Advanced Machine Learning - lab04

Lennart Schilling (lensc874)

2019-10-14

Contents

Lab04. Gaussian Processes. **2**
Assignment 1. Implementing GP Regression. 2
1a. Implementing functions regarding simulation from posterior GP. 3
1b. Deriving posterior using one training point. 6
1c. Deriving posterior using two training points. 7
1d. Deriving posterior using five training points. 8
1e. Deriving posterior using larger l-parameter. 9
Assignment 2. GP Regression with kernlab. 10
2a. Implementing and evaluating own square exponential kernel function. 11
2b. Estimating and plotting GP posterior using gausspr(). 12
2c. Integrating confidence bands using own calculated posterior variances. 14
2d. Fitting another GP. Comparing results. 15
2e. Implementing generalization of periodic kernel. Estimating GP posterior. 17
Assignment 3. GP Classification with kernlab. 19
3a. Fitting GP classifier. Plotting contours over grid. Confusion matrix and accuracy. 19
3b. Using classifier to predict for test set. 22
3c. Fitting classifier using four covariates. Comparing prediction results. 23
Appendix. 24

Lab04. Gaussian Processes.

Assignment 1. Implementing GP Regression.

This first exercise will have you writing your own code for the Gaussian process regression model:
$y = f(x) + \epsilon$ with $\epsilon \sim \mathcal{N}\left(0, \sigma_n^2\right)$ and $f \sim \mathcal{GP}\left(0, k\left(x, x'\right)\right)$
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 $A\backslash b$ means the vector x that solves the equation $Ax = b$ (see p. xvii in the book). This is implemented in R with the help of the function solve.

input: X (inputs), $\mathbf{y}$ (targets), k (covariance function), σ_n^2 (noise level), $\mathbf{x}_*$ (test input)

2: $L := \text{cholesky}(K + \sigma_n^2 I)$

$\boldsymbol{\alpha} := L^\top \backslash (L \backslash \mathbf{y})$ } predictive mean eq. (2.25)

4: $\bar{f}_* := \mathbf{k}_*^\top \boldsymbol{\alpha}$

$\mathbf{v} := L \backslash \mathbf{k}_*$ } predictive variance eq. (2.26)

6: $\mathbb{V}[f_*] := k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{v}^\top \mathbf{v}$

$\log p(\mathbf{y}|X) := -\frac{1}{2}\mathbf{y}^\top \boldsymbol{\alpha} - \sum_i \log L_{ii} - \frac{n}{2}\log 2\pi$ eq. (2.30)

8: **return**: $\bar{f}_*$ (mean), $\mathbb{V}[f_*]$ (variance), $\log p(\mathbf{y}|X)$ (log marginal likelihood)

Algorithm 2.1: Predictions and log marginal likelihood for Gaussian process regression. The implementation addresses the matrix inversion required by eq. (2.25) and (2.26) using Cholesky factorization, see section A.4. For multiple test cases lines 4-6 are repeated. The log determinant required in eq. (2.30) is computed from the Cholesky factor (for large n it may not be possible to represent the determinant itself). The computational complexity is $n^3/6$ for the Cholesky decomposition in line 2, and $n^2/2$ for solving triangular systems in line 3 and (for each test case) in line 5.

1a. Implementing functions regarding simulation from posterior GP.

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).

Implementing function for squared exponential kernel.

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

Implementing function for GP posterior according the given algorithm.

The function will be implemented for the squared exponential kernel.

```
posteriorGP = function(X, # Vector of training inputs.
                       y, # Vector of training targets/outputs.
                       XStar, # Vector of inputs where the posterior distribution is evaluated.
                       hyperParam, # Vector with kernel parameters sigma_f, l.
                       sigmaNoise) { # Noise standard deviation sigma_n.

  # Calculating number of training inputs.
  n = length(X)

  # Calculating covariance matrices.
    # K in algorithm equals to k(X, X).
    k_X_X = SquaredExpKernel(x1 = X, x2 = X,
                             sigma_f = hyperParam[1], l = hyperParam[2])
    # k* in algorithm equals to k(X, X*).
    k_X_XStar = SquaredExpKernel(x1 = X, x2 = XStar,
                                 sigma_f = hyperParam[1], l = hyperParam[2])
    # k(x*,x*)
    k_XStar_XStar = SquaredExpKernel(x1 = XStar, x2 = XStar,
                                     sigma_f = hyperParam[1], l = hyperParam[2])

  # Calculating L (line 2).
    # Calculating upper triangular part.
    L_trans = chol(k_X_X + (sigmaNoise^2)*diag(n))
    # Calculating L as the lower triangular part by transposing.
    L = t(L_trans)
```

```
  # Calculating alpha (line 3).
  alpha = solve(a = L_trans, b = solve(a = L, b = y))

  # Calculating posterior mean function values fStar (line 4).
  fStar_mean = t(k_X_XStar) %*% alpha

  # Calculating posterior covariance matrix for fStar (line 5-6).
  v = solve(a = L, b = k_X_XStar)
  fStar_covMatrix = k_XStar_XStar - t(v) %*% v

  # Returning vector with posterior mean and variance of fStar.
  return(list(mean = fStar_mean,
              variance = diag(fStar_covMatrix))) # Not covariance matrix, but only variances are return
}
```

Implementing function to visualize posteriorGP.

```
plot_posteriorGP = function(X, # Vector of training inputs.
                            y, # Vector of training targets/outputs.
                            XStar, # Vector of inputs where the posterior distribution is evaluated.
                            XStar_mean, # Vector of posterior mean values.
                            XStar_variance) { # Vector of posterior variances.

  # Computing 95%-confidence interval.
  ci = data.frame(upper = XStar_mean + 1.96*sqrt(XStar_variance),
                  lower = XStar_mean - 1.96*sqrt(XStar_variance))

  # Plotting posterior mean values.
  plot(x = XStar,
       y = XStar_mean,
       type= "l",
       lwd = 2,
       col = "blue",
       ylim = c(min(ci$lower)-0.2, max(ci$upper)+2),
       main = "Posterior of f",
       xlab = "Input",
       ylab = "Output")

  # Plotting confidence intrval.
  polygon(x = c(rev(XStar), XStar),
          y = c(rev(ci$upper), ci$lower),
          col = rgb(0, 0, 0, 0.1))
          #col = "grey")
          #density = 20)

  # Plotting training points.
  points(x = X,
         y = y,
         col = "black",
         lwd = 1)

  # Adding legend.
  legend('topleft',
```

```
        legend = c("Mean", "Confidence Interval", "Train points"),
        #bty = "n",
        horiz = T,
        col = c("blue", "grey", "black"),
        lty = c(1, NA, NA),
        pch = c(NA, 15, 1))
}
```

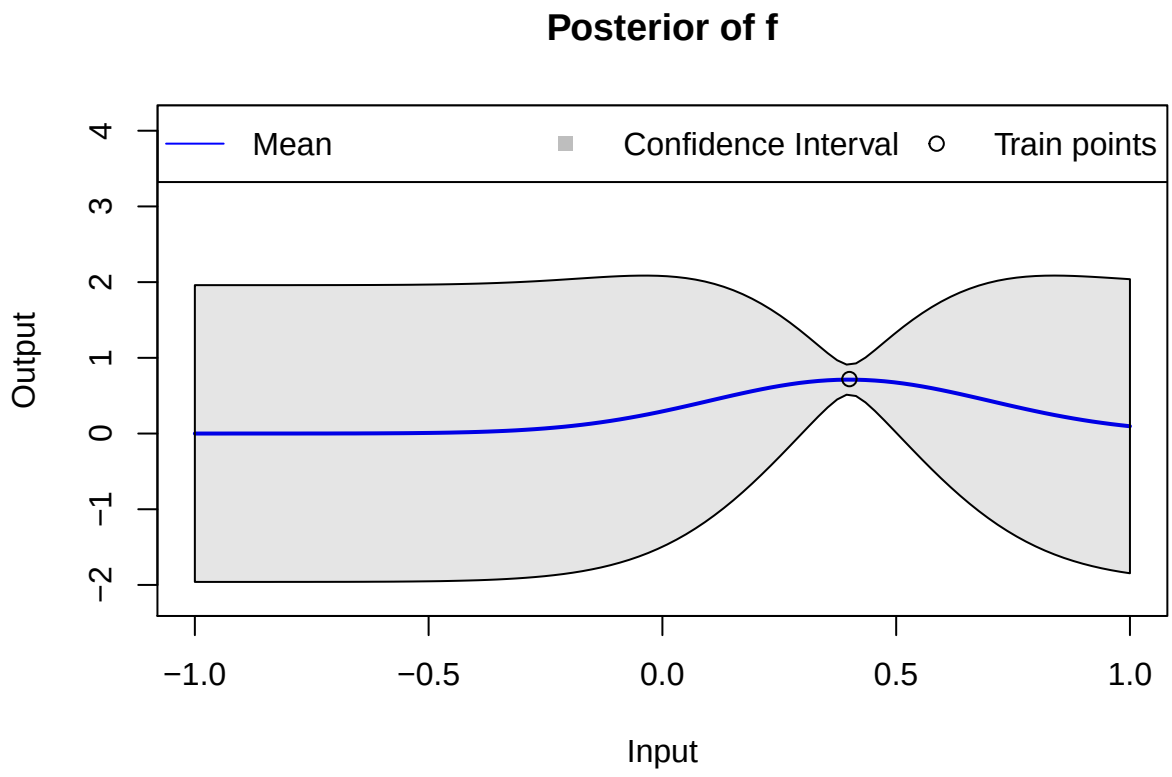
1b. Deriving posterior using one training point.

Now, let the prior hyperparameters be $\sigma_f = 1$ and $l = 0.3$. 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

```
# Defining test points over interval [-1, 1].
XStar = seq(from = -1, to = 1, length.out = 100)

# Deriving posterior.
posterior = posteriorGP(X = 0.4, y = 0.719, # One training point (0.4, 0.719).
                        XStar = XStar, # Test points.
                        hyperParam = c(1, 0.3), # Sigma_f and l.
                        sigmaNoise = 0.1) # Sigma_n.

# Plotting posterior.
plot_posteriorGP(X = 0.4, y = 0.719,
                 XStar = XStar,
                 XStar_mean = posterior$mean,
                 XStar_variance = posterior$variance)
```

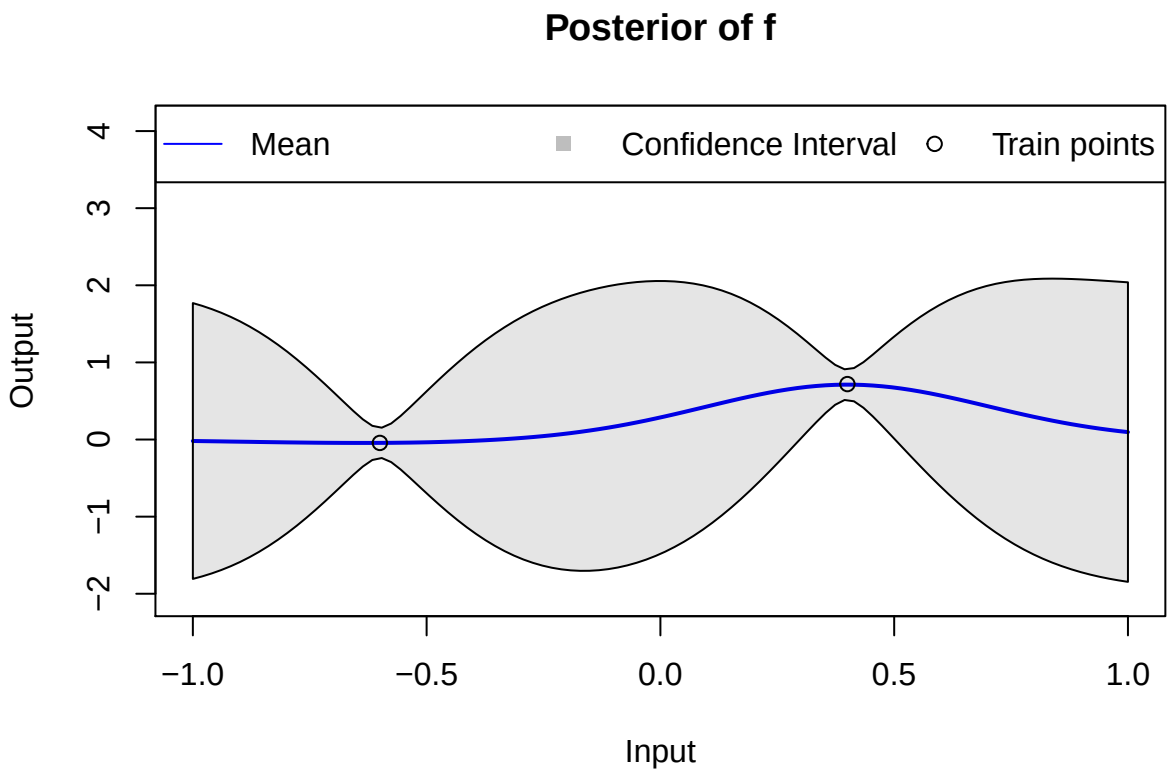


1c. Deriving posterior using two training points.

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.

```
# Deriving posterior.
posterior = posteriorGP(X = c(0.4, -0.6), y = c(0.719, -0.044),
                        # Two training points (0.4, 0.719) and (-0.6, -0.044).
                        XStar = XStar, # Test points.
                        hyperParam = c(1, 0.3), # Sigma_f and l.
                        sigmaNoise = 0.1) # Sigma_n.

# Plotting posterior.
plot_posteriorGP(X = c(0.4, -0.6), y = c(0.719, -0.044),
                 XStar = XStar,
                 XStar_mean = posterior$mean,
                 XStar_variance = posterior$variance)
```



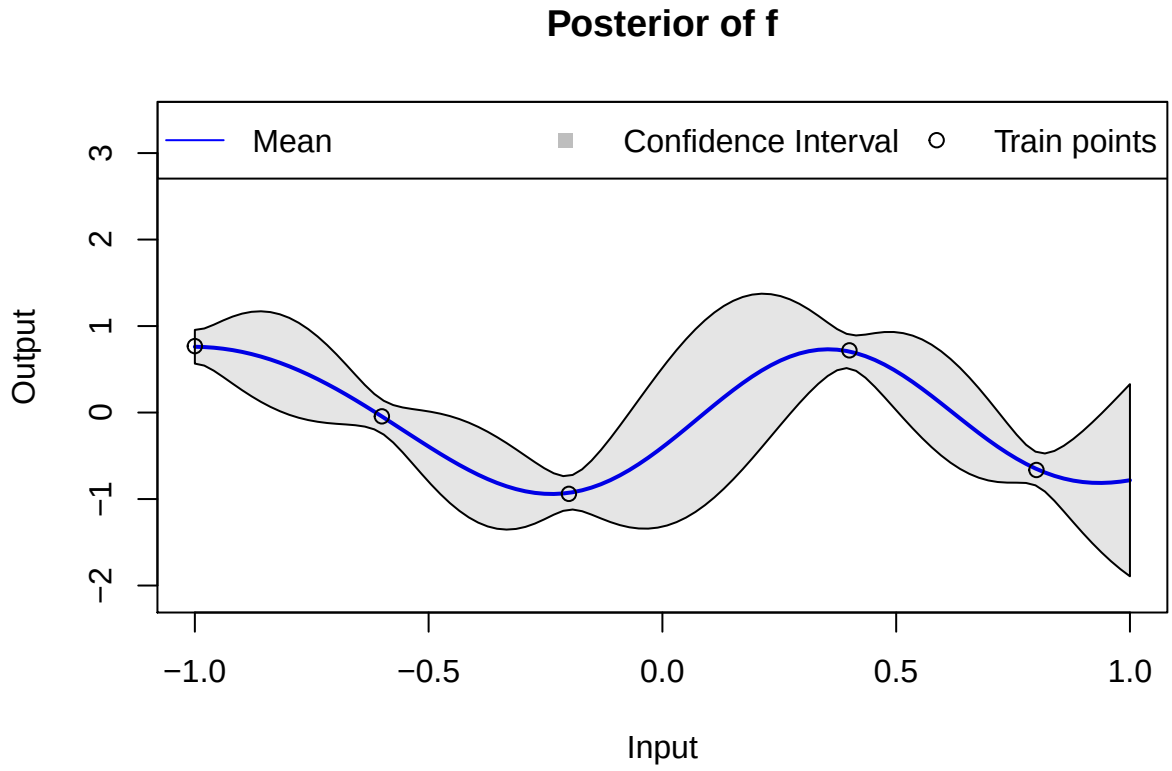
1d. Deriving posterior using five training points.

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

```
# Deriving posterior.
posterior = posteriorGP(X = c(-1.0, -0.6, -0.2, 0.4, 0.8),
                        y = c(0.768, -0.044, -0.940, 0.719, -0.664),
                        XStar = XStar,
                        hyperParam = c(1, 0.3),
                        sigmaNoise = 0.1)

# Plotting posterior.
plot_posteriorGP(X = c(-1.0, -0.6, -0.2, 0.4, 0.8),
                 y = c(0.768, -0.044, -0.940, 0.719, -0.664),
                 XStar = XStar,
                 XStar_mean = posterior$mean,
                 XStar_variance = posterior$variance)
```

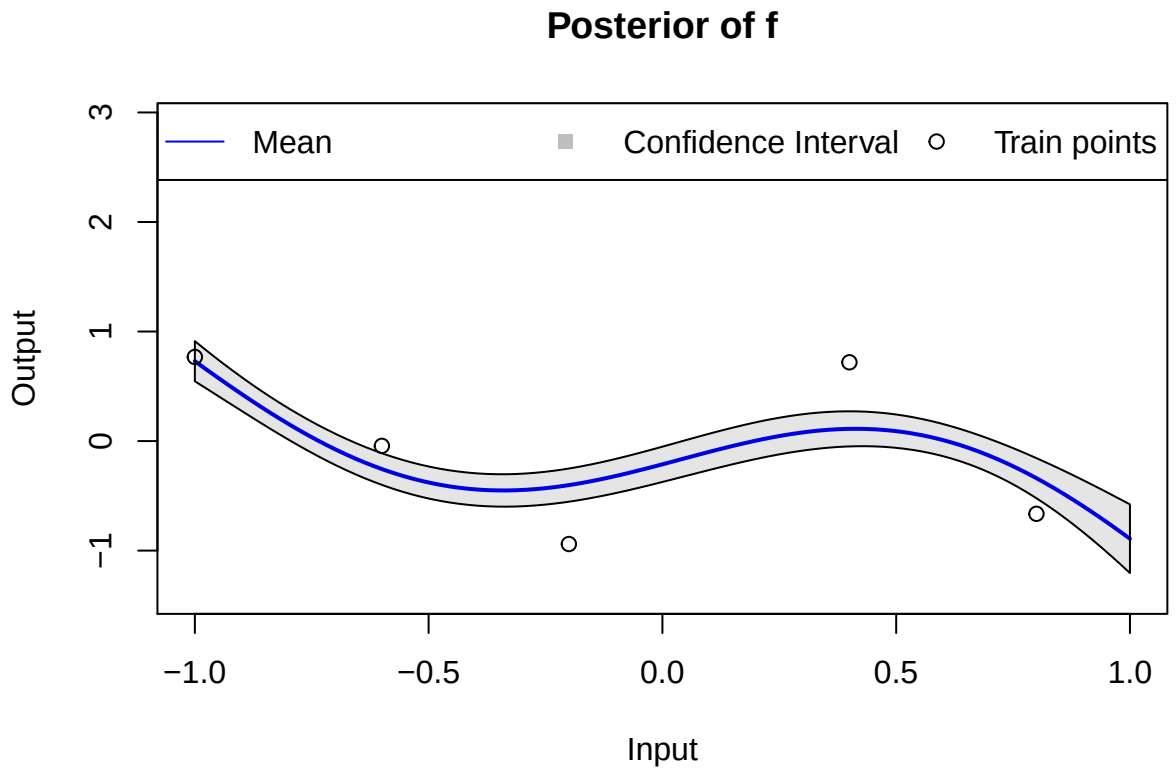


1e. Deriving posterior using larger l-parameter.

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

```
# Deriving posterior.
posterior = posteriorGP(X = c(-1.0, -0.6, -0.2, 0.4, 0.8),
                        y = c(0.768, -0.044, -0.940, 0.719, -0.664),
                        XStar = XStar,
                        hyperParam = c(1, 1), # l is changed to 1.
                        sigmaNoise = 0.1)

# Plotting posterior.
plot_posteriorGP(X = c(-1.0, -0.6, -0.2, 0.4, 0.8),
                 y = c(0.768, -0.044, -0.940, 0.719, -0.664),
                 XStar = XStar,
                 XStar_mean = posterior$mean,
                 XStar_variance = posterior$variance)
```



Since l defines the distance we have to move in the input space so that the function varies signifiantly, a larger choice of l leads to a smoother function. Our plot confirms that. The possible functions are so smooth that they do not catch the actual training points anymore. However, the general pattern of the functions follow the given training points, so that a higher (or lower) output of a training point leads to a higher (or lower, respectively) output in the posterior as well.

Assignment 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, ..., $365 \times 6 = 2190$). Also, create the variable day that records the day number since the start of each year (i.e., day= 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.

```
# Package setup.
library(kernlab)

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

# Adding time and day variables.
data$time = 1:nrow(data)
data$day = rep(x = seq(from = 1, to = 365), times = 6)

# Subsampling data. Only keeping every fifth observation.
time_keep = seq(from = 1, to = nrow(data), by = 5)
data = data[data$time %in% time_keep, ]
```

2a. Implementing and evaluating own square exponential kernel function.

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 $X_* = (2, 3, 4)^T$.

Implementing nested kernel function.

Since the `kernlab`-package requires the kernel to be a function of class kernel taking two vector arguments and returning a scalar, we first define your own nested square exponential kernel function.

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

Evaluating kernel using $\sigma_f = 1$ and $l = 0.3$

```
# Setting kernel.
my_squared_exp_kernel = nested_squared_exp_kernel(sigma_f = 1, l = 0.3)

# Evaluating in point x = 1, x' = 2.
my_squared_exp_kernel(x1 = 1, x2 = 2)
```

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

```
# Computing covariance matrix K( X = (1,3,4), X* = (2,3,4) )
kernelMatrix(kernel = my_squared_exp_kernel,
             x = c(1, 2, 4),
             y = c(2, 3, 4))
```

```
An object of class "kernelMatrix"
             [,1]         [,2]         [,3]
[1,] 3.865920e-03 2.233631e-10 1.928750e-22
[2,] 1.000000e+00 3.865920e-03 2.233631e-10
[3,] 2.233631e-10 3.865920e-03 1.000000e+00
```

```
# Manual solution with same result.
# my_squared_exp_kernel(x1 = c(1, 2, 4), x2 = c(2, 3, 4))
```

2b. Estimating and plotting GP posterior using gausspr().

Consider first the following model:
temp $= f(\text{time}) + \epsilon$ with $\epsilon \sim \mathcal{N}\left(0, \sigma_n^2\right)$ and $f \sim \mathcal{GP}\left(0, k\left(\text{time}, \text{time}'\right)\right)$
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 (2a) with $\sigma_f = 20$ and $l = 0.2$. 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).

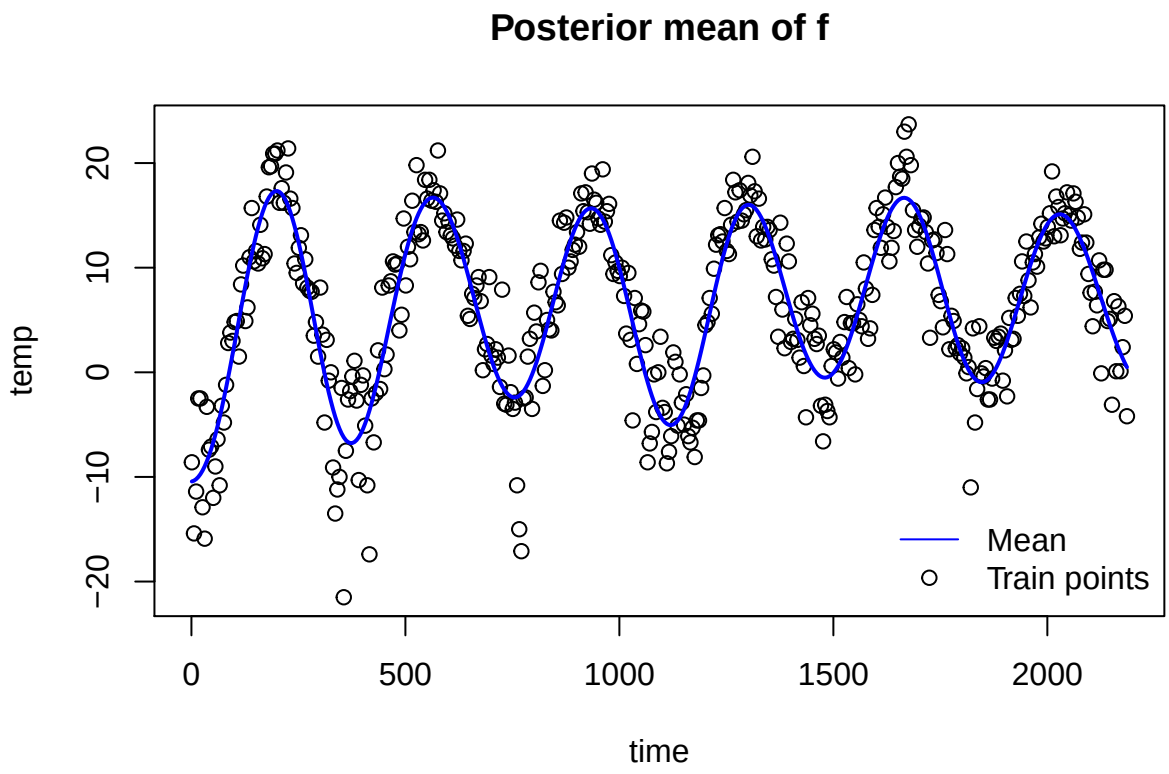
```
# Estimating sigma_n by sd(residuals) of linear quadratic regression fit.
polyFit = lm(data$temp ~  data$time + I(data$time^2) )
sigmaNoise = sd(polyFit$residuals)

# Estimating the above GP regression model using own squared exponential function.
my_squared_exp_kernel = nested_squared_exp_kernel(sigma_f = 20, l = 0.2)
GPfit = gausspr(data$time, # x of training data.
                data$temp, # f(x) of training data.
                kernel = my_squared_exp_kernel,
                var = sigmaNoise^2)

# Estimating the above GP regression model using built-in Square exponential kernel.
#GPfit = gausspr(data$time, data$temp,
#                kernel = rbfdot, kpar = list(sigma = 1/(2*ell^2)),
#                var = sigmaNoise^2)

# Computing posterior mean at every data point in the training dataset.
meanPred_time = predict(GPfit, # Model.
                        data$time) # Test data (equals training data in this case).

# Plotting the fit.
plot(x = data$time, y = data$temp,
     main = "Posterior mean of f",
     xlab = "time",
     ylab = "temp")
lines(data$time, meanPred_time, col="blue", lwd = 2)
legend('bottomright',
       legend = c("Mean", "Train points"),
       bty = "n",
       col = c("blue", "black"),
       lty = c(1, NA),
       pch = c(NA, 1))
```

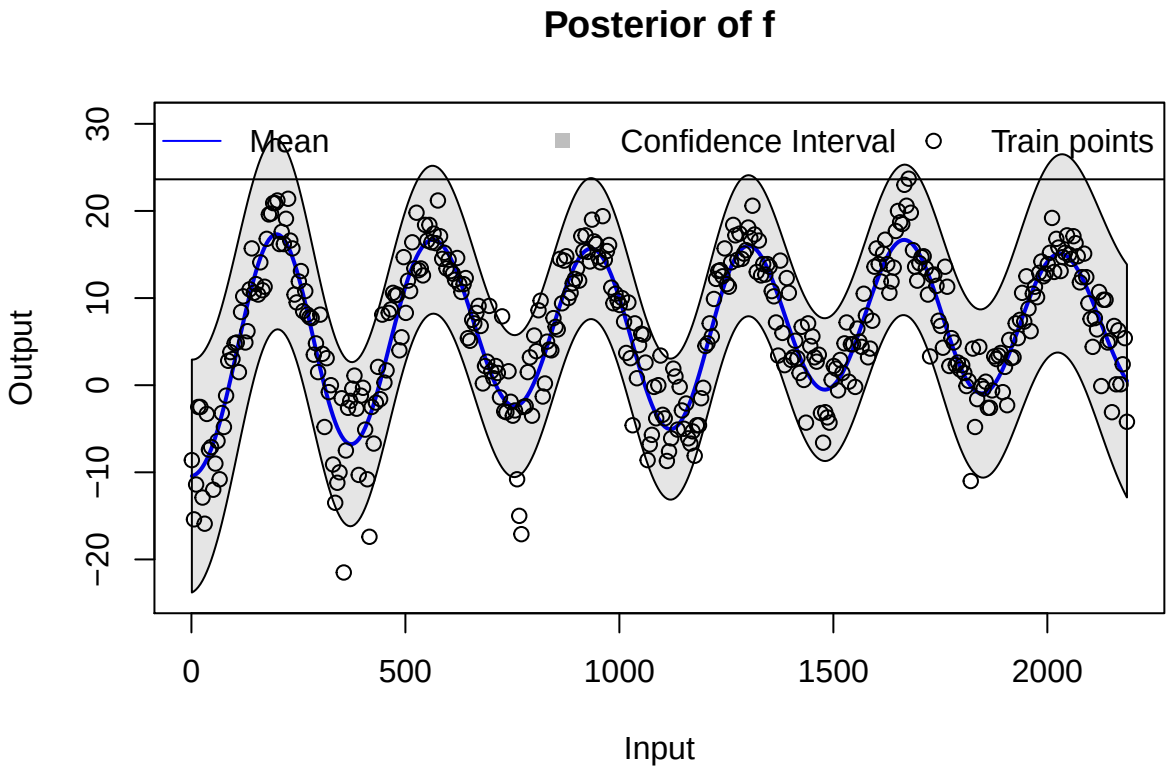

Posterior mean of f
temp
time
Mean
Train points
20
10
0
−10
−20
0
500
1000
1500
2000

2c. Integrating confidence bands using own calculated posterior variances.

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 (2b).
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.

```
# Deriving posterior using implemented function from 1a.
own_posterior = posteriorGP(X = scale(data$time), # Using scaled data as kernlab does.
                            y = scale(data$temp),
                            XStar = scale(data$time), # Again using train as test points.
                            hyperParam = c(20, 2), # Using same kerne parameter.
                            sigmaNoise = sigmaNoise^2)

# Plotting posterior.
plot_posteriorGP(X = data$time,
                 y = data$temp,
                 XStar = data$time,
                 XStar_mean = meanPred_time, # Using mean prediction from kernlab.
                 XStar_variance = own_posterior$variance) # Using variances from own.
```



2d. Fitting another GP. Comparing results.

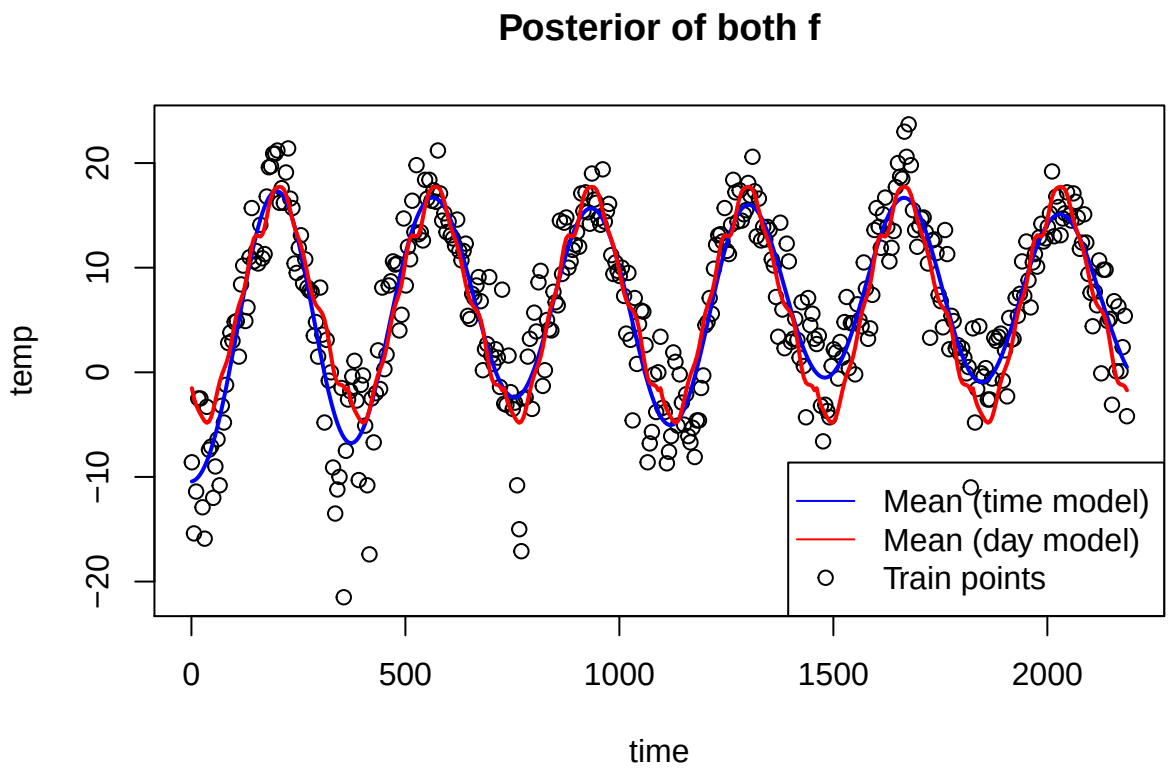
Consider now the following model:
temp $= f(day) + \epsilon$ with $\epsilon \sim \mathcal{N}\left(0, \sigma_n^2\right)$ and $f \sim \mathcal{GP}\left(0, k\left(\text{day}, day'\right)\right)$
Estimate the model using the squared exponential function with $\sigma_f = 20$ and $l = 0.2$. Superimpose the posterior mean from this model on the posterior mean from the model in (2b). 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?

```
# Estimating sigma_n by sd(residuals) of linear quadratic regression fit.
polyFit = lm(data$temp ~  data$day + I(data$day^2) )
sigmaNoise = sd(polyFit$residuals)

# Estimating the above GP regression model using own squared exponential function.
my_squared_exp_kernel = nested_squared_exp_kernel(sigma_f = 20, l = 0.2)
GPfit = gausspr(data$day, # x of training data.
                data$temp, # f(x) of training data.
                kernel = my_squared_exp_kernel,
                var = sigmaNoise^2)

# Computing posterior mean at every data point in the training dataset.
meanPred_day = predict(GPfit, # Model.
                       data$day) # Test data (equals training data in this case).

# Plotting fit "time-model" vs. the fit "day-model".
  # Training points. X-axis shows time.
  plot(x = data$time, y = data$temp,
       main = "Posterior of both f",
       col = "black",
       xlab = "time",
       ylab = "temp")
  # Posterior mean from "time-model".
  lines(data$time, meanPred_time, col="blue", lwd = 2)
  # Posterior mean from "day-model".
  lines(data$time, meanPred_day, col="red", lwd = 2)
  # Legend.
  legend('bottomright',
         legend = c("Mean (time model)", "Mean (day model)", "Train points"),
         #bty = "n",
         col = c("blue", "red", "black"),
         lty = c(1, 1, NA),
         pch = c(NA, NA, 1))
```



As it can be seen, the postrior mean of the new fitted model using `day` to explain the temperature is less smooth than the one from assignment 2b which uses `time` instead. However, as a pro argument one might say that the GP using the `day`-variable has less dimensions. The covariance has a dimension of 365×365). In contrast, using the `time`-variable, the resulting covariance matrix of the GP has a dimension of 2190×2190. Thus, it probably might take less time to compute.

2e. Implementing generalization of periodic kernel. Estimating GP posterior.

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 $d = 365/\text{sd(t ime)}$. 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 $l = 0.2$. Discuss the results.

Implementing nested generalization of periodic kernel.

Again, since the `kernlab`-package requires the kernel to be a function of class kernel taking two vector arguments and returning a scalar, we first define your own nested periodic kernel function.

```
nested_general_periodic_kernel = function(sigma_f, l1, l2, d) {
  general_periodic_kernel = function (x, y = NULL) {
    r = abs(x - y)
    return ((sigma_f^2)*exp(-2*(sin(pi*r/d)^2)/(l1^2))*exp(-0.5*(r/l2)^2))
  }
  class(general_periodic_kernel) = "kernel"
  return (general_periodic_kernel)
}
```

Estimating and plotting GP posterior.

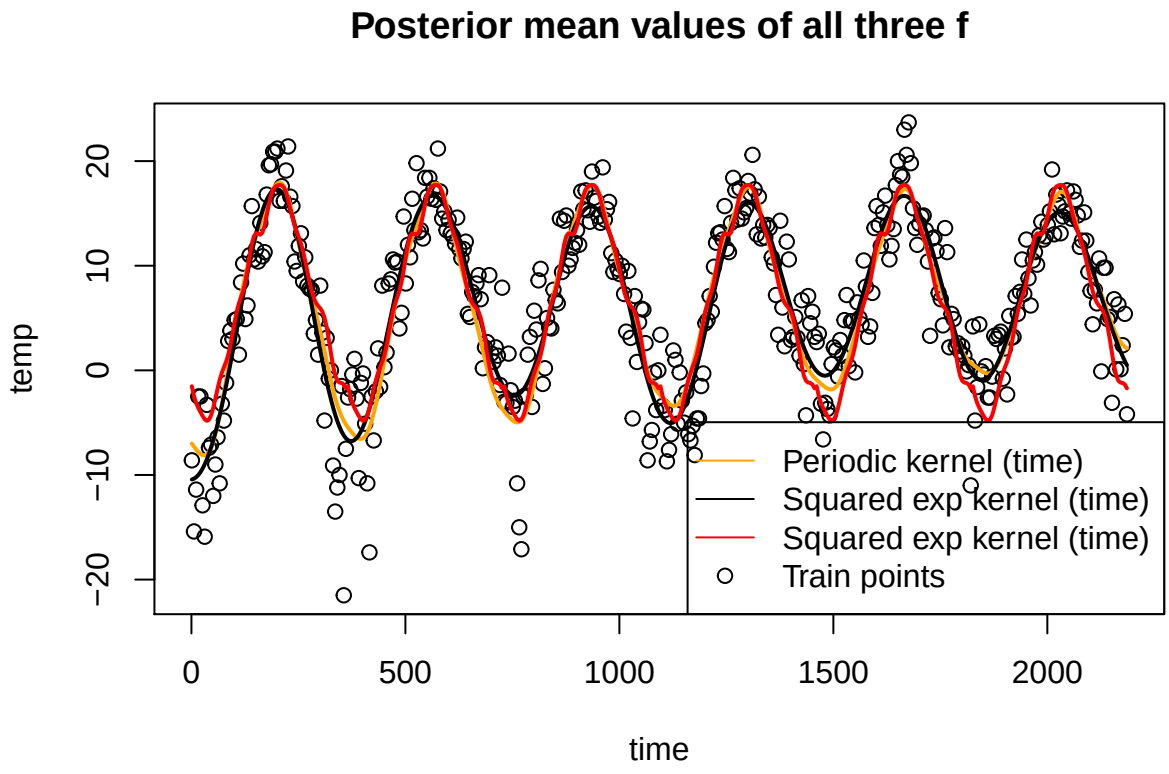
```
# Setting kernel.
my_general_periodic_kernel = nested_general_periodic_kernel(sigma_f = 20,
                                                            l1 = 1,
                                                            l2 = 10,
                                                            d = 365/sd(data$time))

# Estimating GP regression model using periodic_kernel function.
GPfit = gausspr(data$time, # x of training data.
                data$temp, # f(x) of training data.
                kernel = my_general_periodic_kernel,
                var = sigmaNoise^2)

# Computing posterior mean at every data point in the training dataset.
meanPred_time_periodic_kernel = predict(GPfit, # Model.
                                        data$time) # Test data.

# Plotting the fit compared to the previous models.
plot(x = data$time, y = data$temp,
     main = "Posterior mean values of all three f",
     xlab = "time",
     ylab = "temp")
lines(data$time, meanPred_time_periodic_kernel, col="orange", lwd = 2)
lines(data$time, meanPred_time, col = "black", lwd = 2)
lines(data$time, meanPred_day, col="red", lwd = 2)
legend('bottomright',
       legend = c("Periodic kernel (time)",
                  "Squared exp kernel (time)",
                  "Squared exp kernel (time)",
```

```
                "Train points"),
       # bty = "n",
       col = c("orange", "black", "red", "black"),
       lty = c(1, 1, 1, NA),
       pch = c(NA, NA, NA, 1))
```



As a result, the model using the squared exponential kernel and the `time`-variable returns the smoothest posterior.

Assignment 3. GP Classification with kernlab.

Download the banknote fraud data. 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)

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

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

3a. Fitting GP classifier. Plotting contours over grid. Confusion matrix and accuracy.

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.

Fitting classifier.

```
GP_classifier = gausspr(fraud ~ varWave + skewWave, data = train)
```

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

Creating grid to classify on.

```
grid_varWave = seq(from = min(train$varWave),
                   to = max(train$varWave),
                   length = 100)
grid_skewWave = seq(from = min(train$skewWave),
                    to = max(train$skewWave),
                    length = 100)
library(AtmRay)
gridPoints = meshgrid(grid_varWave, grid_skewWave)
gridPoints = cbind(c(gridPoints$x), c(gridPoints$y))
gridPoints = data.frame(gridPoints)
names(gridPoints) = c("varWave", "skewWave")
```

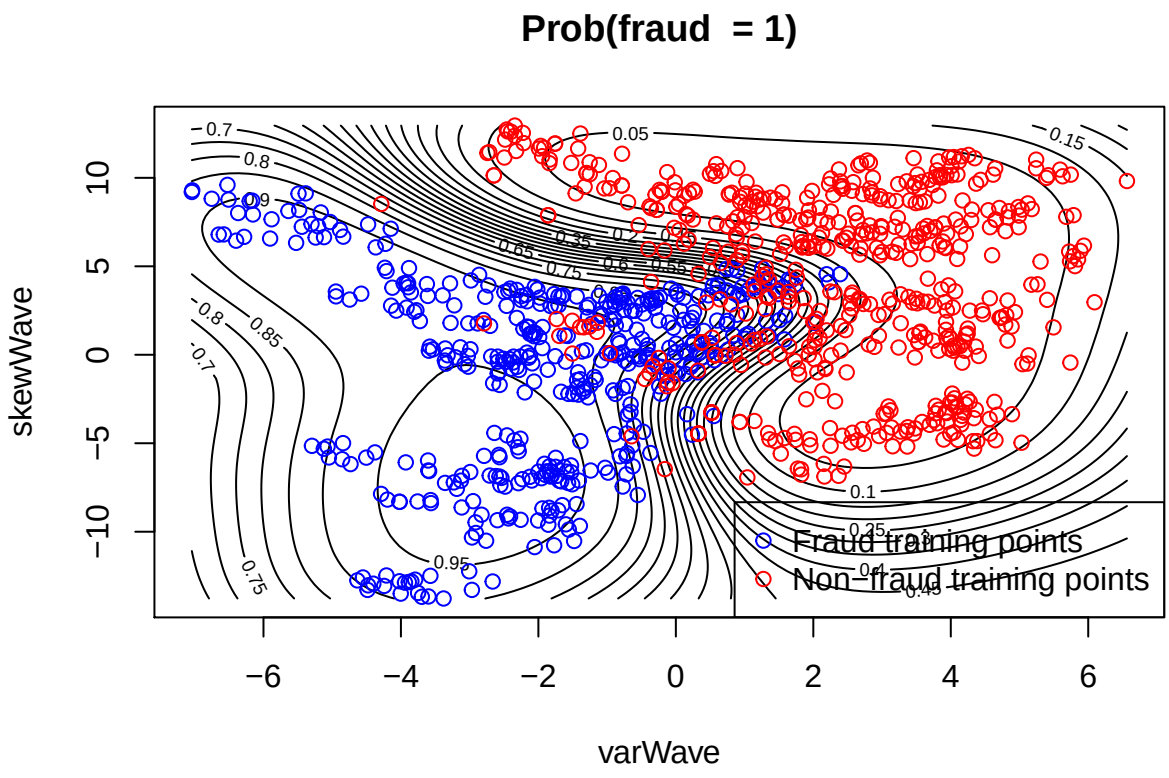
Predicting fraud on grid data using fitted GP classification.

```
GP_pred_grid = predict(GP_classifier, # Classifier.
                       gridPoints, # Data to classify on.
                       type = "probabilities") # Returning probabilities.
```

Plotting contours for varWave and skewWave related to fraud classification.

Given a value for z, lines are drawn for connecting the (x,y) coordinates where that z value occurs.

```
contour(x = grid_varWave,
        y = grid_skewWave,
        # Plotting probabilities of fraud = 1 using 100x100 matrix.
        z = matrix(GP_pred_grid[, 2], 100, byrow = TRUE),
        nlevels = 20, # Number of contour levels.
        xlab = "varWave",
        ylab = "skewWave",
        main = "Prob(fraud  = 1)")
# Adding fraud/non-fraud information of training points.
points(x = train$varWave[which(train$fraud == 1)],
       y = train$skewWave[which(train$fraud == 1)],
       col = "blue")
points(x = train$varWave[which(train$fraud == 0)],
       y = train$skewWave[which(train$fraud == 0)],
       col = "red")
# Adding legend.
legend('bottomright',
       legend = c("Fraud training points", "Non-fraud training points"),
       col = c("blue", "red"),
       pch = c(1, 1))
```



Computing confusion matrix and accuracy for classifier.

```
# Predicting fraud on training data using fitted GP classification.
GP_pred_train = predict(GP_classifier, # Classifier.
                        train) # Data to classify on.
```

```r
# Computing confusion matrix for train data.
confusion_matrix_train = table(GP_pred_train, train$fraud)
knitr::kable(confusion_matrix_train)
```

	0	1
0	512	24
1	44	420

```r
# Computing accuracy.
# sum(GP_pred_train == train$fraud) / length(GP_pred_train)
sum(diag(confusion_matrix_train))/sum(confusion_matrix_train)
```

```
[1] 0.932
```

3b. Using classifier to predict for test set.

Using the estimated model from (3a), make predictions for the test set. Compute the accuracy.

```
# Predicting fraud on test data using fitted GP classification.
GP_pred_test = predict(GP_classifier, # Classifier.
                       test) # Data to classify on.

# Computing confusion matrix for train data.
confusion_matrix_test = table(GP_pred_test, test$fraud)
knitr::kable(confusion_matrix_test)
```

	0	1
0	191	9
1	15	157

```
# Computing accuracy.
# sum(GP_pred_train == train$fraud) / length(GP_pred_train)
sum(diag(confusion_matrix_test))/sum(confusion_matrix_test)
```

```
[1] 0.9354839
```

3c. Fitting classifier using four covariates. Comparing prediction results.

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

```
# Fitting GP classifier using all covariates.
GP_classifier = gausspr(fraud ~ ., data = train)
```

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

```
# Predicting fraud on test data using fitted GP classification.
GP_pred_test = predict(GP_classifier, # Classifier.
                       test) # Data to classify on.

# Computing confusion matrix for train data.
confusion_matrix_test = table(GP_pred_test, test$fraud)
knitr::kable(confusion_matrix_test)
```

	0	1
0	205	0
1	1	166

```
# Computing accuracy.
# sum(GP_pred_train == train$fraud) / length(GP_pred_train)
sum(diag(confusion_matrix_test))/sum(confusion_matrix_test)
```

```
[1] 0.9973118
```

The accuracy still improved up to 99.7%.

Appendix.

The provided R code is attached.

Gaussian Processes.R

```r
#install.packages("mvtnorm")
library("mvtnorm")

# Covariance function
SquaredExpKernel <- function(x1,x2,sigmaF=1,l=3){
  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)
}

# Mean function
MeanFunc <- function(x){
  #m <- sin(x)
  m = rep(0, length(x))
  return(m)
}

# Simulates nSim realizations (function) from a GP with mean m(x) and covariance K(x,x')
# over a grid of inputs (x)
SimGP <- function(m = 0,K,x,nSim,...){
  n <- length(x)
  if (is.numeric(m)) meanVector <- rep(0,n) else meanVector <- m(x)
  covMat <- K(x,x,...)
  f <- rmvnorm(nSim, mean = meanVector, sigma = covMat)
  return(f)
}

xGrid <- seq(-5,5,length=20)

# Plotting one draw
sigmaF <- 1
l <- 1
nSim <- 1
fSim <- SimGP(m=MeanFunc, K=SquaredExpKernel, x=xGrid, nSim, sigmaF, l)
plot(xGrid, fSim[1,], type="p", ylim = c(-3,3))
if(nSim>1){
  for (i in 2:nSim) {
    lines(xGrid, fSim[i,], type="p")
  }
}
lines(xGrid,MeanFunc(xGrid), col = "red", lwd = 3)
lines(xGrid, MeanFunc(xGrid) - 1.96*sqrt(diag(SquaredExpKernel(xGrid,xGrid,sigmaF,l))), col = "blue", lw
lines(xGrid, MeanFunc(xGrid) + 1.96*sqrt(diag(SquaredExpKernel(xGrid,xGrid,sigmaF,l))), col = "blue", lw

# Plotting using manipulate package
```

```r
library(manipulate)

plotGPPrior <- function(sigmaF, l, nSim){
  fSim <- SimGP(m=MeanFunc, K=SquaredExpKernel, x=xGrid, nSim, sigmaF, l)
  plot(xGrid, fSim[1,], type="l", ylim = c(-3,3), ylab="f(x)", xlab="x")
  if(nSim>1){
    for (i in 2:nSim) {
      lines(xGrid, fSim[i,], type="l")
    }
  }
  lines(xGrid,MeanFunc(xGrid), col = "red", lwd = 3)
  lines(xGrid, MeanFunc(xGrid) - 1.96*sqrt(diag(SquaredExpKernel(xGrid,xGrid,sigmaF,l))), col = "blue",
  lines(xGrid, MeanFunc(xGrid) + 1.96*sqrt(diag(SquaredExpKernel(xGrid,xGrid,sigmaF,l))), col = "blue",
  title(paste('length scale =',l,', sigmaf =',sigmaF))
}

manipulate(
  plotGPPrior(sigmaF, l, nSim = 10),
  sigmaF = slider(0, 2, step=0.1, initial = 1, label = "SigmaF"),
  l = slider(0, 2, step=0.1, initial = 1, label = "Length scale, l")
)
```

KernLabDemo.R

```r
#########################################################################################################
## Kernlab demo for Gaussian processes regression and classfication
## Author: Mattias Villani, Linkoping University. http://mattiasvillani.com
#########################################################################################################


#############################################################################
###    Prelims: Setting path and installing/loading packages            #####
#############################################################################
setwd('/Users/mv/Dropbox/Teaching/AdvML/GaussianProcess/Code')
#install.packages('kernlab')
#install.packages("AtmRay") # To make 2D grid like in Matlab's meshgrid
library(kernlab)
library(AtmRay)

################################################
###    Messin' around with kernels        #####
################################################
# This is just to test how one evaluates a kernel function
# and how one computes the covariance matrix from a kernel function
X <- matrix(rnorm(12), 4, 3)
Xstar <- matrix(rnorm(15), 5, 3)
ell <- 1
SEkernel <- rbfdot(sigma = 1/(2*ell^2)) # Note how I reparametrize the rbfdot (which is the SE kernel)
SEkernel(1,2) # Just a test - evaluating the kernel in the points x=1 and x'=2
# Computing the whole covariance matrix K from the kernel. Just a test.
K <- kernelMatrix(kernel = SEkernel, x = X, y = Xstar) # So this is K(X,Xstar)

# Own implementation of Matern with nu = 3/2 (See RW book equation 4.17)
# Note that a call of the form kernelFunc <- Matern32(sigmaf = 1, ell = 0.1) returns a kernel FUNCTION.
```

```r
# You can now evaluate the kernel at inputs: kernelFunc(x = 3, y = 4).
# Note also that class(kernelFunc) is of class "kernel", which is a class defined by kernlab.
Matern32 <- function(sigmaf = 1, ell = 1)
{
  rval <- function(x, y = NULL) {
      r = sqrt(crossprod(x-y));
      return(sigmaf^2*(1+sqrt(3)*r/ell)*exp(-sqrt(3)*r/ell))
    }
  class(rval) <- "kernel"
  return(rval)
}

# Testing our own defined kernel function.
X <- matrix(rnorm(12), 4, 3) # Simulating some data
Xstar <- matrix(rnorm(15), 5, 3)
MaternFunc = Matern32(sigmaf = 1, ell = 2) # MaternFunc is a kernel FUNCTION
MaternFunc(c(1,1),c(2,2)) # Evaluating the kernel in x=c(1,1), x'=c(2,2)
# Computing the whole covariance matrix K from the kernel.
K <- kernelMatrix(kernel = MaternFunc, x = X, y = Xstar) # So this is K(X,Xstar)

################################################
###      Regression on the LIDAR data       ###
################################################
lidarData <- read.table('https://raw.githubusercontent.com/STIMALiU/AdvMLCourse/master/GaussianProcess/
                        header = T)
LogRatio <- lidarData$LogRatio
Distance <- lidarData$Distance

# Estimating the noise variance from a third degree polynomial fit
polyFit <- lm(LogRatio ~  Distance + I(Distance^2) + I(Distance^3) )
sigmaNoise = sd(polyFit$residuals)

plot(Distance,LogRatio)

# Fit the GP with built in Square expontial kernel (called rbfdot in kernlab)
ell <- 2
GPfit <- gausspr(Distance, LogRatio, kernel = rbfdot, kpar = list(sigma = 1/(2*ell^2)), var = sigmaNois
meanPred <- predict(GPfit, Distance) # Predicting the training data. To plot the fit.
lines(Distance, meanPred, col="blue", lwd = 2)

# Fit the GP with home made Matern
sigmaf <- 1
ell <- 2
# GPfit <- gausspr(Distance, LogRatio, kernel = Matern32(ell=1)) # NOTE: this also works and is the sam
GPfit <- gausspr(Distance, LogRatio, kernel = Matern32, kpar = list(sigmaf = sigmaf, ell=ell), var = si
meanPred <- predict(GPfit, Distance)
lines(Distance, meanPred, col="purple", lwd = 2)

# Trying another length scale
sigmaf <- 1
ell <- 1
GPfit <- gausspr(Distance, LogRatio, kernel = Matern32, kpar = list(sigmaf = sigmaf, ell=ell), var = si
meanPred <- predict(GPfit, Distance)
```

```
lines(Distance, meanPred, col="green", lwd = 2)

# And now with a different sigmaf
sigmaf <- 0.1
ell <- 2
GPfit <- gausspr(Distance, LogRatio, kernel = Matern32, kpar = list(sigmaf = sigmaf, ell=ell), var = si
meanPred <- predict(GPfit, Distance)
lines(Distance, meanPred, col="black", lwd = 2)

################################################
####       Classification on Iris data      ###
################################################
data(iris)
GPfitIris <- gausspr(Species ~  Sepal.Length + Sepal.Width + Petal.Length + Petal.Width, data=iris)
GPfitIris

# predict on the training set
predict(GPfitIris,iris[,1:4])
table(predict(GPfitIris,iris[,1:4]), iris[,5]) # confusion matrix

# Now using only Sepal.Length and Sepal.Width to classify
GPfitIris <- gausspr(Species ~  Sepal.Length + Sepal.Width, data=iris)
GPfitIris
# predict on the training set
predict(GPfitIris,iris[,1:2])
table(predict(GPfitIris,iris[,1:2]), iris[,5]) # confusion matrix

# Now using only  Petal.Length + Petal.Width to classify
GPfitIris <- gausspr(Species ~ Petal.Length + Petal.Width, data=iris)
GPfitIris
# predict on the training set
predict(GPfitIris,iris[,3:4])
table(predict(GPfitIris,iris[,3:4]), iris[,5]) # confusion matrix

# class probabilities
probPreds <- predict(GPfitIris, iris[,3:4], type="probabilities")
x1 <- seq(min(iris[,3]),max(iris[,3]),length=100)
x2 <- seq(min(iris[,4]),max(iris[,4]),length=100)
gridPoints <- meshgrid(x1, x2)
gridPoints <- cbind(c(gridPoints$x), c(gridPoints$y))

gridPoints <- data.frame(gridPoints)
names(gridPoints) <- names(iris)[3:4]
probPreds <- predict(GPfitIris, gridPoints, type="probabilities")

# Plotting for Prob(setosa)
contour(x1,x2,matrix(probPreds[,1],100,byrow = TRUE), 20, xlab = "Petal.Length", ylab = "Petal.Width", m
points(iris[iris[,5]=='setosa',3],iris[iris[,5]=='setosa',4],col="red")
points(iris[iris[,5]=='virginica',3],iris[iris[,5]=='virginica',4],col="blue")
points(iris[iris[,5]=='versicolor',3],iris[iris[,5]=='versicolor',4],col="green")

# Plotting for Prob(Versicolor)
contour(x1,x2,matrix(probPreds[,2],100,byrow = TRUE), 20, xlab = "Petal.Length", ylab = "Petal.Width", m
```

```
points(iris[iris[,5]=='setosa',3],iris[iris[,5]=='setosa',4],col="red")
points(iris[iris[,5]=='virginica',3],iris[iris[,5]=='virginica',4],col="blue")
points(iris[iris[,5]=='versicolor',3],iris[iris[,5]=='versicolor',4],col="green")


# Plotting for Prob(virginica)
contour(x1,x2,matrix(probPreds[,3],100,byrow = TRUE), 20, xlab = "Petal.Length", ylab = "Petal.Width", n
points(iris[iris[,5]=='setosa',3],iris[iris[,5]=='setosa',4],col="red")
points(iris[iris[,5]=='virginica',3],iris[iris[,5]=='virginica',4],col="blue")
points(iris[iris[,5]=='versicolor',3],iris[iris[,5]=='versicolor',4],col="green")

# Plotting the decision boundaries
meanPred <- matrix(max.col(probPreds),100,byrow = TRUE)
plot(gridPoints,  pch=".", cex=3, col=ifelse(meanPred==1, "red", ifelse(meanPred==2, "green", "blue")))
points(iris[iris[,5]=='setosa',3],iris[iris[,5]=='setosa',4],col="red", cex=10, pch=".")
points(iris[iris[,5]=='virginica',3],iris[iris[,5]=='virginica',4],col="blue",  cex=10, pch=".")
points(iris[iris[,5]=='versicolor',3],iris[iris[,5]=='versicolor',4],col="green",  cex=10, pch=".")

########################################
# Author: Jose M. Peña, jose.m.pena@liu.se
# GP regression on the canadian wages data
########################################

library(kernlab)
CWData <- read.table(
  'https://raw.githubusercontent.com/STIMALiU/AdvMLCourse/master/GaussianProcess/Code/CanadianWages.dat
  header = T)
logWage<-CWData$logWage
age<-CWData$age
age<-(age-mean(age))/sd(age) # Standarize the age

# Estimating the noise variance from a third degree polynomial fit. I() is needed because, otherwise
# age^2 reduces to age in the formula, i.e. age^2 means adding the main effect and the second order
# interaction, which in this case do not exist. See ?I.
polyFit <- lm(logWage ~  age + I(age^2) + I(age^3))
sigmaNoise = sd(polyFit$residuals)
plot(age,logWage)

# Fit the GP with built-in square expontial kernel (called rbfdot in kernlab)
ell <- 0.5
SEkernel <- rbfdot(sigma = 1/(2*ell^2)) # Note the reparametrization
GPfit <- gausspr(age,logWage, kernel = SEkernel, var = sigmaNoise^2)
meanPred <- predict(GPfit, age) # Predicting the training data
lines(age, meanPred, col="red", lwd = 2)

# The implementation of kernlab for the probability and prediction intervals seem to have a bug: The in
# seem to be too wide, e.g. replace 1.96 with 0.1 to see something.
# GPfit <- gausspr(age,logWage, kernel = SEkernel, var = sigmaNoise^2, variance.model = TRUE)
# meanPred <- predict(GPfit, age)
# lines(age, meanPred, col="red", lwd = 2)
# lines(age, meanPred+1.96*predict(GPfit,age, type="sdeviation"),col="blue")
# lines(age, meanPred-1.96*predict(GPfit,age, type="sdeviation"),col="blue")
```

```r
# Probability and prediction interval implementation.
x<-age
xs<-age # XStar
n <- length(x)
Kss <- kernelMatrix(kernel = SEkernel, x = xs, y = xs)
Kxx <- kernelMatrix(kernel = SEkernel, x = x, y = x)
Kxs <- kernelMatrix(kernel = SEkernel, x = x, y = xs)
Covf = Kss-t(Kxs)%*%solve(Kxx + sigmaNoise^2*diag(n), Kxs) # Covariance matrix of fStar

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

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