Lecture 9

Gaussian Process regression, Part 1

Gaussian Process Regression (GPR) is a powerful, non-parametric Bayesian approach to regression. Unlike traditional parametric regression methods, which assume a specific functional form for the underlying data (such as linear or polynomial), GPR makes predictions based on the assumption that the data can be represented as a sample from a multivariate Gaussian distribution. This approach models the underlying function as a distribution over functions, rather than as a single fixed function.

In GPR, predictions at new points are derived by considering the correlation (or covariance) between these new points and the observed data points. This correlation is defined by a kernel function, which encodes assumptions about the smoothness, periodicity, and other properties of the underlying function. Common kernel functions include the Radial Basis Function (RBF), Matérn, and linear kernels. The strength of GPR lies in its ability to provide not only a prediction but also a measure of uncertainty associated with that prediction, which is useful in many practical applications. However, GPR can be computationally expensive, especially for large datasets, due to the inversion of large covariance matrices, though various approximation methods have been developed to mitigate this challenge.

We are going to use R to build up some examples from scratch to better understand how it works. In geostatistics, this process is also called kriging (although, to implement kriging in R, most packages will require a spatial data frame, since that's context in which it is most commonly used, while packages that implement Gaussian process regression (so named) do not require a spatial data frame.

The Gaussian process regression examples we'll be building from use a Euclidean distance function. One of the places where we can customize our modeling function will be in the way distances are calculated. However, for the time being, we'll stick with the standard metric.

Another element of Gaussian process regression is the kernel: a function used to find the covariance matrix we use in the model. It's common to use an exponential kernel (with negative distance in the exponent) to indicate that the further two points are from each other, the less related they are (decaying exponentially). In this way, nearby points have more influence on the model than far away points. However, the specifics of this kernel can also be customized. In geostatistics, inverse weights are sometimes used and other modelling choices are available. Again, for now, we'll stick with the standard metrics, but this is another place where modifications to the general framework can be made if desired.

We'll start with some sample data, and create a matrix of the results. You'll need to install the library plgp for the distance function, or use dist() from base R. Finally, we add a small epsilon to make the computations go a bit more smoothly. Sometimes our examples can have issues related to collinearity, and the small epsilon can eliminate that issue.

```
n <- 100
X <- matrix(seq(0, 10, length=n), ncol=1)
library(plgp)
D <- distance(X)
eps <- sqrt(.Machine$double.eps)
Sigma <- exp(-D) + diag(eps, n)
```

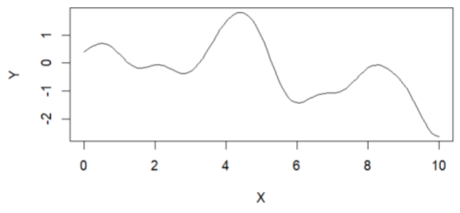
We can create a matrix from these elements and generate our random function that goes through the points. (Note: you may need to install the mvtnorm library here, first.)

```
library(mvtnorm)
Y <- rmvnorm(1, sigma=Sigma)
```

That's it! We've generated a finite realization of a random function under a GP prior with a particular covariance structure. Now all that's left is visualization.

```
plot(X, Y, type="l")
```

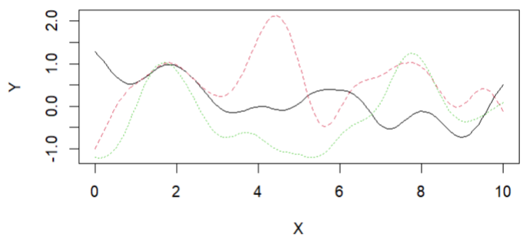
Without setting the random seed, our examples will look different from each other, but this is what mine looks like.



Short distances are highly correlated, long ones essentially uncorrelated. Because the base variance is 1, we get a range of y-values typical of a Gaussian distribution (most values between -2 and 2).

We can generate additional examples of these random functions and plot them on the same graph.

```
Y <- rmvnorm(3, sigma=Sigma)
matplot(X, t(Y), type="l", ylab="Y")
```

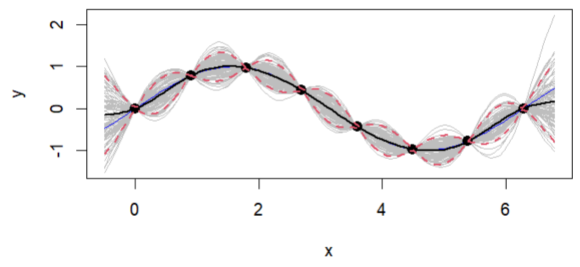


This is a good place to experiment with different distance metrics and different kernels, before the code gets more complex.

Let's consider how this works with a simple one-dimensional prediction example. See the reading from this week for an explanation of each specific step.

```
n <- 8
X <- matrix(seq(0, 2*pi, length=n), ncol=1)
y <- sin(X)
D <- distance(X)
Sigma <- exp(-D) + diag(eps, ncol(D))
```

```
#generate Y
XX <- matrix(seq(-0.5, 2*pi + 0.5, length=100), ncol=1)
DXX <- distance(XX)
SXX <- exp(-DXX) + diag(eps, ncol(DXX))

DX <- distance(XX, X)
SX <- exp(-DX)

Si <- solve(Sigma)
mup <- SX %*% Si %*% y
Sigmap <- SXX - SX %*% Si %*% t(SX)

YY <- rmvnorm(100, mup, Sigmap)

q1 <- mup + qnorm(0.05, 0, sqrt(diag(Sigmap)))
q2 <- mup + qnorm(0.95, 0, sqrt(diag(Sigmap)))

matplot(XX, t(YY), type="l", col="gray", lty=1, xlab="x", ylab="y")
points(X, y, pch=20, cex=2)
lines(XX, sin(XX), col="blue")
lines(XX, mup, lwd=2)
lines(XX, q1, lwd=2, lty=2, col=2)
lines(XX, q2, lwd=2, lty=2, col=2)
```

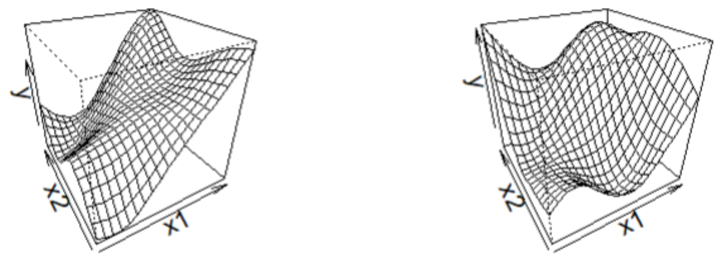


Note that Gaussian process regression is Bayesian, which works a little differently than frequentist approaches. The grey lines are other functions that pass through the data points. The black line is the mean of all the functions. The red dotted lines are the 95% confidence intervals (95% of the functions as within these bounds).

Let’s look at an example in a higher dimension.

```
nx <- 20
x <- seq(0, 2, length=nx)
X <- expand.grid(x, x)

D <- distance(X)
Sigma <- exp(-D) + diag(eps, nrow(X))
```

```
Y <- rmvnorm(2, sigma=Sigma)

par(mfrow=c(1,2))
persp(x, x, matrix(Y[1,], ncol=nx), theta=-30, phi=30, xlab="x1",
  ylab="x2", zlab="y")
persp(x, x, matrix(Y[2,], ncol=nx), theta=-30, phi=30, xlab="x1",
  ylab="x2", zlab="y")
```



Let's try looking at some data generated from the function $y(\vec{x}) = x_1 e^{-x_1^2 - x_2^2}$. This function is nonlinear with some "bumpy" areas and some smooth ones.

```
library(lhs)
X <- randomLHS(40, 2)
X[,1] <- (X[,1] - 0.5)*6 + 1
X[,2] <- (X[,2] - 0.5)*6 + 1
y <- X[,1]*exp(-X[,1]^2 - X[,2]^2)
```

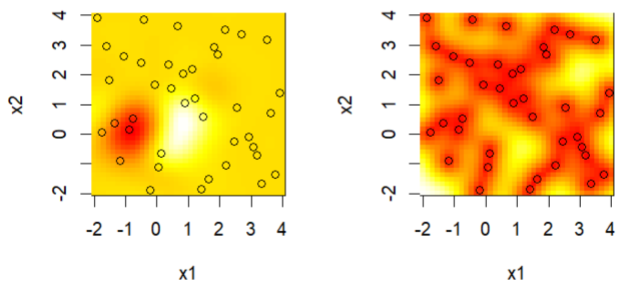
The inputs Will be mapped on a scale of [-2,4], and then we'll create a 40x40 grid of inputs.

```
xx <- seq(-2, 4, length=40)
XX <- expand.grid(xx, xx)
```

Now we repeat much of our prior process for the modeling.

```
D <- distance(X)
Sigma <- exp(-D)

DXX <- distance(XX)
SXX <- exp(-DXX) + diag(eps, ncol(DXX))
DX <- distance(XX, X)
SX <- exp(-DX)

Si <- solve(Sigma)
mup <- SX %*% Si %*% y
Sigmap <- SXX - SX %*% Si %*% t(SX)

sdp <- sqrt(diag(Sigmap))
```

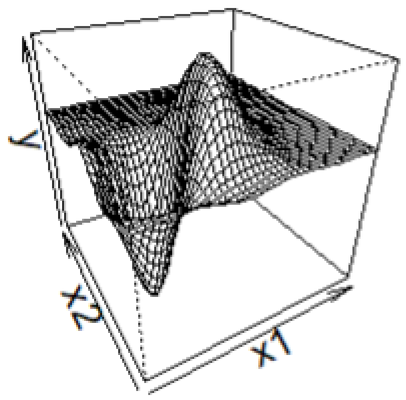
```
par(mfrow=c(1,2))
cols <- heat.colors(128)
image(xx, xx, matrix(mup, ncol=length(xx)), xlab="x1", ylab="x2", col=cols)
points(X[,1], X[,2])
image(xx, xx, matrix(sdp, ncol=length(xx)), xlab="x1", ylab="x2", col=cols)
points(X[,1], X[,2])
```



The left graph is the mean, the right graph is the standard deviation. In both graphs, white represents high values and red low ones. The highest and lowest mean values are near the origin (on either side of (0,0)), while the highest standard deviation values are further from the test points and smaller nearest them.

The 3D graph of the model might help.

```
persp(xx, xx, matrix(mup, ncol=40), theta=-30, phi=30, xlab="x1",  ylab="x2", zlab="y")
```

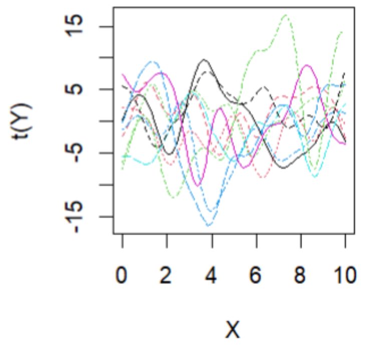


Gaussian process regression has several hyperparameters that can also be set. Let's examine those.

```
n <- 100
X <- matrix(seq(0, 10, length=n), ncol=1)
D <- distance(X)

C <- exp(-D) + diag(eps, n)
```

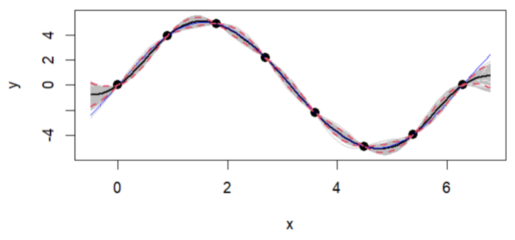
```
tau2 <- 25
Y <- rmvnorm(10, sigma=tau2*C)

matplot(X, t(Y), type="l")
```

The choice of τ affects the amplitude of the “wiggle” in y. Before the amplitude was confined to [-2,2], but here we are more like [-10,10].



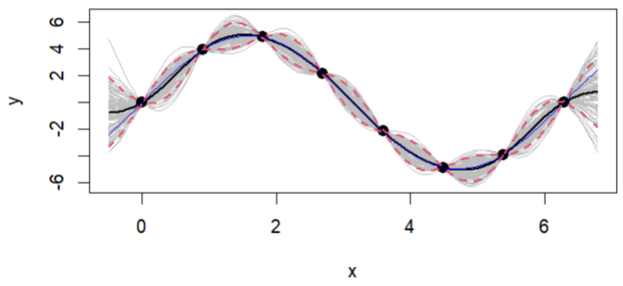
Let’s look at a model of a sine function.

```
n <- 8
X <- matrix(seq(0, 2*pi, length=n), ncol=1)
y <- 5*sin(X)

D <- distance(X)
Sigma <- exp(-D)
XX <- matrix(seq(-0.5, 2*pi + 0.5, length=100), ncol=1)
DXX <- distance(XX)
SXX <- exp(-DXX) + diag(eps, ncol(DXX))
DX <- distance(XX, X)
SX <- exp(-DX)
Si <- solve(Sigma);
mup <- SX %*% Si %*% y
Sigmap <- SXX - SX %*% Si %*% t(SX)

YY <- rmvnorm(100, mup, Sigmap)
q1 <- mup + qnorm(0.05, 0, sqrt(diag(Sigmap)))
q2 <- mup + qnorm(0.95, 0, sqrt(diag(Sigmap)))
matplot(XX, t(YY), type="l", col="gray", lty=1, xlab="x", ylab="y")
points(X, y, pch=20, cex=2)
lines(XX, mup, lwd=2)
lines(XX, 5*sin(XX), col="blue")
lines(XX, q1, lwd=2, lty=2, col=2)
lines(XX, q2, lwd=2, lty=2, col=2)
```



Failing to correctly estimate the τ (using the wrong prior assumptions) can produce an underestimate of the error (as here), particularly outside the range of the data. One solution is to estimate the τ from the data prior to doing the rest of the calculations.

```
CX <- SX
Ci <- Si
CXX <- SXX
tau2hat <- drop(t(y) %*% Ci %*% y / length(y))

2*sqrt(tau2hat)

mup2 <- CX %*% Ci %*% y
Sigmap2 <- tau2hat*(CXX - CX %*% Ci %*% t(CX))

YY <- rmvnorm(100, mup2, Sigmap2)
q1 <- mup + qnorm(0.05, 0, sqrt(diag(Sigmap2)))
q2 <- mup + qnorm(0.95, 0, sqrt(diag(Sigmap2)))

matplot(XX, t(YY), type="l", col="gray", lty=1, xlab="x", ylab="y")
points(X, y, pch=20, cex=2)
lines(XX, mup, lwd=2)
lines(XX, 5*sin(XX), col="blue")
lines(XX, q1, lwd=2, lty=2, col=2); lines(XX, q2, lwd=2, lty=2, col=2)
```



Alternative distance measurements, such as the Mahalanobis distance may improve the quality of predictions.

```
score <- function(Y, mu, Sigma, mah=FALSE)
 {
  Ymmu <- Y - mu
  Sigmai <- solve(Sigma)
  mahdist <- t(Ymmu) %*% Sigmai %*% Ymmu
 if(mah) return(sqrt(mahdist))
 return (- determinant(Sigma, logarithm=TRUE)$modulus - mahdist)
}

Ytrue <- 5*sin(XX)
df <- data.frame(score(Ytrue, mup, Sigmap, mah=TRUE),
  score(Ytrue, mup2, Sigmap2, mah=TRUE))
colnames(df) <- c("tau2=1", "tau2hat")
df
```

So far, we have been doing interpolating rather than regression, since we have assumed no noise on any of our inputs. To do that, we need to estimate a nugget, which will involve maximum likelihood estimation and numerical optimization. Optimization libraries are mostly set up for minimization, and we need maximization, so the code will be applied to the negative likelihood. Some additional tweaks for this example are the optimization is done on the distances, rather than x, and a counter is included to check the number of iterations required.

```
nlg <- function(g, D, Y)
 {
  n <- length(Y)
  K <- exp(-D) + diag(g, n)
  Ki <- solve(K)
  ldetK <- determinant(K, logarithm=TRUE)$modulus
  ll <- - (n/2)*log(t(Y) %*% Ki %*% Y) - (1/2)*ldetK
  counter <<- counter + 1
  return(-ll)
 }
```

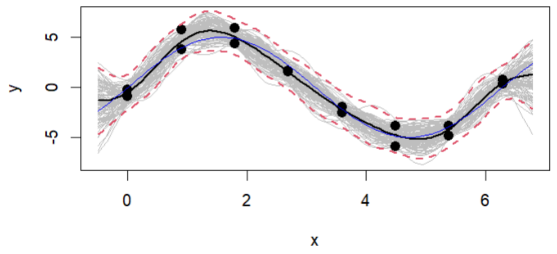
Let’s look at our sine example, but with noise added.

```
X <- rbind(X, X)
n <- nrow(X)
y <- 5*sin(X) + rnorm(n, sd=1)
D <- distance(X)

counter <- 0
g <- optimize(nlg, interval=c(eps, var(y)), D=D, Y=y)$minimum
g

K <- exp(-D) + diag(g, n)
Ki <- solve(K)
tau2hat <- drop(t(y) %*% Ki %*% y / n)
c(tau=sqrt(tau2hat), sigma=sqrt(tau2hat*g))
```

We should expect values close to 2.5 and 1 respectively for τ, σ. Now we have what we need to repeat our analysis.

```
DX <- distance(XX, X)
```

```
KX <- exp(-DX)
KXX <- exp(-DXX) + diag(g, nrow(DXX))

mup <- KX %*% Ki %*% y
Sigmap <- tau2hat*(KXX - KX %*% Ki %*% t(KX))
q1 <- mup + qnorm(0.05, 0, sqrt(diag(Sigmap)))
q2 <- mup + qnorm(0.95, 0, sqrt(diag(Sigmap)))

Sigma.int <- tau2hat*(exp(-DXX) + diag(eps, nrow(DXX)) - KX %*% Ki %*% t(KX))
YY <- rmvnorm(100, mup, Sigma.int)

matplot(XX, t(YY), type="l", lty=1, col="gray", xlab="x", ylab="y")
points(X, y, pch=20, cex=2)
lines(XX, mup, lwd=2)
lines(XX, 5*sin(XX), col="blue")
lines(XX, q1, lwd=2, lty=2, col=2)
lines(XX, q2, lwd=2, lty=2, col=2)
```



While there is now some decrease in variance near the observed points, it does not go to zero as we saw in the examples that assumed no noise.

We can also use derivative-based optimization methods to obtain a faster convergence, which may not matter much in a case as small as these initial examples, but could matter more in situations with more observations and more variables.

```
gnlg <- function(g, D, Y)
 {
  n <- length(Y)
  K <- exp(-D) + diag(g, n)
  Ki <- solve(K)
  KiY <- Ki %*% Y
  dll <- (n/2) * t(KiY) %*% KiY / (t(Y) %*% KiY) - (1/2)*sum(diag(Ki))
  return(-dll)
 }

counter <- 0
out <- optim(0.1*var(y), nlg, gnlg, method="L-BFGS-B", lower=eps,
  upper=var(y), D=D, Y=y)
c(g, out$par)
c(out$counts, actual=counter)
```

We should obtain a g value similar to the previous case with fewer iterations. The rest of the calculation would proceed as before since this only estimates a parameter, not the entire model.

We've been looking a particular rate for the correlation to decay, but this is something we can adjust. If the correlation remains strong for longer, we get a smoother model. If the decay happens more quickly, we can get a more wiggly model. Let's look at how we make this adjustment to the code. This is now has more than one variable to optimize, so we'll need to take that into account when we choose our optimization method.

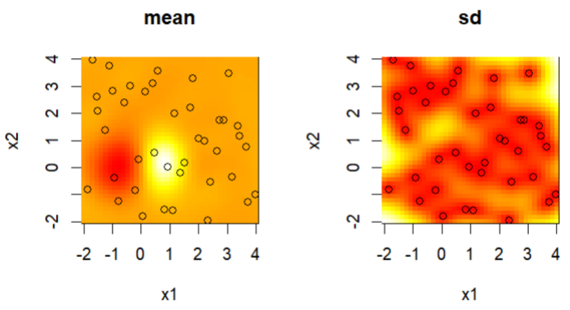
```
nl <- function(par, D, Y)
 {
  theta <- par[1]                    ## change 1
  g <- par[2]
  n <- length(Y)
  K <- exp(-D/theta) + diag(g, n)         ## change 2
  Ki <- solve(K)
  ldetK <- determinant(K, logarithm=TRUE)$modulus
  ll <- - (n/2)*log(t(Y) %*% Ki %*% Y) - (1/2)*ldetK
  counter <<- counter + 1
  return(-ll)
 }

library(lhs)
X2 <- randomLHS(40, 2)
X2 <- rbind(X2, X2)
X2[,1] <- (X2[,1] - 0.5)*6 + 1
X2[,2] <- (X2[,2] - 0.5)*6 + 1
y2 <- X2[,1]*exp(-X2[,1]^2 - X2[,2]^2) + rnorm(nrow(X2), sd=0.01)

D <- distance(X2)
counter <- 0
out <- optim(c(0.1, 0.1*var(y2)), nl, method="L-BFGS-B", lower=eps,
  upper=c(10, var(y2)), D=D, Y=y2)
out$par

brute <- c(out$counts, actual=counter)
brute
```

Another way of approaching this calculation:

```
gradnl <- function(par, D, Y)
 {
  ## extract parameters
  theta <- par[1]
  g <- par[2]

  ## calculate covariance quantities from data and parameters
  n <- length(Y)
  K <- exp(-D/theta) + diag(g, n)
  Ki <- solve(K)
  dotK <- K*D/theta^2
  KiY <- Ki %*% Y

  ## theta component
  dlltheta <- (n/2) * t(KiY) %*% dotK %*% KiY / (t(Y) %*% KiY) -
    (1/2)*sum(diag(Ki %*% dotK))

  ## g component
  dllg <- (n/2) * t(KiY) %*% KiY / (t(Y) %*% KiY) - (1/2)*sum(diag(Ki))

  ## combine the components into a gradient vector
  return(-c(dlltheta, dllg))
 }

counter <- 0
outg <- optim(c(0.1, 0.1*var(y2)), nl, gradnl, method="L-BFGS-B",
  lower=eps, upper=c(10, var(y2)), D=D, Y=y2)
rbind(grad=outg$par, brute=out$par)

rbind(grad=c(outg$counts, actual=counter), brute)
```

And now the components we need to make the predictions.

```
K <- exp(- D/outg$par[1]) + diag(outg$par[2], nrow(X2))
Ki <- solve(K)
tau2hat <- drop(t(y2) %*% Ki %*% y2 / nrow(X2))

gn <- 40
xx <- seq(-2, 4, length=gn)
XX <- expand.grid(xx, xx)
DXX <- distance(XX)
KXX <- exp(-DXX/outg$par[1]) + diag(outg$par[2], ncol(DXX))
DX <- distance(XX, X2)
KX <- exp(-DX/outg$par[1])

mup <- KX %*% Ki %*% y2
Sigmap <- tau2hat*(KXX - KX %*% Ki %*% t(KX))
sdp <- sqrt(diag(Sigmap))

par(mfrow=c(1,2))
image(xx, xx, matrix(mup, ncol=gn), main="mean", xlab="x1", ylab="x2", col=cols)
points(X2)
image(xx, xx, matrix(sdp, ncol=gn), main="sd", xlab="x1",  ylab="x2", col=cols)
points(X2)
```



Splines are often used for low-dimensional models instead of GPs, but let's do a comparison with one particular implementation of multi-dimensional splines in R.

```
fried <- function(n=50, m=6)
 {
  if(m < 5) stop("must have at least 5 cols")
  X <- randomLHS(n, m)
  Ytrue <- 10*sin(pi*X[,1]*X[,2]) + 20*(X[,3] - 0.5)^2 + 10*X[,4] + 5*X[,5]
  Y <- Ytrue + rnorm(n, 0, 1)
  return(data.frame(X, Y, Ytrue))
 }
```

The "surface" we are modeling is $E\{Y(x)\} = 10sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 - 5x_5$.

```
m <- 7
n <- 200
nprime <- 1000
data <- fried(n + nprime, m)
X <- as.matrix(data[1:n,1:m])
y <- drop(data$Y[1:n])
XX <- as.matrix(data[(n + 1):(n + nprime),1:m])
yy <- drop(data$Y[(n + 1):(n + nprime)])
yytrue <- drop(data$Ytrue[(n + 1):(n + nprime)])

D <- distance(X)
out <- optim(c(0.1, 0.1*var(y)), nl, gradnl, method="L-BFGS-B", lower=eps,
        upper=c(10, var(y)), D=D, Y=y)
out

K <- exp(- D/out$par[1]) + diag(out$par[2], nrow(D))
Ki <- solve(K)
tau2hat <- drop(t(y) %*% Ki %*% y / nrow(D))

DXX <- distance(XX)
KXX <- exp(-DXX/out$par[1]) + diag(out$par[2], ncol(DXX))
DX <- distance(XX, X)
KX <- exp(-DX/out$par[1])
```

```
mup <- KX %*% Ki %*% y
Sigmap <- tau2hat*(KXX - KX %*% Ki %*% t(KX))
```

Visualizing is a problem, but we can look at the RMSE.

```
rmse <- c(gpiso=sqrt(mean((yytrue - mup)^2)))
rmse
```

Let’s compare with the spline model.

```
library(mda)
fit.mars <- mars(X, y)
p.mars <- predict(fit.mars, XX)

rmse <- c(rmse, mars=sqrt(mean((yytrue - p.mars)^2)))
rmse
```

In most cases, you’d expect the GP model to be a bit better, but the splines will do better about 5% of the time (indeed, when I ran this example, the splines did a little better), but you can rerun the models a couple of times (redo the random errors), and see what happens.

As with previous cases, we can adjust the distance parameter decay rate to adjust the model for better results.

```
nlsep <- function(par, X, Y)
 {
  theta <- par[1:ncol(X)]
  g <- par[ncol(X)+1]
  n <- length(Y)
  K <- covar.sep(X, d=theta, g=g)
  Ki <- solve(K)
  ldetK <- determinant(K, logarithm=TRUE)$modulus
  ll <- - (n/2)*log(t(Y) %*% Ki %*% Y) - (1/2)*ldetK
  counter <<- counter + 1
  return(-ll)
 }

tic <- proc.time()[3]
counter <- 0
out <- optim(c(rep(0.1, ncol(X)), 0.1*var(y)), nlsep, method="L-BFGS-B",
  X=X, Y=y, lower=eps, upper=c(rep(10, ncol(X)), var(y)))
toc <- proc.time()[3]
out$par

brute <- c(out$counts, actual=counter)
brute

toc - tic
```

```
gradnlsep <- function(par, X, Y)
 {
  theta <- par[1:ncol(X)]
  g <- par[ncol(X)+1]
  n <- length(Y)
  K <- covar.sep(X, d=theta, g=g)
  Ki <- solve(K)
  KiY <- Ki %*% Y

  ## loop over theta components
  dlltheta <- rep(NA, length(theta))
  for(k in 1:length(dlltheta)) {
    dotK <- K * distance(X[,k])/(theta[k]^2)
    dlltheta[k] <- (n/2) * t(KiY) %*% dotK %*% KiY / (t(Y) %*% KiY) -
      (1/2)*sum(diag(Ki %*% dotK))
  }

  ## for g
  dllg <- (n/2) * t(KiY) %*% KiY / (t(Y) %*% KiY) - (1/2)*sum(diag(Ki))

  return(-c(dlltheta, dllg))
 }

tic <- proc.time()[3]
counter <- 0
outg <- optim(c(rep(0.1, ncol(X)), 0.1*var(y)), nlsep, gradnlsep,
  method="L-BFGS-B", lower=eps, upper=c(rep(10, ncol(X)), var(y)), X=X, Y=y)
toc <- proc.time()[3]
thetahat <- rbind(grad=outg$par, brute=out$par)
colnames(thetahat) <- c(paste0("d", 1:ncol(X)), "g")
thetahat

rbind(grad=c(outg$counts, actual=counter), brute)

toc - tic
```

Let's compare our models again.

```
K <- covar.sep(X, d=outg$par[1:ncol(X)], g=outg$par[ncol(X)+1])
Ki <- solve(K)
tau2hat <- drop(t(y) %*% Ki %*% y / nrow(X))
KXX <- covar.sep(XX, d=outg$par[1:ncol(X)], g=outg$par[ncol(X)+1])
KX <- covar.sep(XX, X, d=outg$par[1:ncol(X)], g=0)
mup2 <- KX %*% Ki %*% y
Sigmap2 <- tau2hat*(KXX - KX %*% Ki %*% t(KX))

rmse <- c(rmse, gpsep=sqrt(mean((yytrue - mup2)^2)))
rmse
```

```
scores <- c(gp=score(yy, mup, Sigmap), mars=NA,
  gpsep=score(yy, mup2, Sigmap2))
scores
```

There are lots of options for packages that are capable of implementing what we've done here, and each comes with some level of customization. We've looked at kernlab, and RobustGaSP before. Let's look at one more here.

```
library(laGP)
tic <- proc.time()[3]
gpi <- newGPsep(X, y, d=0.1, g=0.1*var(y), dK=TRUE)

mle <- mleGPsep(gpi, param="both", tmin=c(eps, eps), tmax=c(10, var(y)))
toc <- proc.time()[3]

thetahat <- rbind(grad=outg$par, brute=out$par, laGP=mle$theta)
colnames(thetahat) <- c(paste0("d", 1:ncol(X)), "g")
thetahat

toc - tic

rbind(grad=c(outg$counts, actual=counter), brute,
  laGP=c(mle$its, mle$its, NA))

p <- predGPsep(gpi, XX)

rmse <- c(rmse, laGP=sqrt(mean((yytrue - p$mean)^2)))
scores <- c(scores, laGP=score(yy, p$mean, p$Sigma))
rbind(rmse, scores)

deleteGPsep(gpi)
```

The end of this section (5.2) of the Surrogates book goes through some additional comparisons with packages vs. by-hand constructions.

We'll continue the discussion of Gaussian processes in the next lecture.

Resources:

1. https://bookdown.org/rbg/surrogates/chap5.html
2.