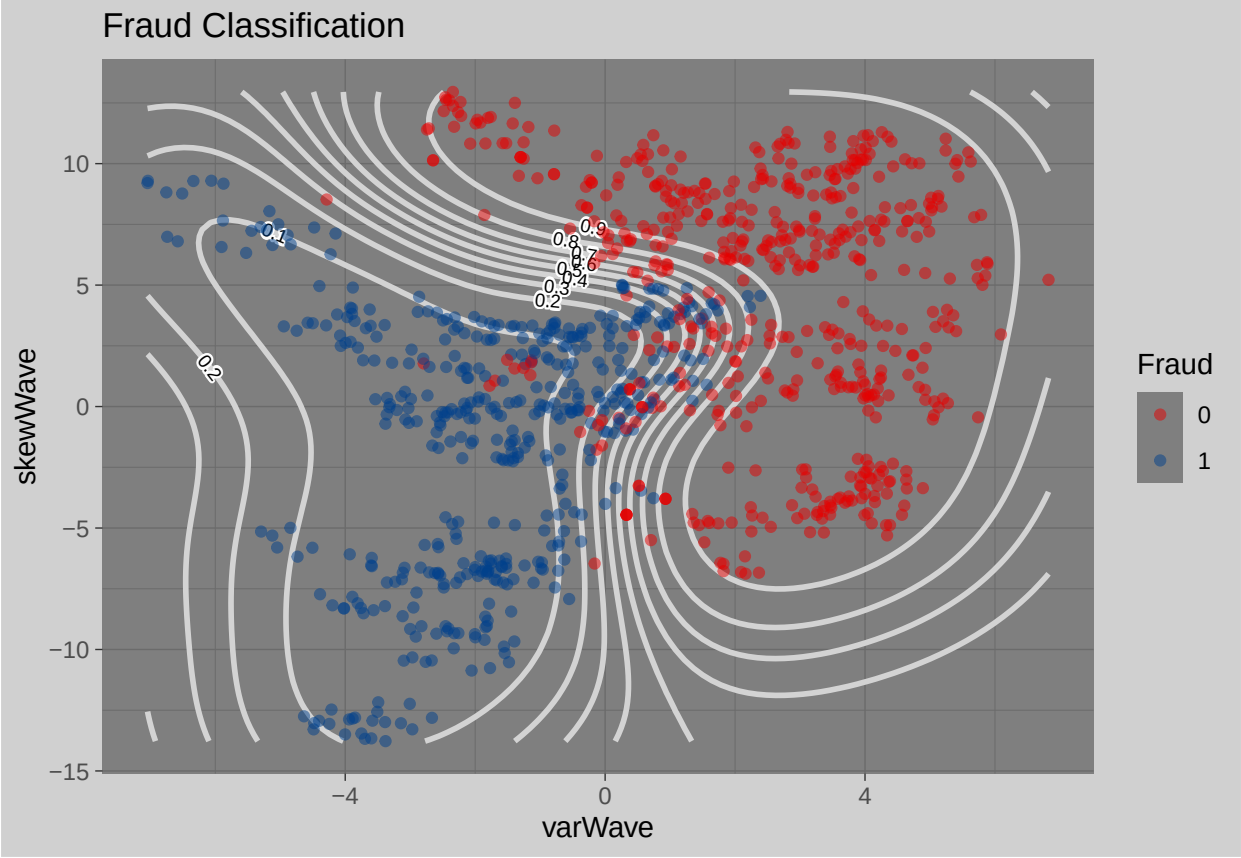
```
datalink ="https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/banknoteFraud.csv"
data <- read.csv(datalink, header=FALSE, sep=",")
names(data) <- c("varWave","skewWave","kurtWave","entropyWave","fraud")
data[,5] <- as.factor(data[,5])

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

train = data[SelectTraining, ]
test = data[-SelectTraining, ]
```

3.1 - GP model and Contour Map

```
# fit gaussian model
GPfitFraud <- gausspr(fraud ~ varWave + skewWave, data=train)
```

```
## Using automatic sigma estimation (sigest) for RBF or laplace kernel
```

```
predTrain = predict(GPfitFraud, train)

# class probabilities
probPreds = predict(GPfitFraud, train[, c("varWave", "skewWave")], type="probabilities")
x1 = seq(min(train$varWave), max(train$varWave), length = 100)
x2 = seq(min(train$skewWave), max(train$skewWave), length = 100)
gridPoints = AtmRay::meshgrid(x1, x2)
gridPoints = cbind(c(gridPoints$x), c(gridPoints$y))

gridPoints = data.frame(gridPoints)
names(gridPoints) = c("varWave", "skewWave")
probPreds = predict(GPfitFraud, gridPoints, type="probabilities")

ggplot() +
  stat_contour(aes(x = gridPoints$varWave,
                   y = gridPoints$skewWave,
                   z = probPreds[,1]),
               color = "lightgray", size=1) +
  metR::geom_text_contour(aes(x = gridPoints$varWave,
                              y = gridPoints$skewWave,
                              z = probPreds[,1]),
                          stroke = 0.2, size=2.5) +
  geom_point(aes(x = train$varWave,
                 y = train$skewWave, color = train$fraud),
             alpha = 0.5) +
  labs(title = "Fraud Classification",
       x = "varWave", y = "skewWave",
       color = "Fraud") +
  theme_dark() +
  theme(plot.background = element_rect(fill = "#d0d0d0"),
        legend.background = element_rect(fill = "#d0d0d0")) +
  scale_color_manual(values=c("#e60000", "#00418c"))
```




```
cm = table(train$fraud, predTrain)
acc = sum(diag(cm))/sum(cm)

kableExtra::kable(cm)
```

	0	1
0	503	41
1	18	438

```
cat(paste("Train Accuracy =", acc))
```

```
## Train Accuracy = 0.941
```

3.2 - Evaluation of test data

```
predTest = predict(GPfitFraud, test)

cm = table(test$fraud, predTest)
acc = sum(diag(cm))/sum(cm)

kableExtra::kable(cm)
```

	0	1
0	199	19
1	9	145

```
cat(paste("Test Accuracy =", acc))
```

```
## Test Accuracy = 0.924731182795699
```

3.3 - Different covarietes

```
# fit gaussian model
GPfitFraud <- gausspr(fraud ~ ., data=train)
```

```
## Using automatic sigma estimation (sigest) for RBF or laplace kernel
```

```
# predict
predTrain = predict(GPfitFraud, train)
predTest = predict(GPfitFraud, test)
# confusion matrixes
cmTrain = table(train$fraud, predTrain)
cmTest = table(test$fraud, predTest)
# accuracies
accTrain = sum(diag(cmTrain))/sum(cmTrain)
accTest = sum(diag(cmTest))/sum(cmTest)
```

```
## Train Accuracy = 0.997
## Test Accuracy = 0.994623655913978
```

```
## Train Confusion Matrix
```

	0	1
0	541	3
1	0	456

```
## Test Confusion Matrix
```

	0	1
0	216	2
1	0	154

We can see we have better performance at last trial with all coverietes both on train and test data. This makes the model more complex because we increase the variable count. Also this performance increased so much, it might be overfit but we do not have enough test data to conclude this.