Lab4 Adv ML

Andreas Stasinakis(andst745)

October 11, 2019

Contents

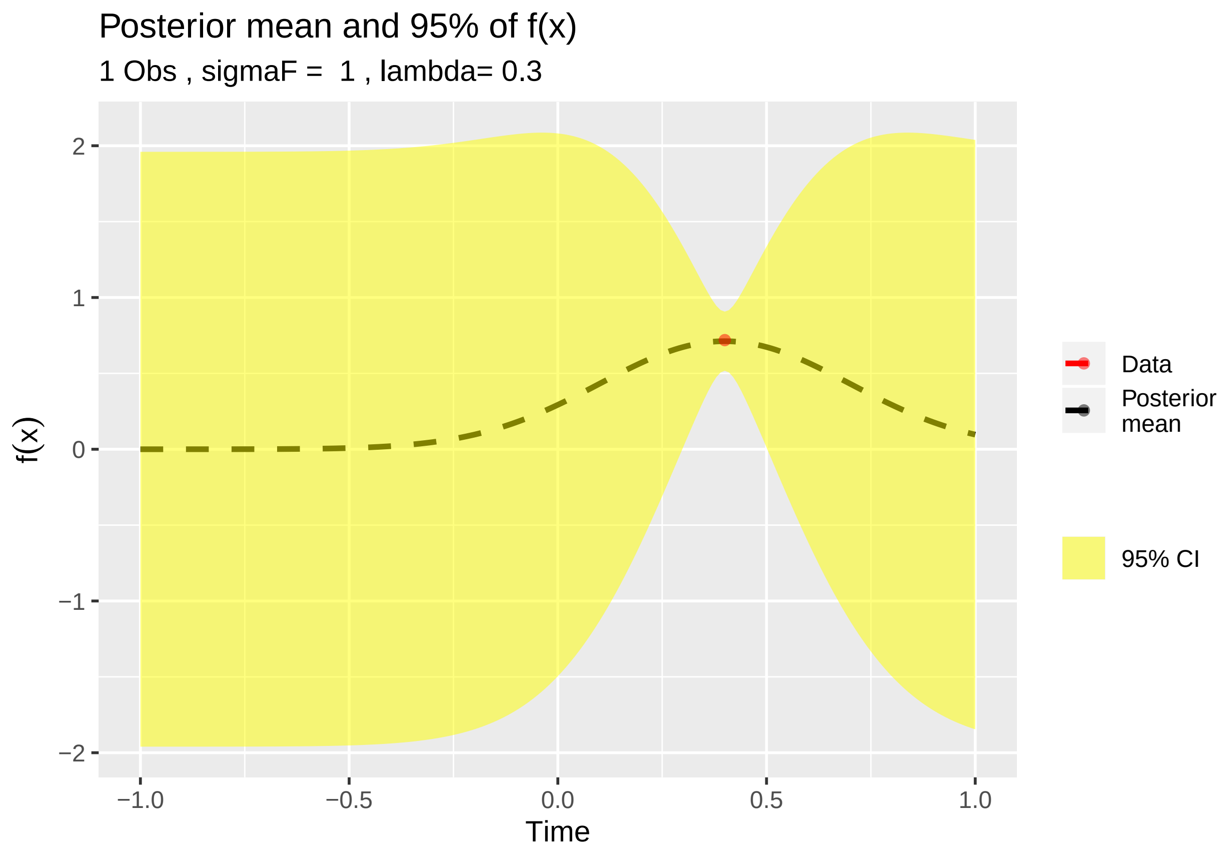
Question 1: GP regression

```r
#import the packages we need
library(kernlab)
library(mvtnorm)
library(ggplot2)
library(kernlab)
library(AtmRay)
library(stringr)
library(gridExtra)
```

```r
#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)

```r
#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
```

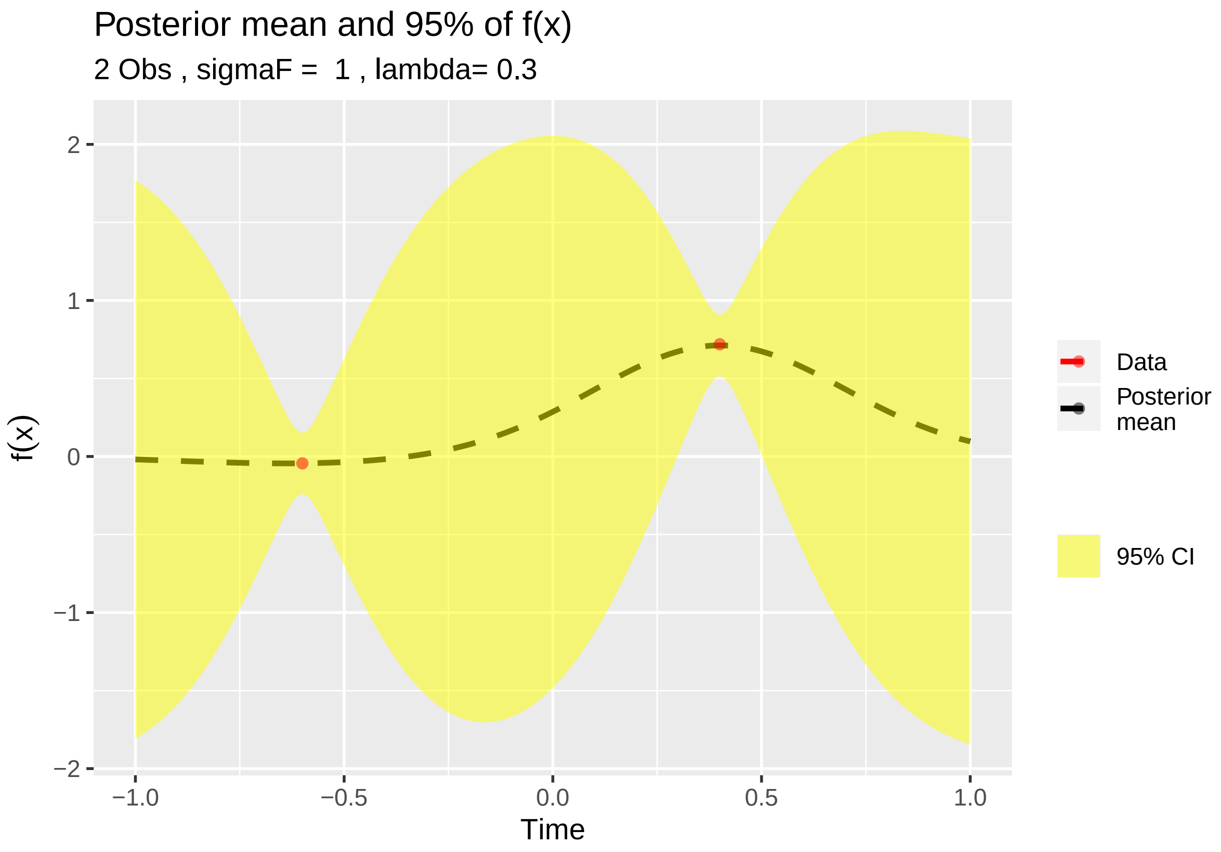


In this Question, we want to implement a GP regression using our own functions. Our target is to caclulate the posterior mean and the posterior covariance matrix. After that we can make predictions or generate from the posterior distribution. We first update our prior knowledge using only one observations. 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")
```

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

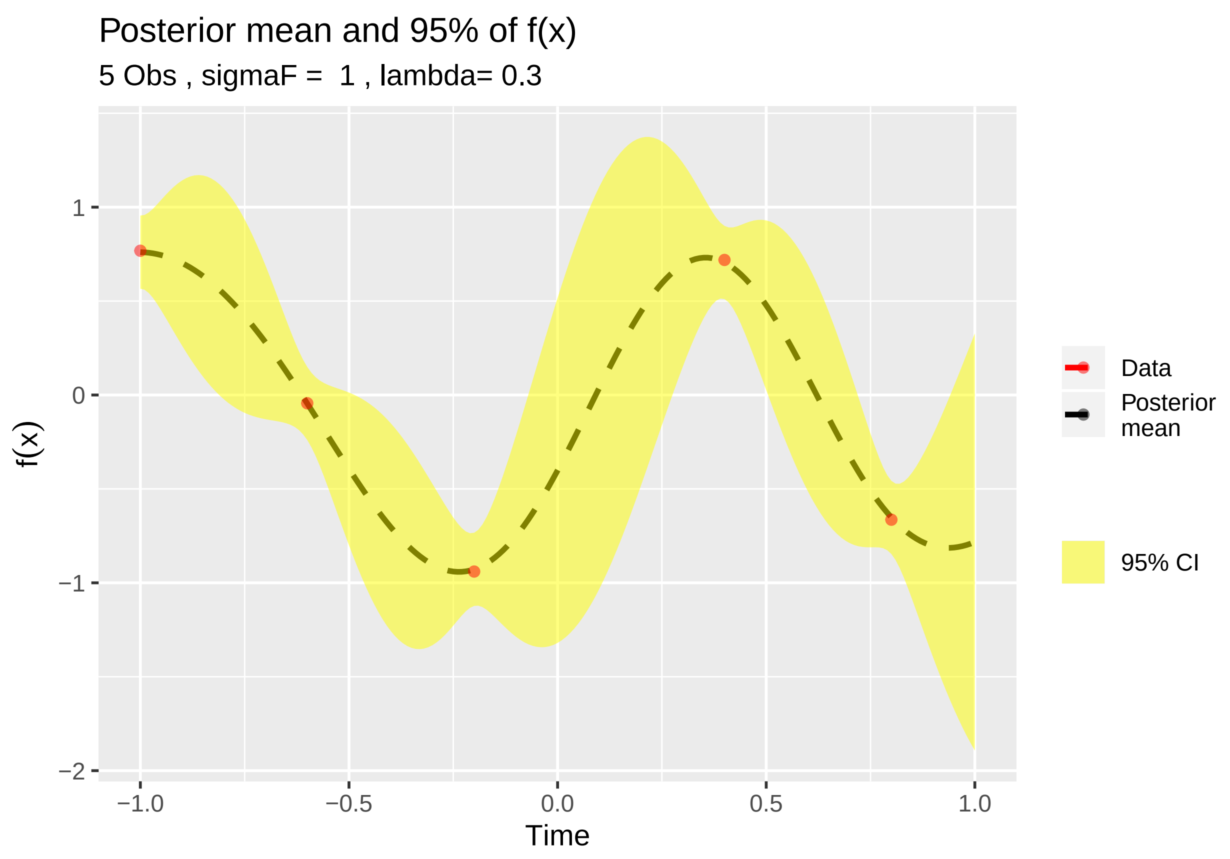


We use one more observation of your data, 2 in total. The output is more precise than with only one observation, the boundaries start to became more tight, but we need still more observations.

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

```r
#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
```

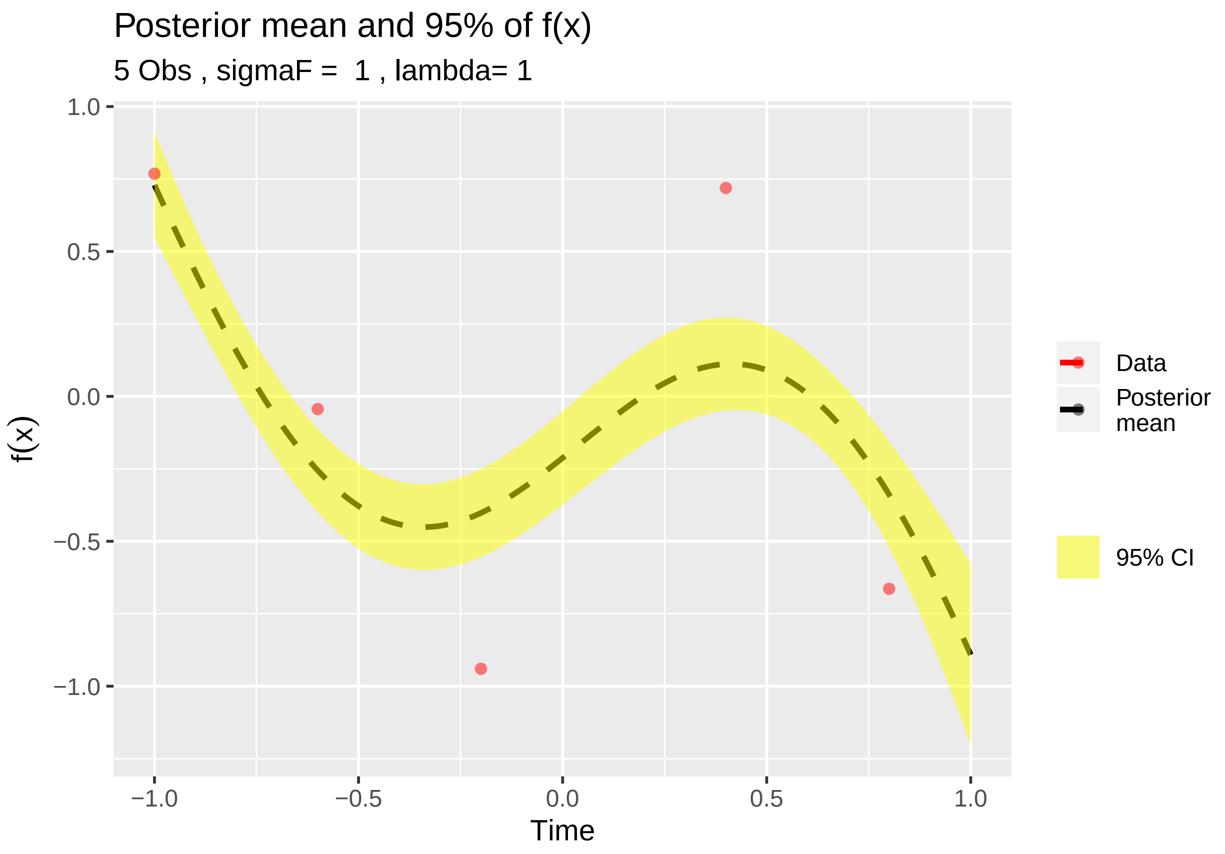


Using 5 observations, we get a smooth function which starts to have a specific shape. Moreover, we can now see a patern of the function, and we can use that posterior means and variances in order to generate from that posterior and make predictions. Of course, the more observations, the more accurate 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")
```

```r
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

2.1 Prepare the data and define the Kernel function

```r
#Function we use in the first question
SQexpKernel <- function(sigmaf, ell)
{
  rval <- function(x, y) {

    return(sigmaf^2*exp(-0.5*( (x-y)/ell)^2 ))
  }
  class(rval) <- "kernel"
  return(rval)
}
```

```r
#import the data
data = read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/
Code/TempTullinge.csv", header=TRUE, sep=";")

#Now we follow the instructions
data$time = seq(1,nrow(data)) #add new col to  count the total number of days
data$day = data$time%%365 #new col for adding days
data$day[which(data$day == 0)] = 365 # instead of 0 we want 365

#We will not use all the data, just every 5th row
index = seq(1,nrow(data), by = 5)
newdata = data[index,]


#The package requires some weird black magic for the kernl matrix
SQexpK = SQexpKernel(1,1) #run the function for two parameters
SQexpK(x = 1,y = 2) #evaluate it for two observations
```

```
## [1] 0.6065307
```

```r
K = kernelMatrix(SQexpK,c(1,3,4),c(2,3,4)) #Kernel matrix for X_star
K
```

```
## 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
```

2.2 Estimate a GP regression model and Posterior mean

```r
#The model is a function of the input + NOISE
# Estimating the noise variance from a two degree polynomial fit
set.seed(12345)
polyFit <- lm(temp ~  time + I(time^2), data = newdata)
sigmaNoise = sd(polyFit$residuals)
allColors = c("red", "black", "yellow")
#Hyperparameters
sigmaf <- 20
ell <- 0.2
```

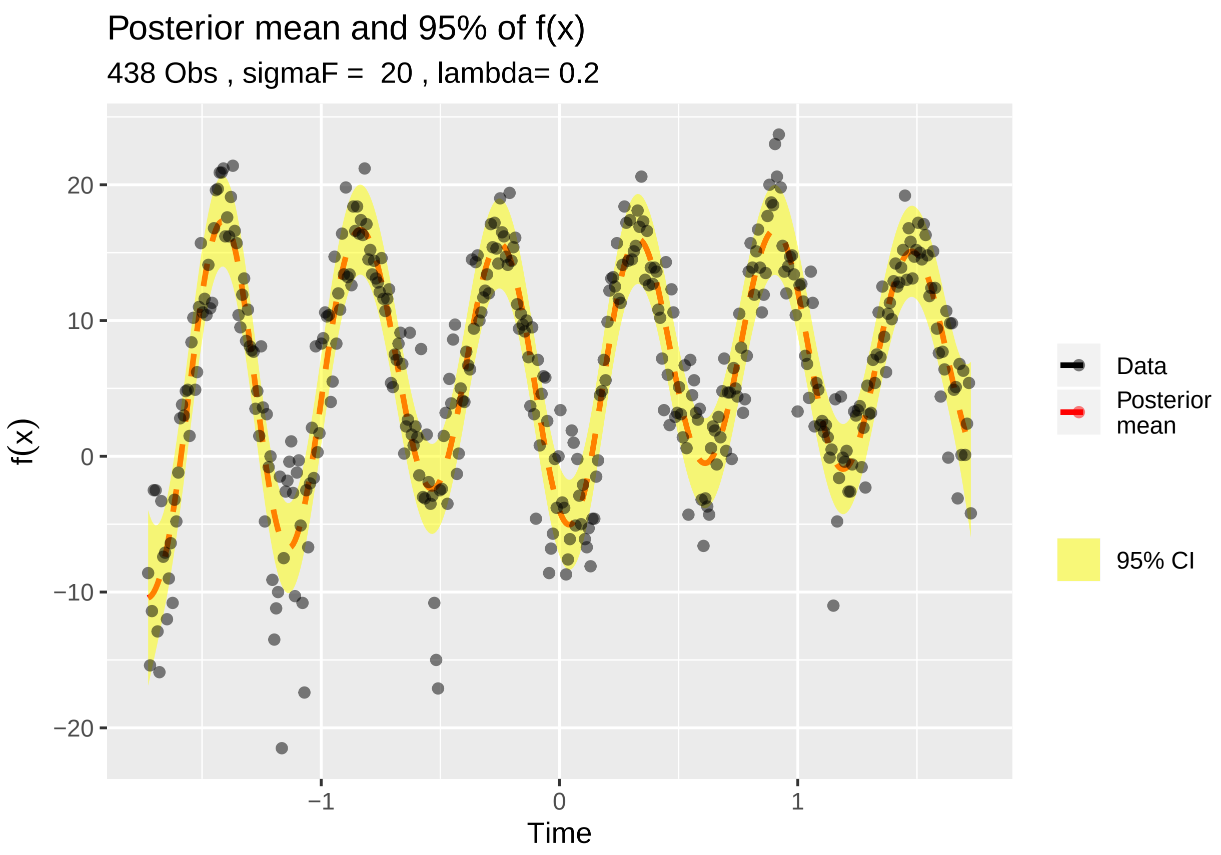
```r
#Fit a gaussian Proccess with target variable the temperature
#and predictios the variable  Time
GPfit =  gausspr(x = newdata$time,#predictor variable
                y = newdata$temp, #response variable
                kernel = SQexpKernel, #Kernel function
                kpar = list(sigmaf = sigmaf, ell=ell), #Hyperparameters
                var = sigmaNoise^2) #Noise of the model

#Make prediction using all data
meanPred = predict(GPfit, newdata$time)


post_plot  = ggplot()+
    geom_line(mapping = aes(x = newdata$time,y = meanPred, col = "Posterior\nmean Time"),
              lty = 2, size = 1.5)+
    geom_point(mapping = aes(x = newdata$time, y = newdata$temp,
                             col = "Data"), alpha = 0.5)
```

In this question, we want to model the temperature in a function of time plus some variance. We first estimate the variance using a quadratic linear regression. We also use as hyperparameters, $\sigma_f = 20$ and $l = 0.2.$, in order to fit the *GP*. We use the package kernlab, we fit the model and make predictions in order to get the posterior mean. We plot the posterior mean together with the 95% probability interval in the next question.

2.3 Compute Posterior variance using posteriorGP

```r
#now we will use the same functions as in task 1 in order to estimating the
#Posterior variance and plot 95% intervals
#we need scaled data in order to have similar results with the package
X = scale(newdata$time)
y = newdata$temp
Xstar = X  #In this case we evaluate the function in the same X points
hyperPar = c(sigmaf,ell)

#use posteriorGP function in order to estimate the variance of the posterior
varPost = posteriorGP(X = X,y = y,Xstar = Xstar,hyperParam = hyperPar,
                      Kernel_function = SquaredExpKernel,
                      sigmaNoise = sigmaNoise)[[2]]


#Compute the 95 % credible intervals
low_CI = meanPred -1.96*sqrt(diag(varPost))
upper_CI = meanPred +1.96*sqrt(diag(varPost))

#Prepare a data frame from the plots
dfKlearn = data.frame(Xstar, #x -axis values
                      meanPred, #mean of the posterior
                      low_CI,#Lower CI of Post
                      upper_CI)
#upper CI of Post
colnames(dfKlearn) = c("Xgrid","PostMean", "Lower CI",  "Upper CI")

#Use the plot function to plot that beautiful model
pltGP = plot_FUN(dfKlearn,X,y,allColors,hyperPar,length(X))
```

```
pltGP
```



We also plot the 95% probabilities intervals. As we can see there are quite a few points that outside of the intervals, but in general the model captures the shape of the data. We could may increase the variance of the function in order to take more wide intervals.

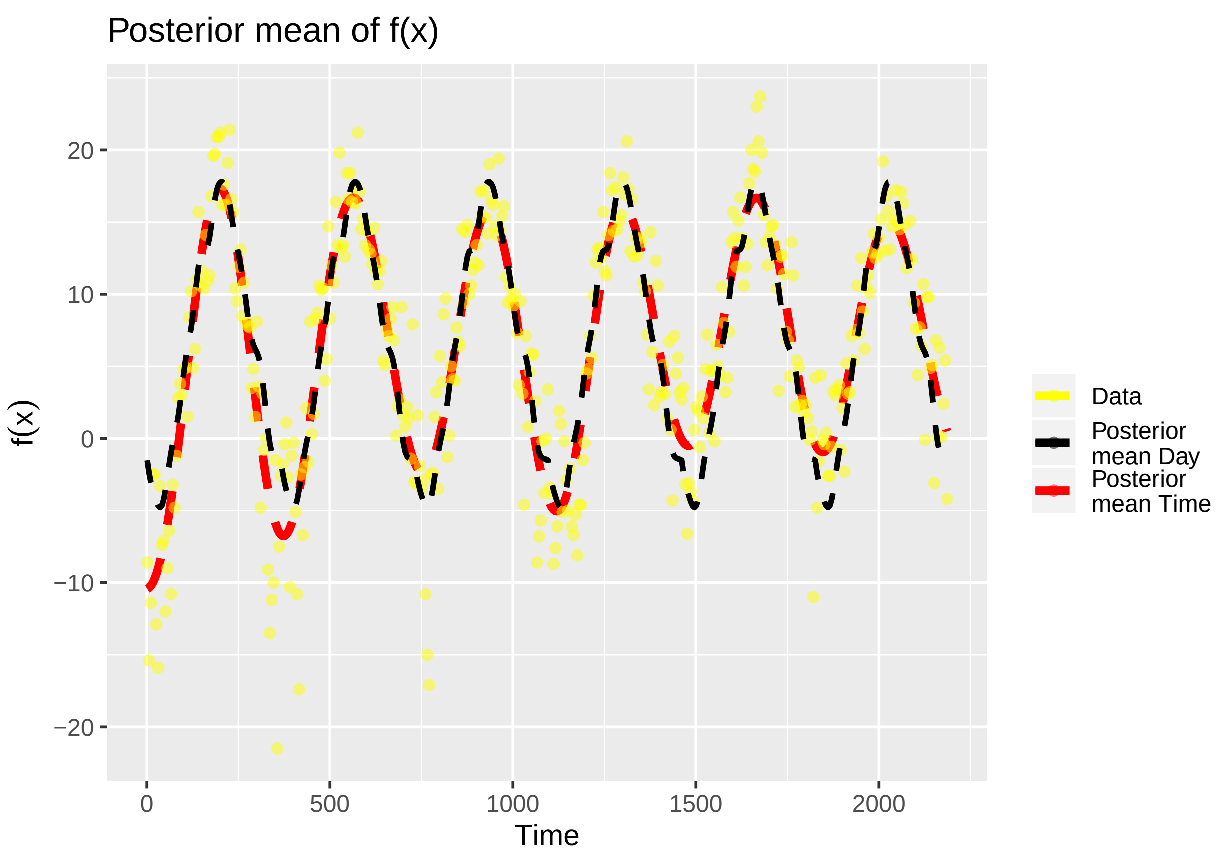
2.4 Same procedure, different Covariates(day)

```
#In this question we change the predictior from time to day
set.seed(12345)
#same as before, we just change the X variable
GPfit_day = gausspr( newdata$day,newdata$temp, kernel = SQexpKernel,
                  kpar = list(sigmaf = sigmaf, ell=ell), var = sigmaNoise^2)

#We make the predictions
meanPred_day = predict(GPfit_day, newdata$day)

#We now plot the result in order to compare it with the above model
#We need to do the same procedure as before in order to take the variance of the
#Posterior distribution

X = scale(newdata$day) #scale the predictor
y = newdata$temp
Xstar = X
X_axis = newdata$time #We are asked to plot the preds vs Time not day
hyperPar = c(sigmaf,ell)
```

```r
varPost_day = posteriorGP(X = X,y = y,Xstar = Xstar,hyperParam = hyperPar,
                          Kernel_function = SquaredExpKernel,
                      sigmaNoise = sigmaNoise)[[2]]

#Compute the 95 % credible intervals
low_CI_day = meanPred_day -1.96*sqrt(diag(varPost_day))
upper_CI_day = meanPred_day +1.96*sqrt(diag(varPost_day))

dfKlearn_day = data.frame(X_axis,meanPred_day,low_CI_day,upper_CI_day)

colnames(dfKlearn_day) = c("Xgrid","PostMean", "Lower CI",  "Upper CI")

#plot the posterior mean and the 95 % Credible interval
plt_day = plot_FUN(dfKlearn_day,X_axis,y,allColors,hyperPar,length(X))

#We also add the posterior mean to the previous plot in order to compare
post_plt  = post_plot +
    geom_line(mapping = aes(x = newdata$time,y = dfKlearn_day[,2],
                            col = "Posterior\nmean Day"),
              lty = 2, size = 1)+
    labs(title = paste("Posterior mean of",expression(f(x))),
         x = "Time", y = expression(f(x)))+
    scale_color_manual(values = c("Posterior\nmean Time" = allColors[1],
                                  "Posterior\nmean Day" = allColors[2],
                                  "Data" = allColors[3]), name = "")
```

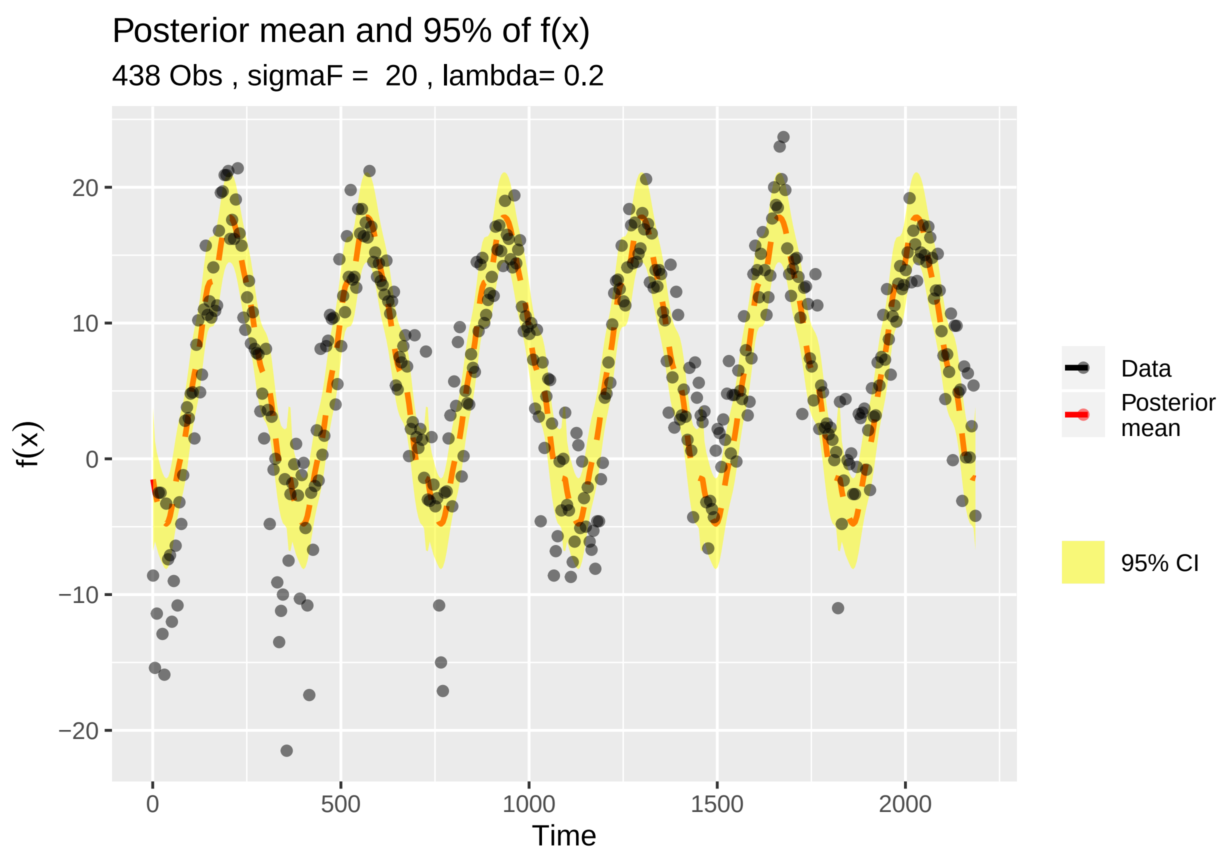
In this task, we change the model's predictor. Now the response variable is a function of day,not time. We add the posterior mean to the plot of the posterior mean of the previous model in order to make comparisons. As we can observe, the posterior mean of the day model, also captures the shape of the function.

```r
post_plt
```



We also plot the 95% probability interval for the day model. It is obvious that model also captures the shape of the function but it has higher picks and it is not that smooth in contrast to the time model. We can also say that the 95% intervals are a little bit tighter.

```
plt_day
```

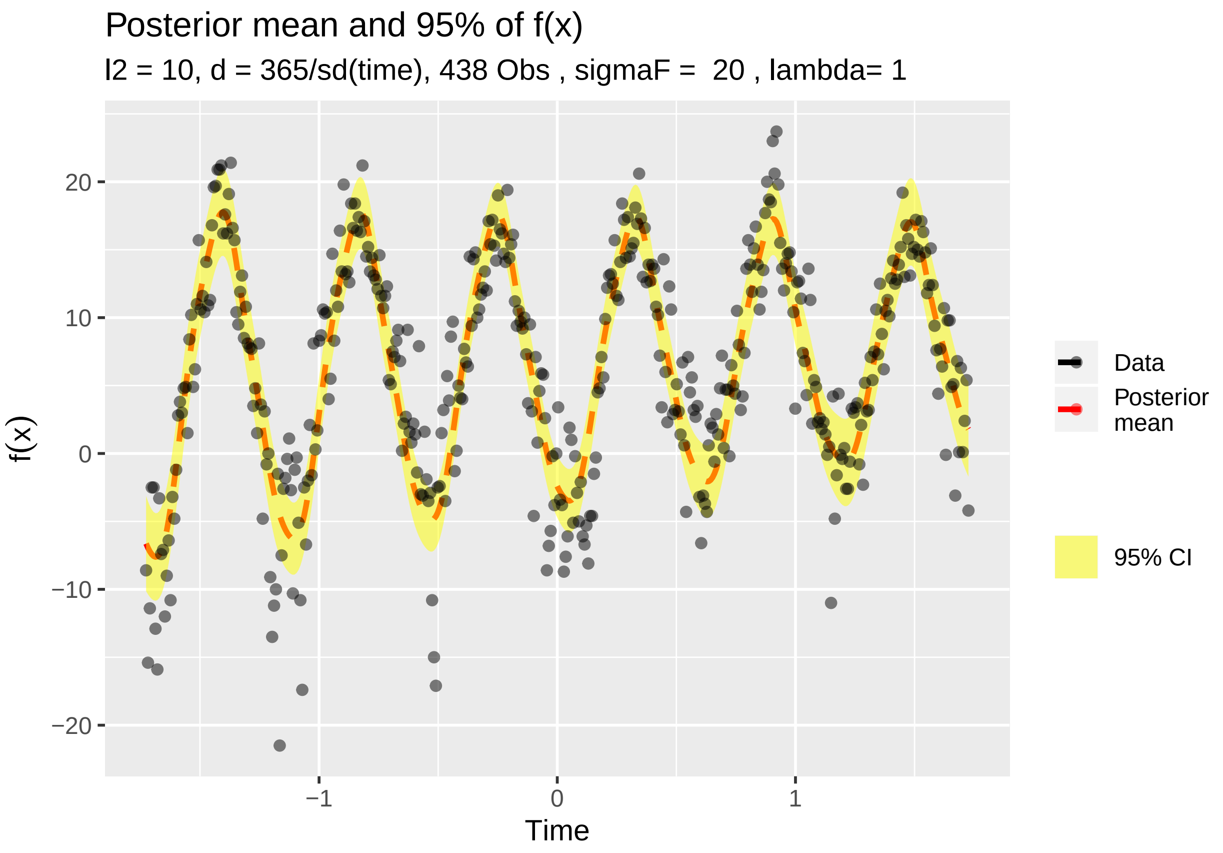


2.5 Same procedure, different kernel function(Periodic) and comparison

```
#We need this function in order to find the posterior variance
perKern = function(x1,x2,par){


    sigmaF = par[1]
    ell1 = par[2]
    ell2 = par[3]
    d = par[4]
    n1 <- length(x1)
    n2 <- length(x2)
    K <- matrix(NA,n1,n2)

    for (i in 1:n2){
      p1 = exp(-2*(sin((pi*abs(x1-x2[i]))/d))^2/(ell1^2))
      p2 = exp(-((abs(x1-x2[i]))^2)/((ell2^2)*2))

      K[,i] = (sigmaf^2)*p1*p2
    }

  return(K)
}
```

```r
Periodic_Kernel <- function(sigmaf, ell1,ell2,d)
{
  rval <- function(x, y) {

    p1 = exp(-2*(sin((pi*abs(x-y))/d))^2/(ell1^2))
    p2 = exp(-((abs(x-y))^2)/((ell2^2)*2))



    return((sigmaf^2)*(p1)*(p2))
  }
  class(rval) <- "kernel"
  return(rval)
}
```

```r
sigmaf = 20
ell1 = 1
ell2 =  10
d = 365/sd(newdata$time)

#Fit a gaussian Proccess with target variable the temperature
#and predictios the variable  Time
GPfit_periodic =  gausspr(x = newdata$time,#predictor variable
                 y = newdata$temp, #response variable
                 kernel = Periodic_Kernel, #Kernel function
                 kpar = list(sigmaf = sigmaf, ell1 = ell1,ell2 = ell2,
                       d = d), #Hyperparameters
                 var = sigmaNoise^2) #Noise of the model

#Make prediction using all data
meanPred_periodic = predict(GPfit_periodic, newdata$time)

#now we will use the same functions as in task 1 in order to estimating the
#Posterior variance and plot 95% intervals

#we need scaled data in order to have similar results with the package
X = scale(newdata$time)
y = newdata$temp
Xstar = X  #In this case we evaluate the function in the same X points
hyperPar = c(sigmaf,ell1,ell2,d)

set.seed(12345)

varPost_period = posteriorGP(X = X,y = y,Xstar = Xstar,hyperParam = hyperPar,
                             Kernel_function = perKern,
                      sigmaNoise = sigmaNoise)[[2]]


#Compute the 95 % credible intervals
low_CI_period = meanPred_periodic -1.96*sqrt(diag(varPost_period))
upper_CI_period = meanPred_periodic +1.96*sqrt(diag(varPost_period))

#Prepare a data frame from the plots
dfKlearn_periodic = data.frame(Xstar, #x -axis values
                        meanPred_periodic, #mean of the posterior
                        low_CI_period,#Lower CI of Post
                        upper_CI_period)
#upper CI of Post
colnames(dfKlearn_periodic) = c("Xgrid","PostMean", "Lower CI",  "Upper CI")

#Use the plot function to plot that beautiful model
all_par = c("l2 = 10, d = 365/sd(time), 438")
pltGP_period = plot_FUN(dfKlearn_periodic,X,y,allColors, hyperPar,all_par)
pltGP_period
```



```
#grid.arrange(grobs = list(pltGP,plt_day,pltGP_period))
```

Finally, we change the kernel function. Therefore instead of the square exponential kernel function, we use a generalization of the periodic kernel function, which is more complicated and it depends on 4 hyperparameters. From the plot, we can see that the last model, has lower picks and mimima. ALso the interval are tighter than the first and the second model.

Question 3: GP Classification with kernlab

3.1 Fit a GP model, Contour plot over 2 covariates and predictions

In this task, we use the package kernlab in order to fit a classification GP. We train the model using only 2 out of for covariates for default parameters. We plot the contour plot and we also calculate the accuracy of the model for the predictions of training data. All the accuracies are presented in the last question in order to compare them.

```r
#Import and prepare the data
data <- read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/
GaussianProcess/Code/banknoteFraud.csv", 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)
#split the data
train = data[SelectTraining,]
test = data[-SelectTraining,]

#fit the model using two covariates
GPfit_class = gausspr(fraud ~ varWave + skewWave,data = train)
```

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

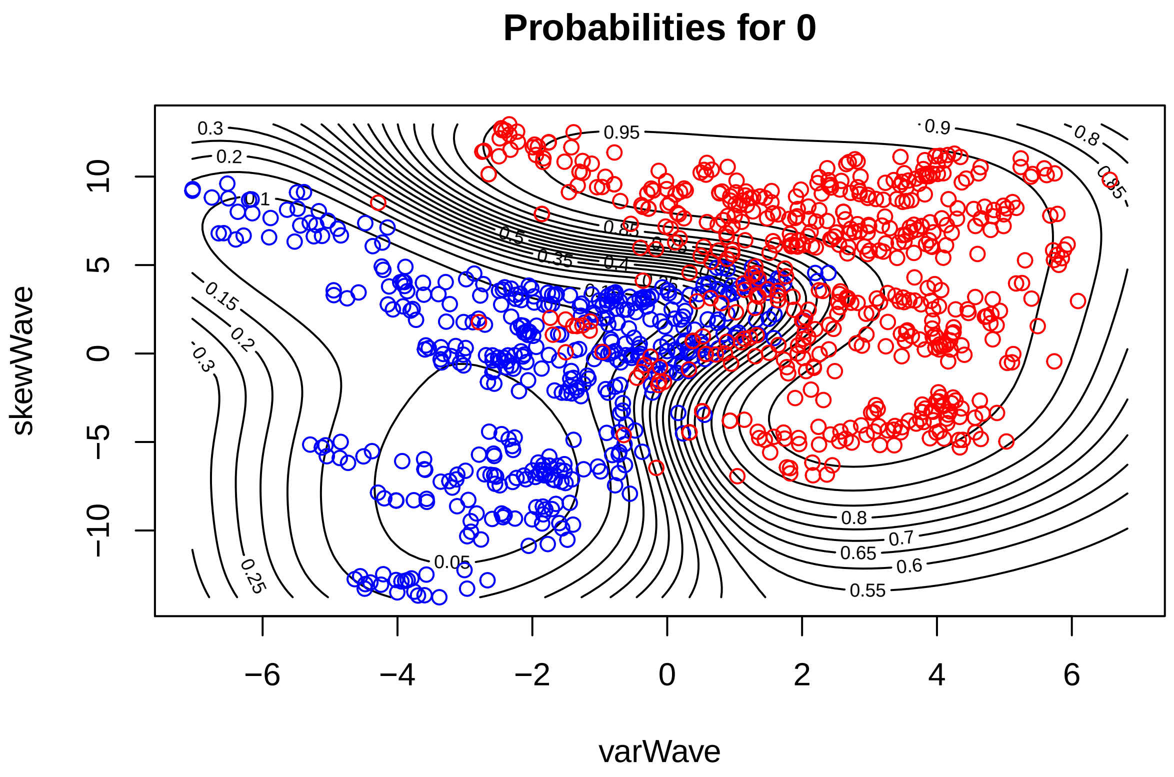
```r
probClass = predict(GPfit_class,train[1:2],type = "probabilities")


# class probabilities
x1 <- seq(min(data[,1]),max(data[,1]),length=100)
x2 <- seq(min(data[,2]),max(data[,2]),length=100)
gridPoints <- meshgrid(x1, x2)
gridPoints <- cbind(c(gridPoints$x), c(gridPoints$y))

gridPoints <- data.frame(gridPoints)
names(gridPoints) <- names(data)[1:2]
probClass <- predict(GPfit_class, gridPoints, type="probabilities")


# # Plotting for Probabilities for being 0
# contour(x1,x2,matrix(probClass[,1],100,byrow = TRUE), 20,
#         xlab = names(gridPoints)[1], ylab = names(gridPoints)[2],
#         main = 'Probabilities for 0')
# points(train[which(train$fraud == 1),1],train[which(train$fraud == 1),2],
#        col="blue")
# points(train[which(train$fraud == 0),1],train[which(train$fraud == 0),2],
#        col="red")
#
# # Plotting for Probabilities for being 0
# contour(x1,x2,matrix(probClass[,2],100,byrow = TRUE), 20,
#         xlab = names(gridPoints)[1], ylab = names(gridPoints)[2],
#         main = 'Probabilities for 0')
# points(train[which(train$fraud == 1),1],train[which(train$fraud == 1),2],
#        col="blue")
# points(train[which(train$fraud == 0),1],train[which(train$fraud == 0),2],
#         col="red")

#confusion matrix and accuracy
predClass = predict(GPfit_class,train[1:2])
conf_matrix = table(predClass, train$fraud)
accur = length(which(predClass == train$fraud))/nrow(train)

#print the confusion matrix
knitr::kable(x = conf_matrix, caption = "confusion matrix for training data")
```

Table 1: confusion matrix for training data

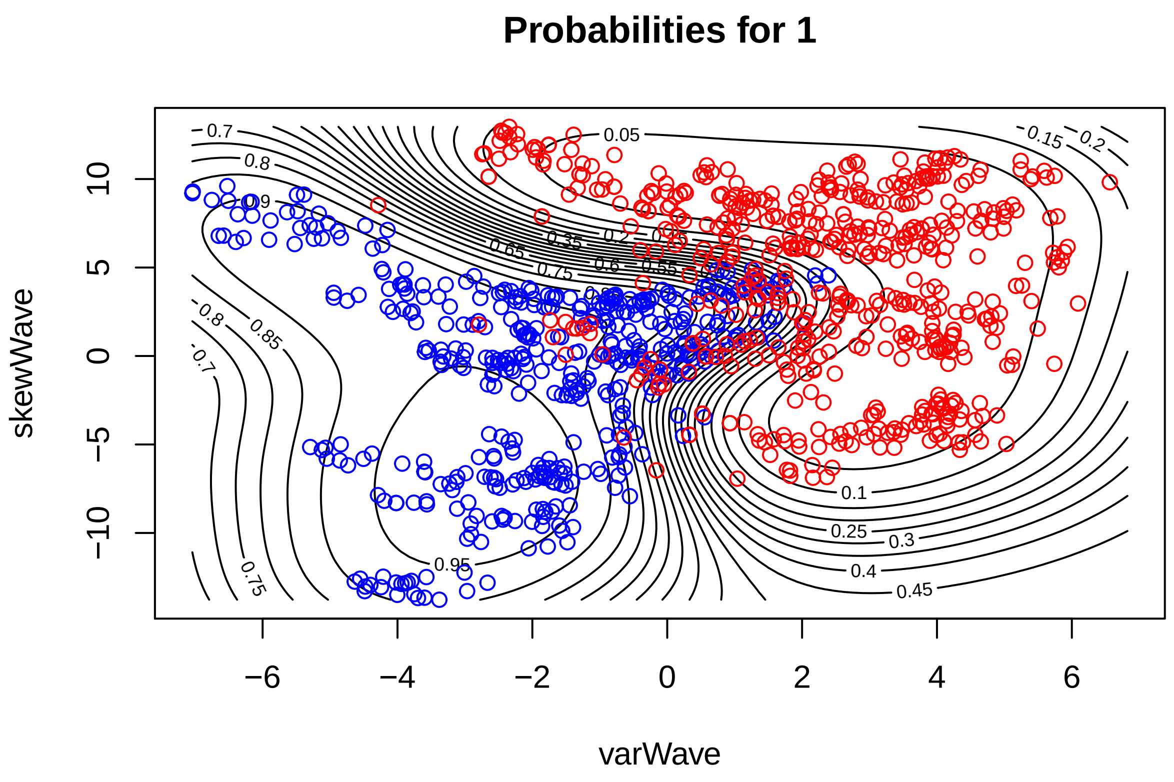
	0	1
0	512	24
1	44	420

We first start with plotting the probabilities of the target value being one. We can see that the probabilities are really high when "varWave" is higher than -2. The highest probabilities(0.95) can be found for "varWave" close to -1 and "skewWave" close to 10. So we expect that that area will be the core of the probability of fraud being 0. We also add the training points where blue points are for fraud equal to 1 and red for fraud equal to zero.

As expected, the majority of the points for fraud equal to 0(red points) are in the top right corner where the probabilities were really high.



We now plot the contour plot for the probabilities of fraud being 1. It is obvious that we expect that most of the points for fraud equal to 1(blue points) to be in the low left corner where the probability is the highest(0.95). On the other hand, it is quite unlikely to find blue points in the top right corner, where the core of the red points is. We also add the training points in order to verify the above statement.



In conclusion, we can say that the model does a decent job as a classifier. It is able to find the boundaries, for most of the different values of the two covariate random variables, except for some cases. More specific, for "varWave" and "skewWave" approximately between -2 and 2, we can not see clear boundaries.

3.2 Generalization of the model

```
#Now we will use the model and make predictions using the test data
#We want to check if it generalizes efficient
pred_test = predict(GPfit_class,test[1:2])
conf_matrix_test = table(pred_test, test$fraud)
accur_test = length(which(pred_test == test$fraud))/nrow(test)

#print the confusion matrix
knitr::kable(x = conf_matrix_test, caption = "confusion matrix for test data")
```

Table 2: confusion matrix for test data

	0	1
0	191	9
1	15	157

Of course the above results are not capable of making general assumptions for the model. In order to see if it is a good model, we have to test it for unseen data(test data). Only in that case, we can comment if the model has a good generalized accuracy and it does not overfit. As we can see the test accuracy is similar to the training accuracy, and both of them are quite high. Therefore, we can say that the model captures the entire trend of the phenomena.

3.3 Fit again a GP using all covariates and comparison

```
#3.3 all covariates
GPfit_class_all = gausspr(fraud ~.,data = train)
```

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

```
#confusion matrix and accuracy
pred_all = predict(GPfit_class_all,test[,1:4])
conf_matrix_all = table(pred_all, test$fraud)
accur_all = length(which(pred_all == test$fraud))/nrow(test)

#print the confusion matrix
knitr::kable(x = conf_matrix_all,
             caption = "confusion matrix using all covariates")
```

Table 3: confusion matrix using all covariates

	0	1
0	205	0
1	1	166

```
#print a table with all accuracies
df = data.frame(data.frame("Accur_Train" = accur,
                           "Accur_Test" =  accur_test,
                           "Accur_AllCov" =  accur_all))
knitr::kable(x = df,
             caption = "Comparison of all accuracies")
```

Table 4: Comparison of all accuracies

Accur_Train	Accur_Test	Accur_AllCov
0.932	0.9354839	0.9973118

For the final question, we train the model using all 4 covariates. As we can see the classifier performs almost perfect. That means, that the other two variables give some important information to the classifier, in order to create more clear prediction boundaries. On the other hand, adding two more predictors make the model more complicated.