732A96: AML - Computer Lab 4

Julius Kittler (julki092)

October 12, 2019

Contents


```
# Set up general options

knitr::opts_chunk$set(echo = FALSE, warning = FALSE, message = FALSE,
                      fig.width=6, fig.height=5#, collapse=TRUE
                      )

set.seed(12345)
options(scipen=999)

# General libraries
library(ggplot2)
library(dplyr)

# Specific libaries
library(kernlab)
library(AtmRay)
```

```
# Auxilary functions
analyze_cm = function(cm, true){

  stopifnot(true %in% colnames(cm))
  levels = c(true, colnames(cm)[-which(colnames(cm) == true)]) # ORDER: 1; 0
  cm = as.data.frame(cm); colnames(cm)[1:2] = c("True", "Pred")
  N = sum(cm$Freq)
  Npos = sum(cm$Freq[which(cm$True == levels[1])])
  Nneg = sum(cm$Freq[which(cm$True == levels[2])])
  TP = sum(cm$Freq[which(cm$True == levels[1] & cm$Pred == levels[1])])
  TN = sum(cm$Freq[which(cm$True == levels[2] & cm$Pred == levels[2])])
  FP = sum(cm$Freq[which(cm$True == levels[2] & cm$Pred == levels[1])])
```

```r
  FN = sum(cm$Freq[which(cm$True == levels[1] & cm$Pred == levels[2])])
  return(data.frame(MCR = (FP+FN)/N, Accuracy = (TP + TN)/N,
                    Recall = TP/Npos, # recall = TPR = sensitivity,
                    Precision = TP/(TP + FP),
                    FPR = FP/Nneg, TNR = TN/Nneg)) # TNR = specificity
}

# cm = table(Y_true, Y_pred, dnn = c("True", "Predicted"))
# knitr::kable(analyze_cm(cm, true = "yes"))
```

---

1 Assignment 1: Implementing GP Regression

2.1. **Implementing GP Regression.** This first exercise will have you writing your own code for the Gaussian process regression model:

$$y = f(x) + \epsilon \text{ with } \epsilon \sim \mathcal{N}(0, \sigma_n^2) \text{ and } f \sim \mathcal{GP}(0, k(x, x'))$$

You must implement Algorithm 2.1 on page 19 of Rasmussen and Willams' book. The algorithm uses the Cholesky decomposition (`chol` in `R`) to attain numerical stability. Note that L in the algorithm is a lower triangular matrix, whereas the R function returns an upper triangular matrix. So, you need to transpose the output of the R function. In the algorithm, the notation $\boldsymbol{A}\backslash\boldsymbol{b}$ means the vector $\boldsymbol{x}$ that solves the equation $\boldsymbol{Ax} = \boldsymbol{b}$ (see p. xvii in the book). This is implemented in R with the help of the function `solve`.

Here is what you need to do:

(1) Write your own code for simulating from the posterior distribution of f using the squared exponential kernel. The function (name it `posteriorGP`) should return a vector with the posterior mean and variance of f, both evaluated at a set of x-values (X_*). You can assume that the prior mean of f is zero for all x. The function should have the following inputs:
- `X`: Vector of training inputs.
- `y`: Vector of training targets/outputs.
- `XStar`: Vector of inputs where the posterior distribution is evaluated, i.e. X_*.
- `hyperParam`: Vector with two elements, σ_f and ℓ.
- `sigmaNoise`: Noise standard deviation σ_n.

Hint: Write a separate function for the kernel (see the file `GaussianProcess.R` on the course web page).

(2) Now, let the prior hyperparameters be $\sigma_f = 1$ and $\ell = 0.3$. Update this prior with a single observation: $(x, y) = (0.4, 0.719)$. Assume that $\sigma_n = 0.1$. Plot the posterior mean of f over the interval $x \in [-1, 1]$. Plot also 95 % probability (pointwise) bands for f.

(3) Update your posterior from (2) with another observation: $(x, y) = (-0.6, -0.044)$. Plot the posterior mean of f over the interval $x \in [-1, 1]$. Plot also 95 % probability (pointwise) bands for f.

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

(4) Compute the posterior distribution of f using all the five data points in the table below (note that the two previous observations are included in the table). Plot the posterior mean of f over the interval $x \in [-1, 1]$. Plot also 95 % probability (pointwise) bands for f.

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

(5) Repeat (4), this time with hyperparameters $\sigma_f = 1$ and $\ell = 1$. Compare the results.

```r
# Utility function for posterior plots
plot_res = function(res, XStar, X, y){
```

```
  ul = res$mean + 1.96 * sqrt(res$variance)
  ll = res$mean - 1.96 * sqrt(res$variance)

  plot(XStar, res$mean, ylim = c(min(ll, y), max(ul, y)), type = "l",
       col = "forestgreen",
       main = "Mean (green) with CI (blue) and Data (black)")
  points(X, y, col = "black", pch = 16)
  lines(XStar, ul, col = "steelblue4")
  lines(XStar, ll, col = "steelblue4")

}
```

1.1 1: posteriorGP

Below, the function `posteriorGP` for simulating from the posterior distribution of `f` is implemented with the squared exponential kernel.

```
# Covariance function: Squared Exp. Kernel
my_kernel = function(x, y, sigmaF, l){

  K = matrix(NA, length(x), length(y))
  for (i in 1:length(x)){
    K[i, ] = sigmaF^2 * exp(-0.5*(x[i] - y)^2 / l^2)
  }
  return(K)

}
```

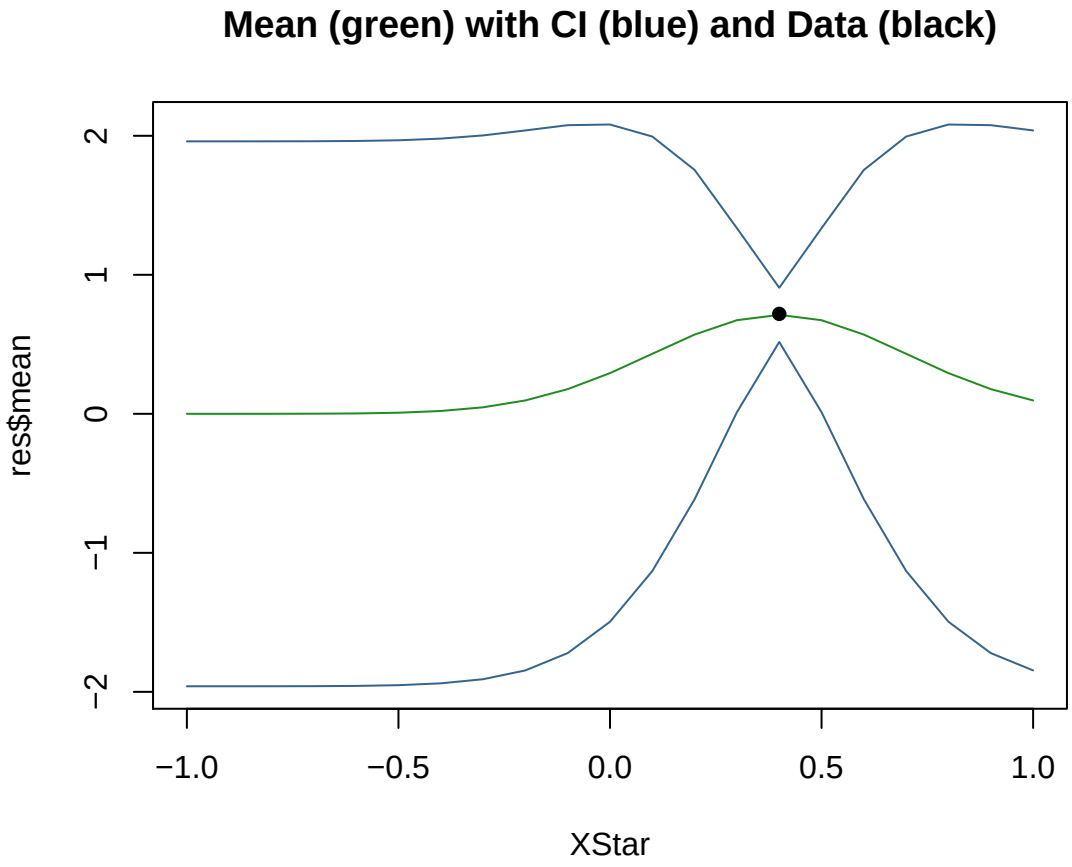
```
# Function to simulate from the posterior
posteriorGP = function(X, y, XStar, hyperParam, sigmaNoise){

  # ------------------------------------------------------------------------
  # Inputs:
  # X: Vector of training inputs.
  # y: Vector of training targets/outputs.
  # XStar: Vector of inputs where the posterior distribution is evaluated
  # hyperParam: Vector with two elements, sigma_f and l.
  # sigmaNoise: Noise standard deviation sigma_n
  #
  # Outputs:
  # mean: a vector with the posterior means for fStar
  # variance: a vector with the posterior variances of fStar
  # llh_marginal: a scalar with the marginal log likelihood
  # ------------------------------------------------------------------------

  # Initialize params
```

```
  n = length(y)
  sigmaf = hyperParam[1]
  l = hyperParam[2]
  K = my_kernel(X, X, sigmaf, l)
  KStar = my_kernel(X, XStar, sigmaf, l)

  # Conduct computations
  L = t(chol(K + sigmaNoise^2 * diag(nrow(K))))
  alpha = solve(t(L), solve(L, y))
  fStar = t(KStar) %*% alpha
  v = solve(L, KStar)
  V_fStar = my_kernel(XStar, XStar, sigmaf, l) - t(v) %*% v
  llh_marginal = -1/2 * t(y) %*% alpha - sum(log(L[row(L) == col(L)])) - n/2 * log(2*pi)

  # Return results
  return(list(mean = as.numeric(fStar),
              variance = V_fStar[row(V_fStar) == col(V_fStar)],
              llh_marginal = as.numeric(llh_marginal)))
}
```

1.2 2: Prior Update with 1 observation

Here, the prior hyperparameters are updated with 1 single observation. We can see that the posterior mean (green) goes through this observation as expected. Furthermore, the credible interval near the observation is much thinner obviously.

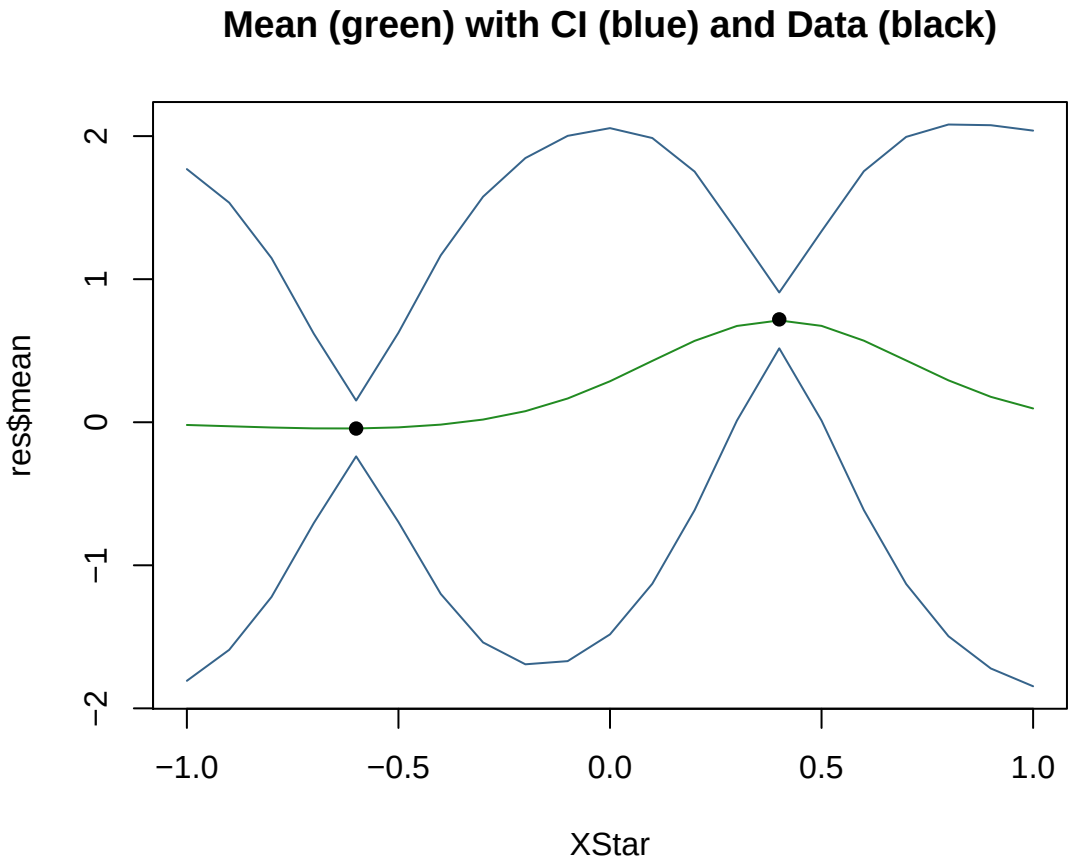
```
# Parmaters
hyperParam = c(1, 0.3) # sigmaf, l
sigmaNoise = 0.1

# Observation
X = 0.4
y = 0.719

# Test points
XStar = seq(-1, 1, 0.1)

# Compute posterior
res = posteriorGP(X, y, XStar, hyperParam, sigmaNoise)

# Visualization of Posterior
plot_res(res, XStar, X, y)
```



1.3 3: Prior Update with 2 observations

Here, the prior hyperparameters are updated with another observation. Again, the posterior mean (green) also goes through this second observation as expected. Furthermore, the credible interval near the new observation is much thinner obviously.

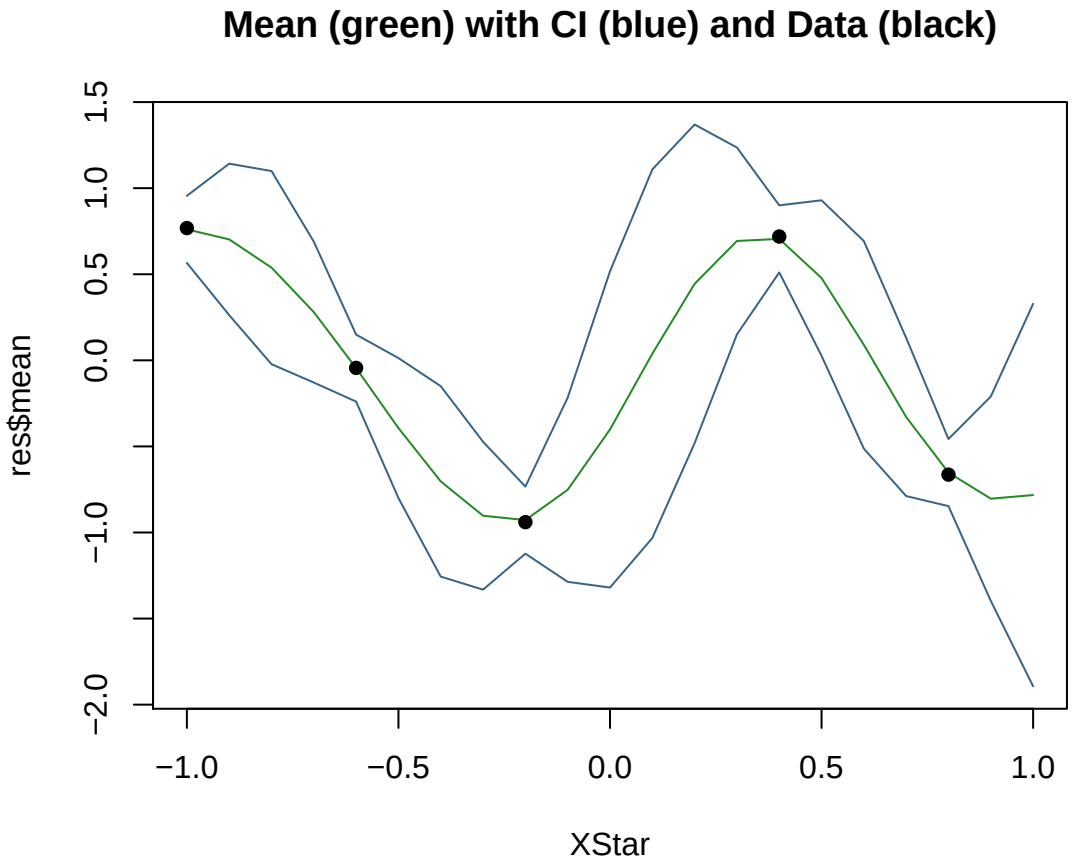
```
# Parmaters
hyperParam = c(1, 0.3) # sigmaf, l
sigmaNoise = 0.1

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

# Test points
XStar = seq(-1, 1, 0.1)

# Compute posterior
res = posteriorGP(X, y, XStar, hyperParam, sigmaNoise)

# Visualization of Posterior
plot_res(res, XStar, X, y)
```



1.4 4: Prior Update with 5 observations

With five observations, the posterior mean clearly takes a smooth and non-linear shape. One can see that the posterior mean also seems reasonable for intervals betwen the observed data points.

```
# Parmaters
hyperParam = c(1, 0.3) # sigmaf, l
sigmaNoise = 0.1

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

# Test points
XStar = seq(-1, 1, 0.1)

# Compute posterior
res = posteriorGP(X, y, XStar, hyperParam, sigmaNoise)

# Visualization of Posterior
plot_res(res, XStar, X, y)
```



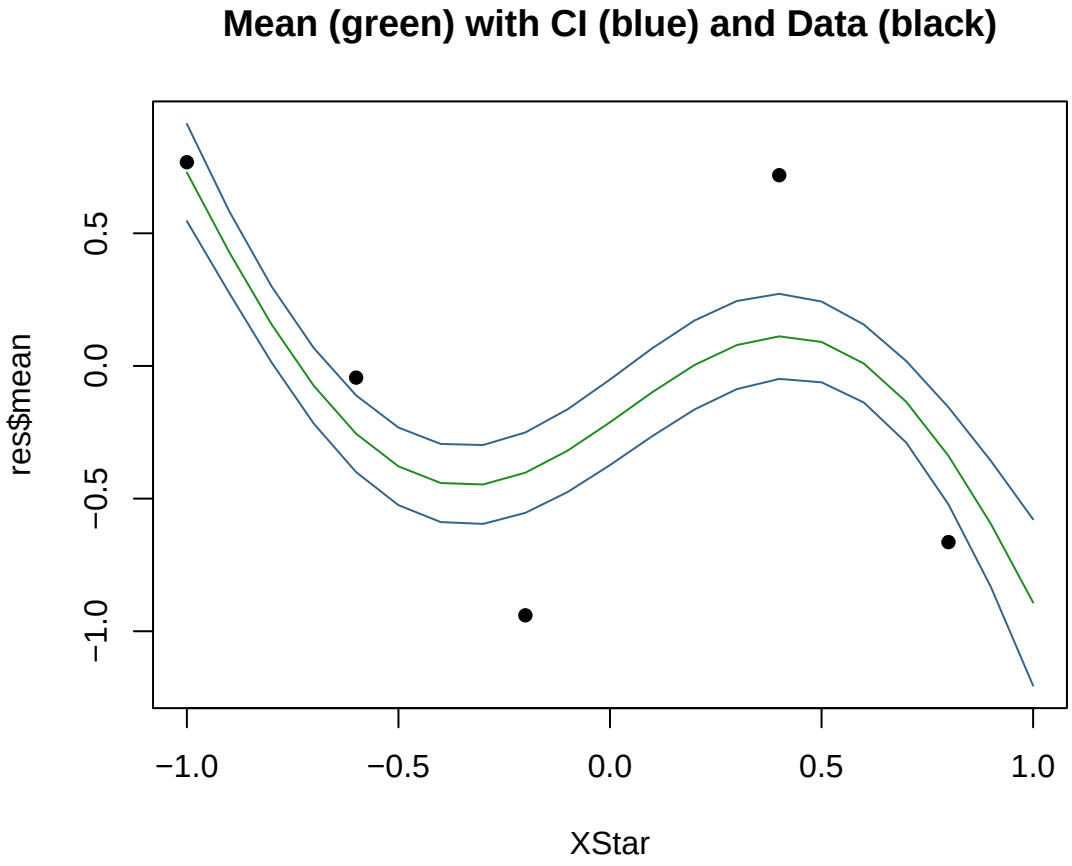
1.5 5: Prior Update with 5 observations (large l)

Here, we increase the parameter `l` from 0.3 to 1. We can see that the mean is much smoother than in the previous task (and correspondingly the credible bands as well). This is expected since the parameter `l` regulates the smoothness of the function, where a larger `l` corresponds to more smoothness (i.e. less flexibility).

It is visible that the posterior mean does not go through all data points anymore. Since we would might this to be the case, we could argue that the value `l=1` is too large. A smaller value might be better to allow for some more flexibility.

```
# Parmaters
hyperParam = c(1, 1) # sigmaf, l
sigmaNoise = 0.1

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

# Test points
XStar = seq(-1, 1, 0.1)

# Compute posterior
```

```
res = posteriorGP(X, y, XStar, hyperParam, sigmaNoise)

# Visualization of Posterior
plot_res(res, XStar, X, y)
```



2 Assignment 2: GP Regression with kernlab

2.2. **GP Regression with `kernlab`.** In this exercise, you will work with the daily mean temperature in Stockholm (Tullinge) during the period January 1, 2010 - December 31, 2015. We have removed the leap year day February 29, 2012 to make things simpler. You can read the dataset with the command:

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

Create the variable `time` which records the day number since the start of the dataset (i.e., `time`= 1, 2, ..., 365 × 6 = 2190). Also, create the variable `day` that records the day number since the start of each year (i.e., `day`= 1, 2, ..., 365, 1, 2, ..., 365). Estimating a GP on 2190 observations can take some time on slower computers, so let us subsample the data and use only every fifth observation. This means that your time and day variables are now `time`= 1, 6, 11, ..., 2186 and `day`= 1, 6, 11, ..., 361, 1, 6, 11, ..., 361.

(1) Familiarize yourself with the functions `gausspr` and `kernelMatrix` in `kernlab`. Do `?gausspr` and read the input arguments and the output. Also, go through the file `KernLabDemo.R` available on the course website. You will need to understand it. Now, define your own square exponential kernel function (with parameters ℓ (`ell`) and σ_f (`sigmaf`)), evaluate it in the point $x = 1, x' = 2$, and use the `kernelMatrix` function to compute the covariance matrix $K(X, X_*)$ for the input vectors $X = (1,3,4)^T$ and $X_* = (2,3,4)^T$.

(2) Consider first the following model:

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

Let σ_n^2 be the residual variance from a simple quadratic regression fit (using the `lm` function in R). Estimate the above Gaussian process regression model using the squared exponential function from (1) with $\sigma_f = 20$ and $\ell = 0.2$. Use the `predict` function in R to compute the posterior mean at every data point in the training dataset. Make a scatterplot of the data and superimpose the posterior mean of f as a curve (use `type="l"` in the plot function). Play around with different values on σ_f and ℓ (no need to write this in the report though).

(3) `kernlab` can compute the posterior variance of f, but it seems to be a bug in the code. So, do your own computations for the posterior variance of f and plot the 95 % probability (pointwise) bands for f. Superimpose these bands on the figure with the posterior mean that you obtained in (2).

Hint: Note that Algorithm 2.1 on page 19 of Rasmussen and Willams' book already does the calculations required. Note also that `kernlab` scales the data by default to have zero mean and standard deviation one. So, the output of your implementation of Algorithm 2.1 will not coincide with the output of `kernlab` unless you scale the data first. For this, you may want to use the R function `scale`.

(4) Consider now the following model:

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

Estimate the model using the squared exponential function with $\sigma_f = 20$ and $\ell = 0.2$. Superimpose the posterior mean from this model on the posterior mean from the model in (2). Note that this plot should also have the time variable on the horizontal axis. Compare the results of both models. What are the pros and cons of each model?

(5) Finally, implement a generalization of the periodic kernel given in the lectures:

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

Note that we have two different length scales here, and ℓ_2 controls the correlation between the same day in different years. Estimate the GP model using the time variable with this kernel and hyperparameters $\sigma_f = 20$, $\ell_1 = 1$, $\ell_2 = 10$ and $d = 365/\texttt{sd(time)}$. The reason for the rather strange period here is that `kernlab` standardizes the inputs to have standard deviation of 1. Compare the fit to the previous two models (with $\sigma_f = 20$ and $\ell = 0.2$). Discuss the results.

```
# Data import
df = read.csv2("input/TempTullinge.csv")

# Add variables
df$time = 1:nrow(df)
df$day = rep(1:365, 6)
df$temp = as.numeric(as.character(df$temp))

# Subset to every 5th obs.
df = df[seq(1, nrow(df), 5), ]

# Show data set
head(df)
```

```
##        date  temp time day
## 1  01/01/10  -8.6    1   1
## 6  06/01/10 -15.4    6   6
## 11 11/01/10 -11.4   11  11
## 16 16/01/10  -2.5   16  16
```

```
## 21 21/01/10  -2.5   21  21
## 26 26/01/10 -12.9   26  26
```

```
tail(df)
```

```
##          date temp time day
## 2161 02/12/15  0.1 2161 336
## 2166 07/12/15  6.3 2166 341
## 2171 12/12/15  0.1 2171 346
## 2176 17/12/15  2.4 2176 351
## 2181 22/12/15  5.4 2181 356
## 2186 27/12/15 -4.2 2186 361
```

2.1 1: Kernel Function and Kernel Matrix

We are going to use the following functions:

```
?kernlab::gausspr
?kernlab::kernelMatrix
```

Square Exponential Kernel Function

Instead of using `rfbdot` (as `kernel` parameter for `gausspr`), we are supposed to define our own square exponential function. The square exponential function was defined as follows in the lecture:

$$k(\mathbf{x}, \mathbf{x}') = \text{cov}(f(\mathbf{x}), f(\mathbf{x}')) = \sigma_f^2 \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2l^2}\right)$$

Note that the kernel function itself does not return a matrix. Instead, it returns 1 scalar given 2 scalars or vectors. This returned scalar is later on used as variance or covariance in the covariance matrix.

```
# Define closure (needed for gausspr)
SquaredExpKernel = function(sigmaF, l){
  kernel  = function(x, y){
    r = sqrt(t(x-y) %*% (x-y)) # euclidean distance
    K = sigmaF^2 * exp(-0.5 * r^2 / l^2)
    return(K)
  }
  class(kernel) = "kernel"
  return(kernel)
}
```

Here, we conduct a quick comparison of the results we get when applying the own implementation of the squared exponential kernel with R's equivalent implementation `rbfdot`. Ideally, we expect both to give the same results, which is indeed the case.

```
# Take hyperparams from KernLabDemo.R
l = 1
```

```
sigmaF = 1

# Modify rbfdot kernel with hyperparameter
SE_kernel_rbfdot = rbfdot(sigma = 1/(2*l^2))
SE_kernel = SquaredExpKernel(sigmaF = 1, l = 1)

# Example
SE_kernel(x = 1, y = 2)
```

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

```
SE_kernel_rbfdot(1, 2)
```

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

Covariance matrix with `kernelMatrix`

Here, we create a covariance matrix with the square exponential Kernel for two input vectors `X` and `XStar`. Again, we can see that the resulting covariance matrix is the same for `rbfdot` and for our implementation.

```
X = c(1, 3, 4)
XStar = c(2, 3, 4)
kernelMatrix(kernel = SE_kernel, x = X, y = XStar)
```

```
## 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
```

```
kernelMatrix(kernel = SE_kernel_rbfdot, x = X, y = XStar)
```

```
## 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 2: GP Regression and Prediction (based on `time`)

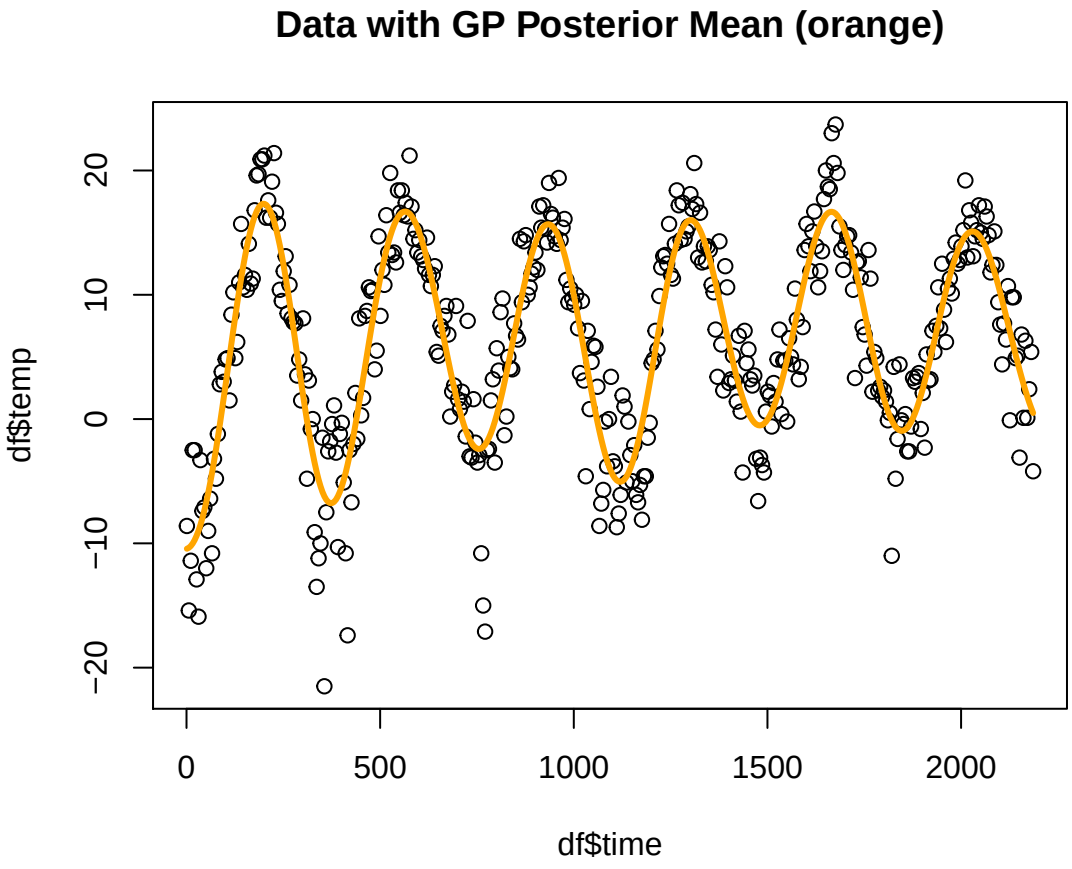
Here, we estimate the following Gaussian process regression model:

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

Playing around with the parameters `sigmaF` and `l` led to the following insights:

- Increasing `sigmaF` (e.g. from 20 to 200) does not noticeably change the results. I.e. the predicted means look very similar.
- Decreasing `sigmaF` (e.g. from 20 to 2) decreases the variance of the means s.t. the mean curve does not fit the data anymore. The mean will go up and down when the temperature in the data goes up and down respectively. However, the mean won't go up and down enough to get close to the data points.
- Decreasing `l` (e.g.from 0.2 to 0.02) lets the mean oscillate slightly (unnecessarily). It will seem as if noise was added at some time points. Overall, the mean values still fit the data well.
- Increasing `l` (e.g. from 0.2 to 2) results in mean values that resemble almost a linear line. I.e. the mean values are too smooth and do not fit the data well.

```r
# Update SE_Kernel with given parameters (for later)
SE_kernel = SquaredExpKernel(sigmaF = 20, l = 0.2)

# Find sigmaNoise
res = lm(temp ~ time + I(time^2), df)
sigmaNoise = sd(res$residuals)

# Fit the GP with built in Square expontial kernel (called rbfdot in kernlab)
res = gausspr(temp ~ time, data = df, kernel = SquaredExpKernel,
              kpar = list(sigmaF = 20, l = 0.2), var = sigmaNoise^2)

plot(df$time, df$temp, main = "Data with GP Posterior Mean (orange)")
meanPred = predict(res, df) # Predicting the training data. To plot the fit.
lines(df$time, meanPred, col="orange", lwd = 3)
```

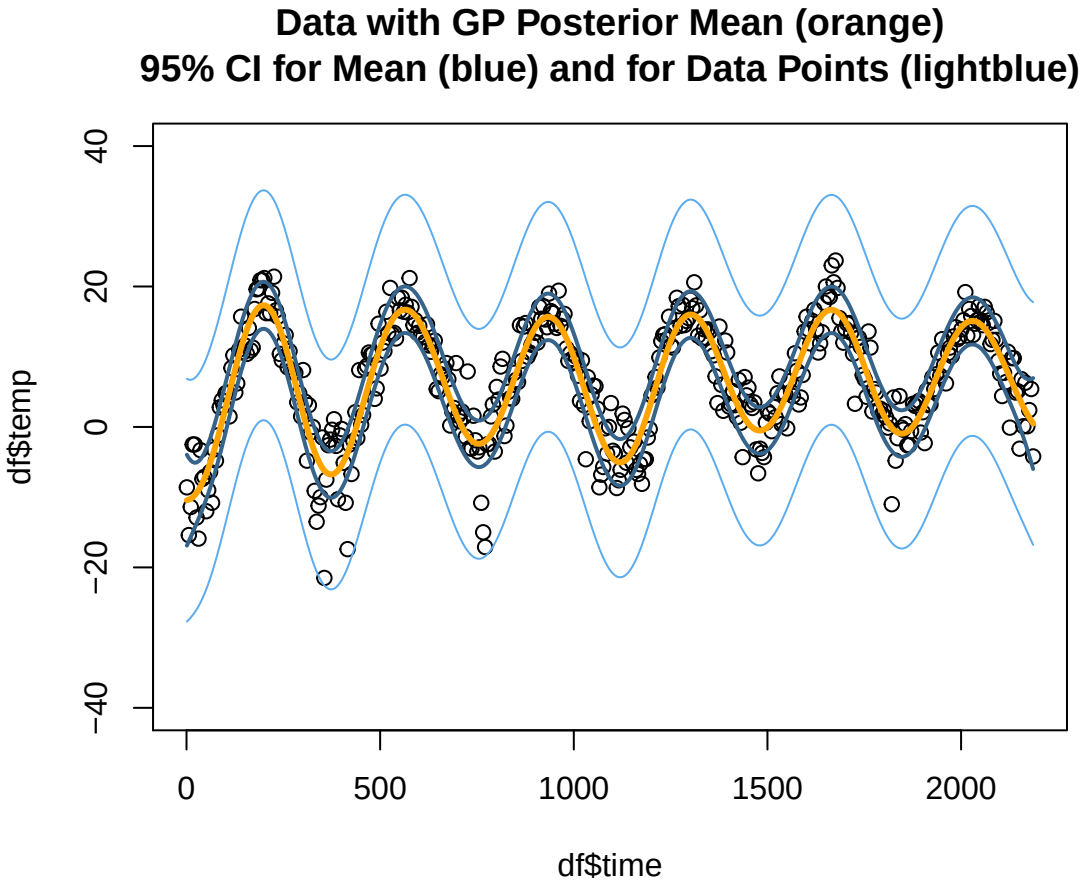


2.3 3: Adding Credible Intervals for Mean and Points

Here, we add credible intervals for the posterior means (dark blue) and the data points (light blue).

```
# Find sigmaNoise
res = lm(temp ~ time + I(time^2), df)
sigmaNoise = sd(res$residuals)

# Fit the GP with built in Square expontial kernel (called rbfdot in kernlab)
res = gausspr(temp ~ time, data = df, kernel = SquaredExpKernel,
                kpar = list(sigmaF = 20, l = 0.2), var = sigmaNoise^2)

plot(df$time, df$temp, ylim = c(-40, 40),
     main = "Data with GP Posterior Mean (orange)
95% CI for Mean (blue) and for Data Points (lightblue)")
meanPred = predict(res, df) # Predicting the training data. To plot the fit.
lines(df$time, meanPred, col="orange", lwd = 3)

# Probability and prediction interval implementation.
# Note: Scaling required since mean was automatically scaled by kernlab as well.
x = scale(df$time) #
xs = scale(df$time) # XStar
n = length(x)
```

```r
Kss = kernelMatrix(kernel = SE_kernel, x = xs, y = xs)
Kxx = kernelMatrix(kernel = SE_kernel, x = x, y = x)
Kxs = kernelMatrix(kernel = SE_kernel, x = x, y = xs)
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Covariance matrix of fStar

# Probability intervals for fStar
lines(df$time, meanPred - 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)
lines(df$time, meanPred + 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)

# Prediction intervals for yStar
lines(df$time, meanPred - 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")
lines(df$time, meanPred + 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")
```



2.4 4: GP Regression and Prediction (based on day)

Here, we repeat the task from step 2. But this time, however, we use `day` as predictive variable (instead of the `time`). We still plot the `time` on the x-axis so that we can compare the results.

We can see that the resulting means and credible bands are not as smooth as before. Moreover, there are more points that lie outside the lightblue credible bands (that are meant for the points). Nevertheless, we still have to say that the results are quite good after all. We can conclude that the temperature can largely be predicted based on the day in the year. The exact date may not be

necessary for the temperature prediction because there is e.g. no strong trend across the years.

Pros

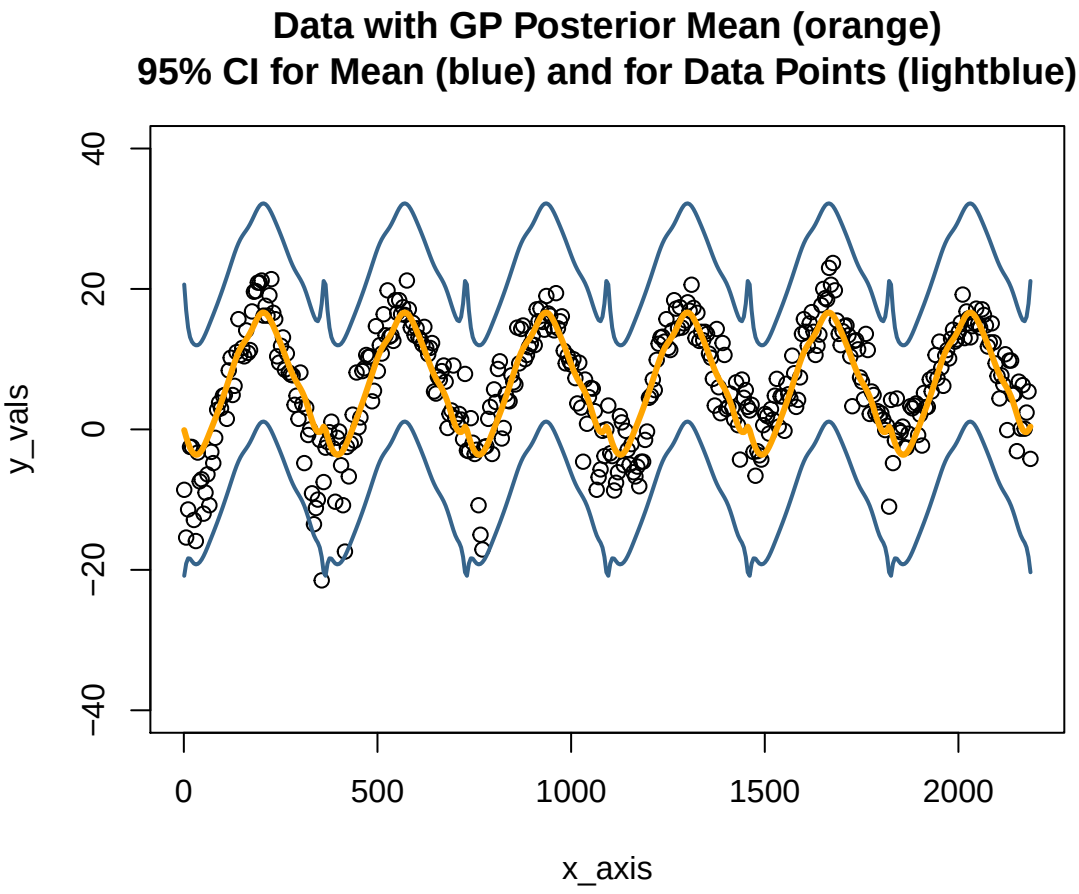
- With only `day` as covariate, the model has higher simplicity.
- With only `day` as covariate, the model might generalize better to the future. This is so because the model is more general, only using the day in the year instead of the exact date.

Cons

- With only `day` as covariate, the differences between years may not be modelled/predicted appropriately. I.e. if there was a trend across years, the `day` covariate would not be sufficient.

```
# Set parameters ----------------------------------------------------------

# SigmaNoise
res = lm(temp ~ day + I(day^2), df)
sigmaNoise = sd(res$residuals)
sigmaNoise = 50
# Hyperparameters
sigmaF = 20
l = 0.2

# Kernel
my_kernel = SquaredExpKernel

# Formula
my_formula = as.formula("temp ~ day")

# Names
x_axis_name = "time"
y_name = "temp"
x_name = "day"

# Fit Model ---------------------------------------------------------------

# Update kernel with given parameters
my_kernel_params = my_kernel(sigmaF = sigmaF, l = l)

x_axis = df[[x_axis_name]]
x_vals = df[[x_name]]
y_vals = df[[y_name]]

# Fit the GP with built in Square expontial kernel (called rbfdot in kernlab)
res = gausspr(my_formula, data = df, kernel = my_kernel,
              kpar = list(sigmaF = sigmaF, l = l), var = sigmaNoise^2)

plot(x_axis, y_vals, ylim = c(-40, 40),
     main = "Data with GP Posterior Mean (orange)
```

```
95% CI for Mean (blue) and for Data Points (lightblue)")
meanPred = predict(res, df) # Predicting the training data. To plot the fit.
lines(x_axis, meanPred, col="orange", lwd = 3)

# Probability and prediction interval implementation.
# Note: Scaling required since mean was automatically scaled by kernlab as well.
x = scale(x_vals) #
xs = scale(x_vals) # XStar
n = length(x)
Kss = kernelMatrix(kernel = my_kernel_params, x = xs, y = xs)
Kxx = kernelMatrix(kernel = my_kernel_params, x = x, y = x)
Kxs = kernelMatrix(kernel = my_kernel_params, x = x, y = xs)
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Covariance matrix of fStar

# Probability and prediction interval implementation.
# Note: Scaling required since mean was automatically scaled by kernlab as well.
x = scale(x_vals) #
xs = scale(x_vals) # XStar
n = length(x)
Kss = kernelMatrix(kernel = my_kernel_params, x = xs, y = xs)
Kxx = kernelMatrix(kernel = my_kernel_params, x = x, y = x)
Kxs = kernelMatrix(kernel = my_kernel_params, x = x, y = xs)
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Cov. matrix of fStar

# Probability intervals for fStar
lines(x_axis, meanPred - 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)
lines(x_axis, meanPred + 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)

# Prediction intervals for yStar
lines(x_axis, meanPred - 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")
lines(x_axis, meanPred + 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")
```



2.5 5: Periodic Kernel

Here, we implement a generalization of the periodic kernel (see screenshot above for the formula) and fit another Gaussian process regression model, this time again using the `time` variable.

The resulting posterior mean and credible bands seem to resemble the results from the model in 1.2 (with `time`) very closely. In other words, the result is better than the result from 1.3 (with `day`). This is so, because there are no unnecessary oscillations visible and the credible intervals also seem to include all data points.

Considering the 2 used kernels (only with `time`), we could say that the...

squared exponential kernel

- is better if we prefer simplicity (and don't want to set additional parameters `d` and `l2`)
- is better if there is no actual periodicity

periodic kernel

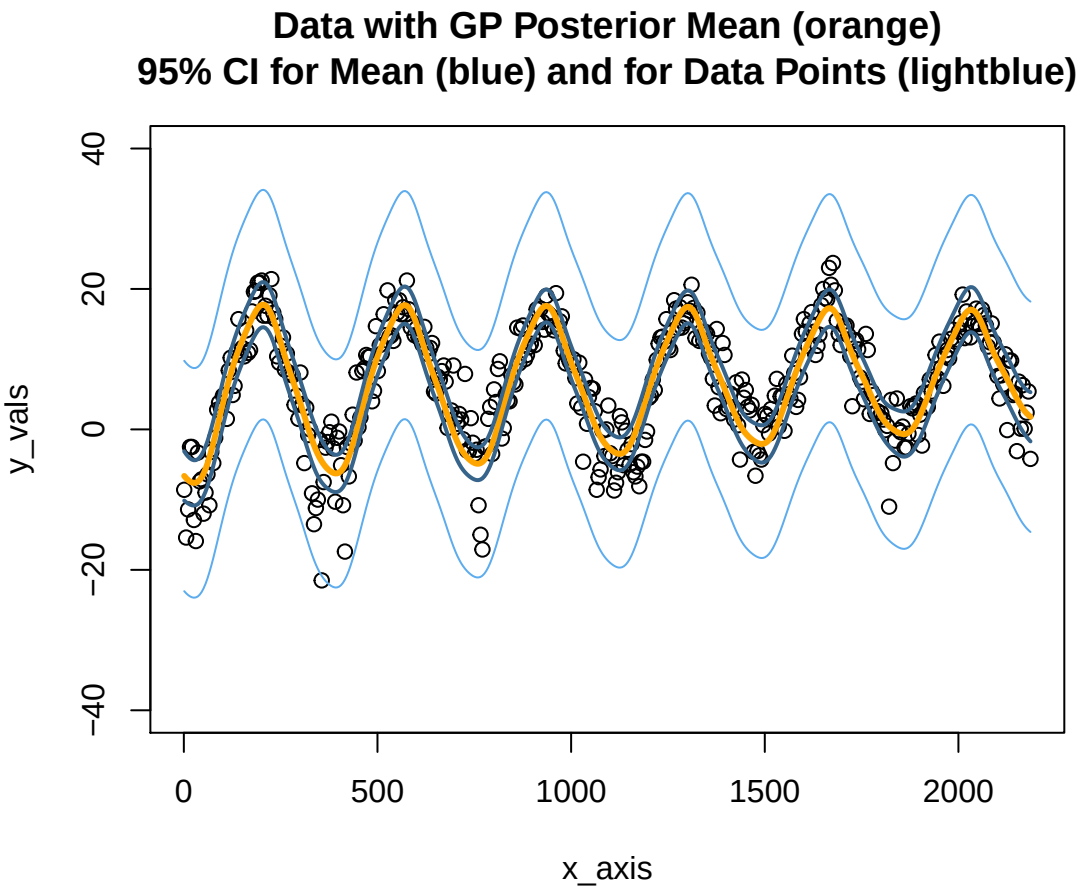
- is better if we prefer more control over certain parameters e.g. `d` and `l2`
- is appropriate if there is periodicity

```
# Define closure (needed for gausspr)
PeriodicKernel = function(sigmaF, l1, l2, d){
  kernel  = function(x, y){
```

```
    r = sqrt(t(x-y) %*% (x-y)) # euclidean distance
    K = sigmaF^2 * exp(-2 * sin(pi*r/d)^2 / l1^2) *
                   exp(-0.5 * r^2 / l2^2)
    return(K)
  }
  class(kernel) = "kernel"
  return(kernel)
}
```

```
# Set parameters ----------------------------------------------------------

# SigmaNoise
res = lm(temp ~ time + I(time^2), df)
sigmaNoise = sd(res$residuals)

# Hyperparameters
sigmaF = 20
l1 = 1
l2 = 10
d = 365/sd(df$time)

# Kernel
my_kernel = PeriodicKernel

# Formula
my_formula = as.formula("temp ~ time")

# Names
x_axis_name = "time"
y_name = "temp"
x_name = "time"

# Fit Model ---------------------------------------------------------------

# Update kernel with given parameters
my_kernel_params = my_kernel(sigmaF = sigmaF, l1 = l1, l2 = l2, d = d)

x_axis = df[[x_axis_name]]
x_vals = df[[x_name]]
y_vals = df[[y_name]]

# Fit the GP with built in Square expontial kernel (called rbfdot in kernlab)
res = gausspr(my_formula, data = df, kernel = my_kernel,
              kpar = list(sigmaF = sigmaF, l1 = l1, l2 = l2, d = d),
              var = sigmaNoise^2)
```

```r
plot(x_axis, y_vals, ylim = c(-40, 40),
     main = "Data with GP Posterior Mean (orange)
95% CI for Mean (blue) and for Data Points (lightblue)")
meanPred = predict(res, df) # Predicting the training data. To plot the fit.
lines(x_axis, meanPred, col="orange", lwd = 3)

# Probability and prediction interval implementation.
# Note: Scaling required since mean was automatically scaled by kernlab as well.
x = scale(x_vals) #
xs = scale(x_vals) # XStar
n = length(x)
Kss = kernelMatrix(kernel = my_kernel_params, x = xs, y = xs)
Kxx = kernelMatrix(kernel = my_kernel_params, x = x, y = x)
Kxs = kernelMatrix(kernel = my_kernel_params, x = x, y = xs)
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Covariance matrix of fStar

# Probability and prediction interval implementation.
# Note: Scaling required since mean was automatically scaled by kernlab as well.
x = scale(x_vals) #
xs = scale(x_vals) # XStar
n = length(x)
Kss = kernelMatrix(kernel = my_kernel_params, x = xs, y = xs)
Kxx = kernelMatrix(kernel = my_kernel_params, x = x, y = x)
Kxs = kernelMatrix(kernel = my_kernel_params, x = x, y = xs)
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Cov. matrix of fStar

# Probability intervals for fStar
lines(x_axis, meanPred - 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)
lines(x_axis, meanPred + 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)

# Prediction intervals for yStar
lines(x_axis, meanPred - 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")
lines(x_axis, meanPred + 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")
```


Data with GP Posterior Mean (orange)
95% CI for Mean (blue) and for Data Points (lightblue)
y_vals
40
20
0
−20
−40
0
500
1000
1500
2000
x_axis

3 Assignment 3: GP Classification with kernlab

2.3. **GP Classification with `kernlab`.** Download the banknote fraud 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])
```

You can read about this dataset here. Choose 1000 observations as training data using the following command (i.e., use the vector `SelectTraining` to subset the training observations):

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

(1) Use the R package `kernlab` to fit a Gaussian process classification model for fraud on the training data. Use the default kernel and hyperparameters. Start using only the covariates `varWave` and `skewWave` in the model. Plot contours of the prediction probabilities over a suitable grid of values for `varWave` and `skewWave`. Overlay the training data for fraud = 1 (as blue points) and fraud = 0 (as red points). You can reuse code from the file `KernLabDemo.R` available on the course website. Compute the confusion matrix for the classifier and its accuracy.

(2) Using the estimated model from (1), make predictions for the test set. Compute the accuracy.

(3) Train a model using all four covariates. Make predictions on the test set and compare the accuracy to the model with only two covariates.

First, the data is imported and split into training and test data.

```
# Data import
df = read.csv('input/banknoteFraud.csv', header = FALSE)
names(df) = c("varWave","skewWave","kurtWave","entropyWave","fraud")
df[,5] = as.factor(df[,5])

# Split data
set.seed(111)
idx_tr = sample(1:dim(df)[1], size = 1000, replace = FALSE)
df_tr = df[idx_tr, ]
df_te = df[-idx_tr, ]

head(df_tr)
```

```
##     varWave skewWave  kurtWave entropyWave fraud
## 814 -2.1333   1.5685 -0.084261    -1.74530     1
```

```
## 997 -2.3142   2.0838 -0.468130    -1.67670     1
## 508  4.6014   5.6264 -2.123500     0.19309     0
## 705  3.7022   6.9942 -1.851100    -0.12889     0
## 517 -2.3983  12.6060  2.946400    -5.78880     0
## 572  2.2517  -5.1422  4.291600    -1.24870     0
```

3.1 1: GP Classification (2 covariates)

GP Classification is performed with the 2 covariates `varWave` and `skewWave`. The accuracy on the training data and the corresponding confusion matrix is printed out below. We can conclude that the fit is quite good.

```
# Set parameters, fit model
classname = "fraud"
classvals = c(0, 1)
varnames = c("varWave", "skewWave")
res = gausspr(fraud ~ varWave + skewWave, data = df_tr)
```

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

```
# Confusion matrix
cm = table(predict(res, df_tr[,colnames(df_tr) %in% varnames]), # predicted
      df_tr[["fraud"]], dnn = c("pred", "true")) # true
cm
```

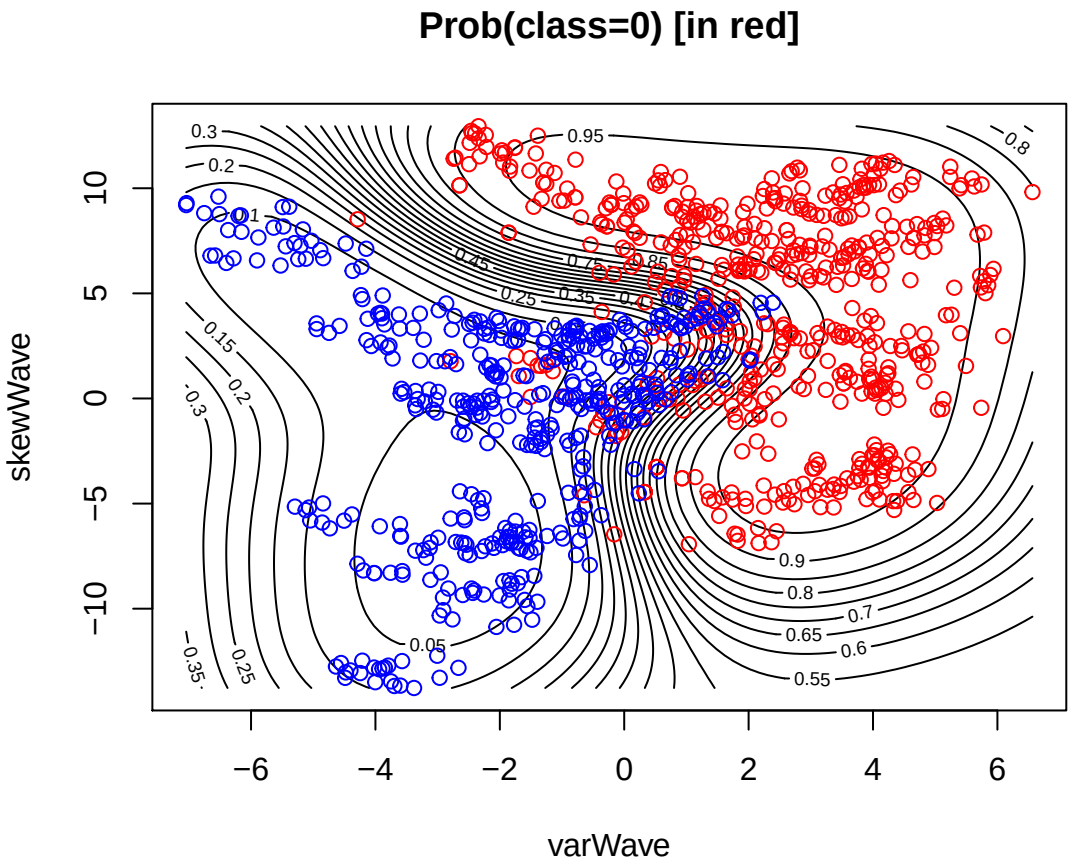
```
##      true
## pred   0   1
##    0 512  24
##    1  44 420
```

```
knitr::kable(analyze_cm(cm, true = "1"))
```

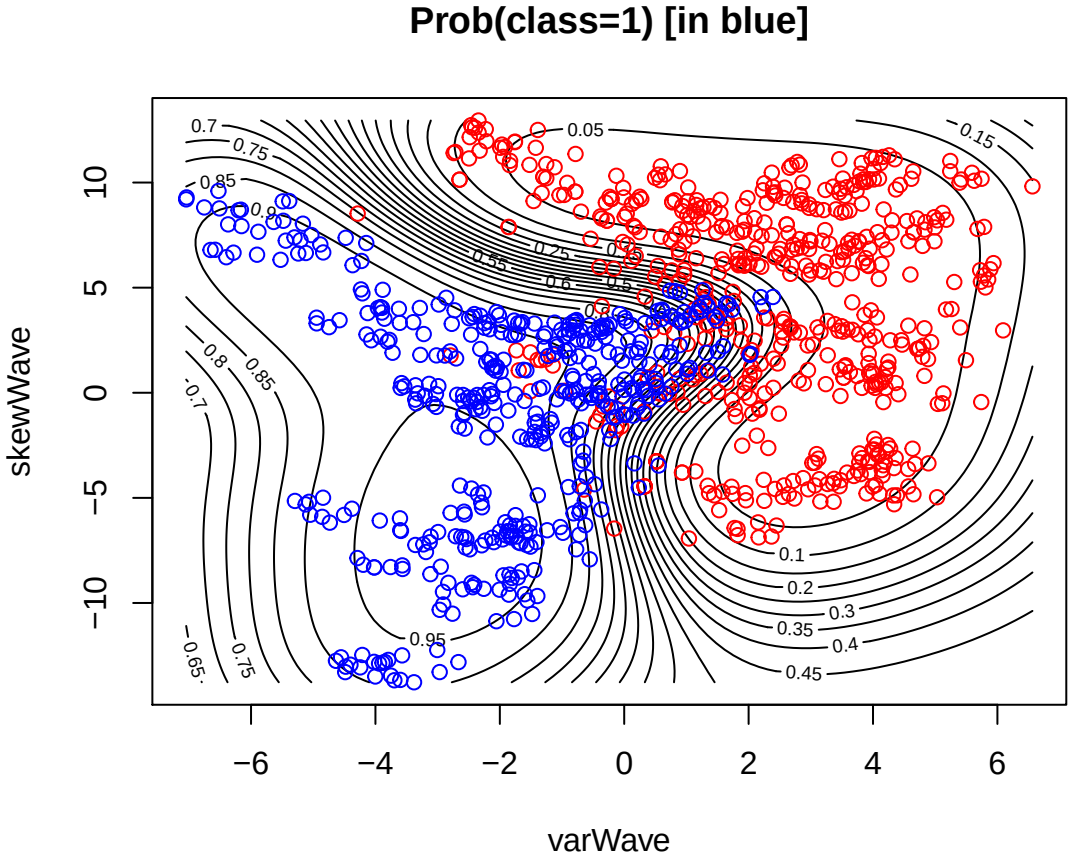
MCR	Accuracy	Recall	Precision	FPR	TNR
0.068	0.932	0.9051724	0.9459459	0.0447761	0.9552239

```
# Create plot
probs = predict(res, df_tr[,colnames(df_tr) %in% varnames], type="probabilities")
x1 = seq(min(df_tr[[varnames[1]]]),max(df_tr[[varnames[1]]]),length=100)
x2 = seq(min(df_tr[[varnames[2]]]),max(df_tr[[varnames[2]]]),length=100)
gridPoints = meshgrid(x1, x2)
gridPoints = cbind(c(gridPoints$x), c(gridPoints$y))
gridPoints = data.frame(gridPoints)
names(gridPoints) = varnames
probs = predict(res, gridPoints, type="probabilities")

# Plotting for Prob (classval 1). Note: Add more points for more classes
contour(x1,x2,matrix(probs[,colnames(probs) == classvals[1]],100,byrow = TRUE),
```

```
        20, xlab = varnames[1], ylab = varnames[2],
        main = paste0('Prob(class=', classvals[1],') [in red]'))
points(df_tr[[varnames[1]]][df_tr[[classname]]==classvals[1]],
       df_tr[[varnames[2]]][df_tr[[classname]]==classvals[1]], col="red")
points(df_tr[[varnames[1]]][df_tr[[classname]]==classvals[2]],
       df_tr[[varnames[2]]][df_tr[[classname]]==classvals[2]], col="blue")
```




```
# Plotting for Prob (classval 2). Note: Add more points for more classes
contour(x1,x2,matrix(probs[,colnames(probs) == classvals[2]],100,byrow = TRUE),
        20, xlab = varnames[1], ylab = varnames[2],
        main = paste0('Prob(class=', classvals[2],') [in blue]'))
points(df_tr[[varnames[1]]][df_tr[[classname]]==classvals[1]],
       df_tr[[varnames[2]]][df_tr[[classname]]==classvals[1]], col="red")
points(df_tr[[varnames[1]]][df_tr[[classname]]==classvals[2]],
       df_tr[[varnames[2]]][df_tr[[classname]]==classvals[2]], col="blue")
```



3.2 2: Evaluation on Test Data (2 covariates)

Evaluated on the test data, we can see that the accuracy is even slightly better than for the training data. Hence, we can conclude that the model generalizes very well to new data from the same origin.

```
# Confusion matrix
cm = table(predict(res, df_te[,colnames(df_te) %in% varnames]), # predicted
      df_te[["fraud"]], dnn = c("pred", "true")) # true
cm
```

```
##      true
## pred   0   1
##    0 191   9
##    1  15 157
```

```
knitr::kable(analyze_cm(cm, true = "1"))
```

MCR	Accuracy	Recall	Precision	FPR	TNR
0.0645161	0.9354839	0.9127907	0.9457831	0.045	0.955

3.3 3: GP Classification and Evaluation (4 covariates)

Here, we use all 4 covariates instead of only 2. We can see that the performance on the test data improves noticeably by approx. 6 percent points. Obviously, the additional 2 covariates also provide useful information for classification.

```
# Set parameters, fit model
classname = "fraud"
classvals = c(0, 1)
varnames = c("varWave", "skewWave", "kurtWave", "entropyWave")
res = gausspr(fraud ~ varWave + skewWave + kurtWave + entropyWave, data = df_tr)
```

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

```
# Confusion matrix
cm = table(predict(res, df_te[,colnames(df_te) %in% varnames]), # predicted
      df_te[["fraud"]], dnn = c("pred", "true")) # true
cm
```

```
##      true
## pred   0   1
##    0 205   0
##    1   1 166
```

```
knitr::kable(analyze_cm(cm, true = "1"))
```

MCR	Accuracy	Recall	Precision	FPR	TNR
0.0026882	0.9973118	0.994012	1	0	1

4 Appendix

```
# Set up general options

knitr::opts_chunk$set(echo = FALSE, warning = FALSE, message = FALSE,
                      fig.width=6, fig.height=5#, collapse=TRUE
                      )

set.seed(12345)
options(scipen=999)

# General libraries
library(ggplot2)
library(dplyr)

# Specific libaries
library(kernlab)
```

```r
library(AtmRay)


# Auxilary functions
analyze_cm = function(cm, true){

  stopifnot(true %in% colnames(cm))
  levels = c(true, colnames(cm)[-which(colnames(cm) == true)]) # ORDER: 1; 0
  cm = as.data.frame(cm); colnames(cm)[1:2] = c("True", "Pred")
  N = sum(cm$Freq)
  Npos = sum(cm$Freq[which(cm$True == levels[1])])
  Nneg = sum(cm$Freq[which(cm$True == levels[2])])
  TP = sum(cm$Freq[which(cm$True == levels[1] & cm$Pred == levels[1])])
  TN = sum(cm$Freq[which(cm$True == levels[2] & cm$Pred == levels[2])])
  FP = sum(cm$Freq[which(cm$True == levels[2] & cm$Pred == levels[1])])

  FN = sum(cm$Freq[which(cm$True == levels[1] & cm$Pred == levels[2])])
  return(data.frame(MCR = (FP+FN)/N, Accuracy = (TP + TN)/N,
                    Recall = TP/Npos, # recall = TPR = sensitivity,
                    Precision = TP/(TP + FP),
                    FPR = FP/Nneg, TNR = TN/Nneg)) # TNR = specificity
}

# cm = table(Y_true, Y_pred, dnn = c("True", "Predicted"))
# knitr::kable(analyze_cm(cm, true = "yes"))


# Rstudio guide

## Setup
#```{r setup, include=TRUE, results='hide', message=FALSE, warning=FALSE}
#```

## Appendix
# ```{r, ref.label = knitr::all_labels(), echo = TRUE, eval = FALSE}
#```

## Add image
#```{r label, out.height = "400px", out.width = "800px"}
#knitr::include_graphics("image.png")
#```

## Add image
#![Caption for the picture.](/path/to/image.png)
```

```r
# ------------------------------------------------------------------------------
# Assignment 1
# ------------------------------------------------------------------------------

knitr::include_graphics("images/Q1.png")

# Utility function for posterior plots
plot_res = function(res, XStar, X, y){

  ul = res$mean + 1.96 * sqrt(res$variance)
  ll = res$mean - 1.96 * sqrt(res$variance)

  plot(XStar, res$mean, ylim = c(min(ll, y), max(ul, y)), type = "l",
       col = "forestgreen",
       main = "Mean (green) with CI (blue) and Data (black)")
  points(X, y, col = "black", pch = 16)
  lines(XStar, ul, col = "steelblue4")
  lines(XStar, ll, col = "steelblue4")

}


# Covariance function: Squared Exp. Kernel
my_kernel = function(x, y, sigmaF, l){

  K = matrix(NA, length(x), length(y))
  for (i in 1:length(x)){
    K[i, ] = sigmaF^2 * exp(-0.5*(x[i] - y)^2 / l^2)
  }
  return(K)

}



# Function to simulate from the posterior
posteriorGP = function(X, y, XStar, hyperParam, sigmaNoise){

  # ----------------------------------------------------------------------
  # Inputs:
  # X: Vector of training inputs.
  # y: Vector of training targets/outputs.
  # XStar: Vector of inputs where the posterior distribution is evaluated
  # hyperParam: Vector with two elements, sigma_f and l.
  # sigmaNoise: Noise standard deviation sigma_n
  #
```

```r
  # Outputs:
  # mean: a vector with the posterior means for fStar
  # variance: a vector with the posterior variances of fStar
  # llh_marginal: a scalar with the marginal log likelihood
  # ------------------------------------------------------------------------

  # Initialize params
  n = length(y)
  sigmaf = hyperParam[1]
  l = hyperParam[2]
  K = my_kernel(X, X, sigmaf, l)
  KStar = my_kernel(X, XStar, sigmaf, l)

  # Conduct computations
  L = t(chol(K + sigmaNoise^2 * diag(nrow(K))))
  alpha = solve(t(L), solve(L, y))
  fStar = t(KStar) %*% alpha
  v = solve(L, KStar)
  V_fStar = my_kernel(XStar, XStar, sigmaf, l) - t(v) %*% v
  llh_marginal = -1/2 * t(y) %*% alpha - sum(log(L[row(L) == col(L)])) - n/2 * log(2*pi)

  # Return results
  return(list(mean = as.numeric(fStar),
              variance = V_fStar[row(V_fStar) == col(V_fStar)],
              llh_marginal = as.numeric(llh_marginal)))
}



# Parmaters
hyperParam = c(1, 0.3) # sigmaf, l
sigmaNoise = 0.1

# Observation
X = 0.4
y = 0.719

# Test points
XStar = seq(-1, 1, 0.1)

# Compute posterior
res = posteriorGP(X, y, XStar, hyperParam, sigmaNoise)

# Visualization of Posterior
plot_res(res, XStar, X, y)
```

```r
# Parmaters
hyperParam = c(1, 0.3) # sigmaf, l
sigmaNoise = 0.1

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

# Test points
XStar = seq(-1, 1, 0.1)

# Compute posterior
res = posteriorGP(X, y, XStar, hyperParam, sigmaNoise)

# Visualization of Posterior
plot_res(res, XStar, X, y)


# Parmaters
hyperParam = c(1, 0.3) # sigmaf, l
sigmaNoise = 0.1

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

# Test points
XStar = seq(-1, 1, 0.1)

# Compute posterior
res = posteriorGP(X, y, XStar, hyperParam, sigmaNoise)

# Visualization of Posterior
plot_res(res, XStar, X, y)

# Parmaters
hyperParam = c(1, 1) # sigmaf, l
sigmaNoise = 0.1

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

# Test points
XStar = seq(-1, 1, 0.1)
```

```r
# Compute posterior
res = posteriorGP(X, y, XStar, hyperParam, sigmaNoise)

# Visualization of Posterior
plot_res(res, XStar, X, y)



# ------------------------------------------------------------------------------
# Assignment 2
# ------------------------------------------------------------------------------
knitr::include_graphics("images/Q2.1.png")
knitr::include_graphics("images/Q2.2.png")

# Data import
df = read.csv2("input/TempTullinge.csv")

# Add variables
df$time = 1:nrow(df)
df$day = rep(1:365, 6)
df$temp = as.numeric(as.character(df$temp))

# Subset to every 5th obs.
df = df[seq(1, nrow(df), 5), ]

# Show data set
head(df)
tail(df)


?kernlab::gausspr
?kernlab::kernelMatrix


# Define closure (needed for gausspr)
SquaredExpKernel = function(sigmaF, l){
  kernel  = function(x, y){
    r = sqrt(t(x-y) %*% (x-y)) # euclidean distance
    K = sigmaF^2 * exp(-0.5 * r^2 / l^2)
    return(K)
  }
  class(kernel) = "kernel"
  return(kernel)
}
```

```
### Note: We may also define a function that directly gives us a Kernel matrix
# However this works well only if we have a 1 dimensional x-space. If so, then
# kernelMatrix(kernel = SE_kernel, x, y) == SE_kernel(x, y)

SquaredExpKernelMatrix = function(sigmaF, l){
  kernel  = function(x, y){

    K = matrix(NA, length(x), length(y))
    for (i in 1:length(x)){
      r = x[i] - y
      K[i, ] = sigmaF^2 * exp(-0.5 * r^2 / l^2)
    }
    return(K)
  }

  class(kernel) = "kernel"
  return(kernel)
}


# Take hyperparams from KernLabDemo.R
l = 1
sigmaF = 1

# Modify rbfdot kernel with hyperparameter
SE_kernel_rbfdot = rbfdot(sigma = 1/(2*l^2))
SE_kernel = SquaredExpKernel(sigmaF = 1, l = 1)

# Example
SE_kernel(x = 1, y = 2)
SE_kernel_rbfdot(1, 2)


X = c(1, 3, 4)
XStar = c(2, 3, 4)
kernelMatrix(kernel = SE_kernel, x = X, y = XStar)
kernelMatrix(kernel = SE_kernel_rbfdot, x = X, y = XStar)


# Update SE_Kernel with given parameters (for later)
SE_kernel = SquaredExpKernel(sigmaF = 20, l = 0.2)

# Find sigmaNoise
res = lm(temp ~ time + I(time^2), df)
sigmaNoise = sd(res$residuals)
```

```r
# Fit the GP with built in Square expontial kernel (called rbfdot in kernlab)
res = gausspr(temp ~ time, data = df, kernel = SquaredExpKernel,
                kpar = list(sigmaF = 20, l = 0.2), var = sigmaNoise^2)

plot(df$time, df$temp, main = "Data with GP Posterior Mean (orange)")
meanPred = predict(res, df) # Predicting the training data. To plot the fit.
lines(df$time, meanPred, col="orange", lwd = 3)



# Find sigmaNoise
res = lm(temp ~ time + I(time^2), df)
sigmaNoise = sd(res$residuals)

# Fit the GP with built in Square expontial kernel (called rbfdot in kernlab)
res = gausspr(temp ~ time, data = df, kernel = SquaredExpKernel,
                kpar = list(sigmaF = 20, l = 0.2), var = sigmaNoise^2)

plot(df$time, df$temp, ylim = c(-40, 40),
     main = "Data with GP Posterior Mean (orange)
95% CI for Mean (blue) and for Data Points (lightblue)")
meanPred = predict(res, df) # Predicting the training data. To plot the fit.
lines(df$time, meanPred, col="orange", lwd = 3)

# Probability and prediction interval implementation.
# Note: Scaling required since mean was automatically scaled by kernlab as well.
x = scale(df$time) #
xs = scale(df$time) # XStar
n = length(x)
Kss = kernelMatrix(kernel = SE_kernel, x = xs, y = xs)
Kxx = kernelMatrix(kernel = SE_kernel, x = x, y = x)
Kxs = kernelMatrix(kernel = SE_kernel, x = x, y = xs)
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Covariance matrix of fStar

# Probability intervals for fStar
lines(df$time, meanPred - 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)
lines(df$time, meanPred + 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)

# Prediction intervals for yStar
lines(df$time, meanPred - 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")
lines(df$time, meanPred + 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")



# Set parameters ----------------------------------------------------------
```

```r
# SigmaNoise
res = lm(temp ~ day + I(day^2), df)
sigmaNoise = sd(res$residuals)
sigmaNoise = 50
# Hyperparameters
sigmaF = 20
l = 0.2

# Kernel
my_kernel = SquaredExpKernel

# Formula
my_formula = as.formula("temp ~ day")

# Names
x_axis_name = "time"
y_name = "temp"
x_name = "day"

# Fit Model ------------------------------------------------------------------

# Update kernel with given parameters
my_kernel_params = my_kernel(sigmaF = sigmaF, l = l)

x_axis = df[[x_axis_name]]
x_vals = df[[x_name]]
y_vals = df[[y_name]]

# Fit the GP with built in Square expontial kernel (called rbfdot in kernlab)
res = gausspr(my_formula, data = df, kernel = my_kernel,
              kpar = list(sigmaF = sigmaF, l = l), var = sigmaNoise^2)

plot(x_axis, y_vals, ylim = c(-40, 40),
     main = "Data with GP Posterior Mean (orange)
95% CI for Mean (blue) and for Data Points (lightblue)")
meanPred = predict(res, df) # Predicting the training data. To plot the fit.
lines(x_axis, meanPred, col="orange", lwd = 3)

# Probability and prediction interval implementation.
# Note: Scaling required since mean was automatically scaled by kernlab as well.
x = scale(x_vals) #
xs = scale(x_vals) # XStar
n = length(x)
Kss = kernelMatrix(kernel = my_kernel_params, x = xs, y = xs)
Kxx = kernelMatrix(kernel = my_kernel_params, x = x, y = x)
Kxs = kernelMatrix(kernel = my_kernel_params, x = x, y = xs)
```

```r
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Covariance matrix of fStar

# Probability and prediction interval implementation.
# Note: Scaling required since mean was automatically scaled by kernlab as well.
x = scale(x_vals) #
xs = scale(x_vals) # XStar
n = length(x)
Kss = kernelMatrix(kernel = my_kernel_params, x = xs, y = xs)
Kxx = kernelMatrix(kernel = my_kernel_params, x = x, y = x)
Kxs = kernelMatrix(kernel = my_kernel_params, x = x, y = xs)
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Cov. matrix of fStar

# Probability intervals for fStar
lines(x_axis, meanPred - 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)
lines(x_axis, meanPred + 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)

# Prediction intervals for yStar
lines(x_axis, meanPred - 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")
lines(x_axis, meanPred + 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")


# Define closure (needed for gausspr)
PeriodicKernel = function(sigmaF, l1, l2, d){
  kernel  = function(x, y){
    r = sqrt(t(x-y) %*% (x-y)) # euclidean distance
    K = sigmaF^2 * exp(-2 * sin(pi*r/d)^2 / l1^2) *
                   exp(-0.5 * r^2 / l2^2)
    return(K)
  }
  class(kernel) = "kernel"
  return(kernel)
}


### Note: We may also define a function that directly gives us a Kernel matrix
# However this works well only if we have a 1 dimensional x-space. If so, then
# kernelMatrix(kernel = my_kernel_params, x, y) == my_kernel_params(x, y)
PeriodicKernelMatrix = function(sigmaF, l1, l2, d){

  kernel  = function(x, y){

    K = matrix(NA, length(x), length(y))

    for (i in 1:length(x)){
      K[i, ] = sigmaF^2 * exp(-2 * sin(pi*abs(x[i] - y)/d)^2 / l1^2) *
                          exp(-0.5 * (x[i] - y)^2 / l2^2)
```

```
    }
    return(K)

  }

  class(kernel) = "kernel" # needed for
  return(kernel)
}



# Set parameters ---------------------------------------------------------------

# SigmaNoise
res = lm(temp ~ time + I(time^2), df)
sigmaNoise = sd(res$residuals)

# Hyperparameters
sigmaF = 20
l1 = 1
l2 = 10
d = 365/sd(df$time)

# Kernel
my_kernel = PeriodicKernel

# Formula
my_formula = as.formula("temp ~ time")

# Names
x_axis_name = "time"
y_name = "temp"
x_name = "time"

# Fit Model --------------------------------------------------------------------

# Update kernel with given parameters
my_kernel_params = my_kernel(sigmaF = sigmaF, l1 = l1, l2 = l2, d = d)

x_axis = df[[x_axis_name]]
x_vals = df[[x_name]]
y_vals = df[[y_name]]

# Fit the GP with built in Square expontial kernel (called rbfdot in kernlab)
res = gausspr(my_formula, data = df, kernel = my_kernel,
              kpar = list(sigmaF = sigmaF, l1 = l1, l2 = l2, d = d),
```

```
              var = sigmaNoise^2)

plot(x_axis, y_vals, ylim = c(-40, 40),
     main = "Data with GP Posterior Mean (orange)
95% CI for Mean (blue) and for Data Points (lightblue)")
meanPred = predict(res, df) # Predicting the training data. To plot the fit.
lines(x_axis, meanPred, col="orange", lwd = 3)

# Probability and prediction interval implementation.
# Note: Scaling required since mean was automatically scaled by kernlab as well.
x = scale(x_vals) #
xs = scale(x_vals) # XStar
n = length(x)
Kss = kernelMatrix(kernel = my_kernel_params, x = xs, y = xs)
Kxx = kernelMatrix(kernel = my_kernel_params, x = x, y = x)
Kxs = kernelMatrix(kernel = my_kernel_params, x = x, y = xs)
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Covariance matrix of fStar

# Probability and prediction interval implementation.
# Note: Scaling required since mean was automatically scaled by kernlab as well.
x = scale(x_vals) #
xs = scale(x_vals) # XStar
n = length(x)
Kss = kernelMatrix(kernel = my_kernel_params, x = xs, y = xs)
Kxx = kernelMatrix(kernel = my_kernel_params, x = x, y = x)
Kxs = kernelMatrix(kernel = my_kernel_params, x = x, y = xs)
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Cov. matrix of fStar

# Probability intervals for fStar
lines(x_axis, meanPred - 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)
lines(x_axis, meanPred + 1.96*sqrt(diag(Covf)), col = "steelblue4", lwd = 2)

# Prediction intervals for yStar
lines(x_axis, meanPred - 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")
lines(x_axis, meanPred + 1.96*sqrt((diag(Covf) + sigmaNoise^2)), col = "steelblue2")



# ------------------------------------------------------------------------------
# Assignment 3
# ------------------------------------------------------------------------------
knitr::include_graphics("images/Q3.png")

# Data import
df = read.csv('input/banknoteFraud.csv', header = FALSE)
names(df) = c("varWave","skewWave","kurtWave","entropyWave","fraud")
df[,5] = as.factor(df[,5])
```

```r
# Split data
set.seed(111)
idx_tr = sample(1:dim(df)[1], size = 1000, replace = FALSE)
df_tr = df[idx_tr, ]
df_te = df[-idx_tr, ]

head(df_tr)


# Set parameters, fit model
classname = "fraud"
classvals = c(0, 1)
varnames = c("varWave", "skewWave")
res = gausspr(fraud ~ varWave + skewWave, data = df_tr)

# Confusion matrix
cm = table(predict(res, df_tr[,colnames(df_tr) %in% varnames]), # predicted
      df_tr[["fraud"]], dnn = c("pred", "true")) # true
cm
knitr::kable(analyze_cm(cm, true = "1"))

# Create plot
probs = predict(res, df_tr[,colnames(df_tr) %in% varnames], type="probabilities")
x1 = seq(min(df_tr[[varnames[1]]]),max(df_tr[[varnames[1]]]),length=100)
x2 = seq(min(df_tr[[varnames[2]]]),max(df_tr[[varnames[2]]]),length=100)
gridPoints = meshgrid(x1, x2)
gridPoints = cbind(c(gridPoints$x), c(gridPoints$y))
gridPoints = data.frame(gridPoints)
names(gridPoints) = varnames
probs = predict(res, gridPoints, type="probabilities")

# Plotting for Prob (classval 1). Note: Add more points for more classes
contour(x1,x2,matrix(probs[,colnames(probs) == classvals[1]],100,byrow = TRUE),
        20, xlab = varnames[1], ylab = varnames[2],
        main = paste0('Prob(class=', classvals[1],') [in red]'))
points(df_tr[[varnames[1]]][df_tr[[classname]]==classvals[1]],
       df_tr[[varnames[2]]][df_tr[[classname]]==classvals[1]], col="red")
points(df_tr[[varnames[1]]][df_tr[[classname]]==classvals[2]],
       df_tr[[varnames[2]]][df_tr[[classname]]==classvals[2]], col="blue")

# Plotting for Prob (classval 2). Note: Add more points for more classes
contour(x1,x2,matrix(probs[,colnames(probs) == classvals[2]],100,byrow = TRUE),
        20, xlab = varnames[1], ylab = varnames[2],
        main = paste0('Prob(class=', classvals[2],') [in blue]'))
points(df_tr[[varnames[1]]][df_tr[[classname]]==classvals[1]],
       df_tr[[varnames[2]]][df_tr[[classname]]==classvals[1]], col="red")
```

```
points(df_tr[[varnames[1]]][df_tr[[classname]]==classvals[2]],
       df_tr[[varnames[2]]][df_tr[[classname]]==classvals[2]], col="blue")


# Confusion matrix
cm = table(predict(res, df_te[,colnames(df_te) %in% varnames]), # predicted
      df_te[["fraud"]], dnn = c("pred", "true")) # true
cm
knitr::kable(analyze_cm(cm, true = "1"))


# Set parameters, fit model
classname = "fraud"
classvals = c(0, 1)
varnames = c("varWave", "skewWave", "kurtWave", "entropyWave")
res = gausspr(fraud ~ varWave + skewWave + kurtWave + entropyWave, data = df_tr)

# Confusion matrix
cm = table(predict(res, df_te[,colnames(df_te) %in% varnames]), # predicted
      df_te[["fraud"]], dnn = c("pred", "true")) # true
cm
knitr::kable(analyze_cm(cm, true = "1"))
```