Surrogates 7020

Chapter 5(Part 2): Gaussian Process Regression

Dr. Alex Bledar Konomi

Department of Mathematical Sciences
University of Cincinnati

GP hyperparameters

- The GP is called (most of the times) a non-parametric model or regression.
- All this business about nonparametric regression and here we are introducing parameters. (but with a different name: hyperparamters)
- How can one have hyperparameters without parameters to start with, or at least to somehow distinguish from?
- To make things even more confusing, we go about learning those hyperparameters in the usual way, by optimizing something, just like parameters. These hyperparameters are more of a fine tuning.
- The fact is that you can express the GPs with latent variables (which we are not going to cover) and then model the latent variable covariance and not the mean.

GP hyperparameters

Suppose you want your GP prior to generate random functions with an amplitude larger than two. You could introduce a scale parameter τ^2 and then take $\boldsymbol{\Sigma} = \tau^2 \boldsymbol{C}_n$. Here $\boldsymbol{C}_n$ is basically the same as our $\boldsymbol{\Sigma}_n$ from before: a correlation function for which $C(x, x) = 1$ and $C(x, x') < 1$ for $x \neq x'$, and positive definite; for example

$$C(\boldsymbol{x}, \boldsymbol{x}') = \exp\{-||\boldsymbol{x} - \boldsymbol{x}'||^2\}.$$

But we need a more nuanced notion of covariance to allow more flexibility on scale, so we're re-parameterizing a bit. Now our MVN generator looks like:

$$\boldsymbol{Y} \sim \mathcal{N}_n(0, \tau^2 \boldsymbol{C}_n)$$

Write down the likelihood! Maximize the (log) likelihood with respect to τ^2, just differentiate and solve.

Bayesian Approach

How would this analysis change if we were to take a Bayesian approach? A homework exercise (5.5) invites the curious reader to investigate the form of the posterior under prior $\tau^2 \sim IG(a/2, b/2)$. For example, what happens when $a = b = 0$ which is equivalent to $p(\tau^2) \propto 1/\tau^2$, a so-called reference prior in this context (Berger, De Oliveira, and Sansó 2001; Berger, Bernardo, and Sun 2009)?

Prediction Distribution

- Estimate of scale $\hat{\tau}^2$ in hand, we may simply "plug it in" to the predictive equations.
- Technically, when you estimate a variance and plug it into a (multivariate) Gaussian, you're turning that Gaussian into a (multivariate) Student-t, in this case with n degrees of freedom (DoF). (There's no loss of DoF when the mean is assumed to be zero.) For details, see for example Gramacy and Polson (2011).

- For now, presume that n is large enough so that this distinction doesn't matter. So to summarize, we have the following scale-adjusted (approximately) MVN predictive equations:

$$\boldsymbol{Y}(\mathcal{X})|\boldsymbol{D}_n \sim \mathcal{N}_{n'}(\boldsymbol{\mu}(\mathcal{X}), \tau^2 C(\mathcal{X}))$$

$$\boldsymbol{\mu}(\mathcal{X}) = C(\mathcal{X}, \boldsymbol{X}_n)\boldsymbol{C}_n^{-1}\boldsymbol{Y}_n$$

$$\Sigma(\mathcal{X}) = \hat{\tau}^2[C(\mathcal{X}, \mathcal{X}) - C(\mathcal{X}, \boldsymbol{X}_n)\boldsymbol{C}_n^{-1}C(\mathcal{X}, \boldsymbol{X}_n)^T]$$

Application

```
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)


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

Application

```
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)
```

Noise and nuggets

- We've been saying "regression" for a while, but actually interpolation is a more apt description. Regression is about extracting signal from noise, or about smoothing over noisy data, and so far our example training data have no noise.
- By inspecting a GP prior, in particular its correlation structure $C(\boldsymbol{x}, \boldsymbol{x}')$, it's clear that the current setup precludes idiosyncratic behavior because correlation decays smoothly as a function of distance. Observe that $C(\boldsymbol{x}, \boldsymbol{x}') \longrightarrow 1$ as $\boldsymbol{x} \longrightarrow \boldsymbol{x}'$, implying that the closer $\boldsymbol{x}$ is to $\boldsymbol{x}'$ the higher the correlation, until correlation is perfect, which is what "connects the dots" when conditioning on data and deriving the predictive distribution.
- We must introduce a discontinuity between diagonal and off-diagonal entries in the correlation matrix $\boldsymbol{C}_n$ to smooth over noise.

Correlation Form with Nugget

$$K(\boldsymbol{x}, \boldsymbol{x}') = C(\boldsymbol{x}, \boldsymbol{x}') + g\delta(\boldsymbol{x}, \boldsymbol{x}').$$

- Above, $g > 0$ is a new hyperparameter called the nugget (or sometimes nugget effect), which determines the size of the discontinuity as $\boldsymbol{x}' \longrightarrow \boldsymbol{x}$. The function δ is more like the Kronecker delta, although the way it's written above makes it look like the Dirac delta. Observe that g generalizes Neal' s ϵ jitter.
- Neither delta is perfect in terms of describing what to do in practice. The simplest, correct description, of how to break continuity is to only add g on a diagonal – when indices of $\boldsymbol{x}$ are the same, not simply for identical values – and nowhere else.
- Never add g to an off-diagonal correlation even if that correlation is based on zero distances: i.e., identical $\boldsymbol{x}$ and $\boldsymbol{x}'$-values.

Covariance Representation

Specifically, $K(xi, xj) = C(\boldsymbol{x}_i, \boldsymbol{x}_j)$ when $i \neq j$, even if $\boldsymbol{x}_i = \boldsymbol{x}_j$; only $K(\boldsymbol{x}_i, \boldsymbol{x}_i) = C(\boldsymbol{x}_i, \boldsymbol{x}_i) + g$. This leads to the following representation of the data-generating mechanism.

$$\boldsymbol{Y} \sim \mathcal{N}_n(0, \tau^2 \boldsymbol{K}_n)$$

Unfolding terms, covariance matrix $\boldsymbol{\Sigma}$ contains entries $\boldsymbol{\Sigma}^{ij} = \tau^2(C(\boldsymbol{x}_i, \boldsymbol{x}_j) + g\delta(\boldsymbol{x}_i, \boldsymbol{x}_j))$. Or in other words:

$$\boldsymbol{\Sigma} = \tau^2(\boldsymbol{C} + g\mathbb{I}_n)$$

Latent Variable Representation

This is operationally equivalent to positing the following model:

$$Y(\boldsymbol{x}) = w(\boldsymbol{x}) + \epsilon,$$

where $w(\boldsymbol{x}) \sim GP$ with scale τ^2, i.e., $\boldsymbol{W} \sim \mathcal{N}_n(0, \tau^2 \boldsymbol{C}_n)$, and ϵ is independent Gaussian noise with variance $\tau^2 g$, i.e., ϵ iid $\sim \mathcal{N}(0, \tau^2 g)$.

Latent Variable Representation

A more aesthetically pleasing model might instead use $\boldsymbol{w}(\boldsymbol{x}) \sim GP$ with scale τ^2, i.e., $\boldsymbol{W} \sim \mathcal{N}_n(0, \tau^2 \boldsymbol{C}_n)$, and where $\epsilon(\boldsymbol{x})$ is iid Gaussian noise with variance σ^2, i.e., $\epsilon(\boldsymbol{x}) iid \sim \mathcal{N}(0, \sigma^2)$.

- An advantage of this representation is two totally "separate" hyperparameters, with one acting to scale noiseless spatial correlations, and another determining the magnitude of white noise. Those two formulations are actually equivalent. There's a 1:1 mapping between the two.
- Many researchers prefer the latter to the former on intuition grounds. But inference in the latter is harder. Conditional on g, $\hat{\tau}^2$ is available in closed form, which we'll show momentarily.

Inference

- Conditional on g, $\hat{\tau}^2$ is available in closed form, which we'll show momentarily. Conditional on σ^2, numerical methods are required for $\hat{\tau}^2$.
- Recall that $\boldsymbol{C}_n$ is an $n \times n$ matrix of inverse exponentiated pairwise squared Euclidean distances. How, then, to estimate two hyperparameters: scale τ^2 and nugget g? Again, we have all the usual suspects (MoM, likelihood, CV, variogram) but likelihood-based methods are by far most common.

Profile Likelihood of the nugget

First, suppose that g is known. The MLE of τ^2 given a fixed g is

$$\hat{\tau}^2 = \frac{\boldsymbol{Y}_n^T \boldsymbol{K}_n^{-1} \boldsymbol{Y}_n}{n} = \frac{\boldsymbol{Y}_n^T (\boldsymbol{C}_n + g\mathbb{I}_n)^{-1} \boldsymbol{Y}_n}{n}.$$

Plug $\hat{\tau}^2$ back into our log likelihood to get a concentrated (or profile) log likelihood involving just the remaining parameter g.

$$\begin{aligned} l(g) &= -\frac{n}{2} log 2\pi - \frac{n}{2} log(\tau^2) - \frac{1}{2} log|\boldsymbol{K}_n| - \frac{1}{2\tau^2} \boldsymbol{Y}_n^T \boldsymbol{K}_n^{-1} \boldsymbol{Y}_n \quad (1) \\ &= c - \frac{n}{2} \log \boldsymbol{Y}_n^T \boldsymbol{K}_n^{-1} \boldsymbol{Y}_n - \frac{1}{2} \log |\boldsymbol{K}_n| \quad (2) \end{aligned}$$

Maximizing $l(g)$ requires numerical methods.

UNIVERSITY OF Cincinnati

Profile Likelihood for Optimization

The simplest thing to do is throw it into optimize and let a polished library do all the work. Since most optimization libraries prefer to minimize, we'll code up $-l(g)$ in R. The nlg function below doesn't directly work on $\boldsymbol{X}$ inputs, rather through distances D. This is slightly more efficient since distances can be pre-calculated, rather than re-calculated in each evaluation for new g.

```
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)
 }

$
```

Optimization

The *counter* is there for comparing alternatives on efficiency grounds in numerical optimization, via the number of times our likelihood objective function is evaluated. Although optimization libraries often provide iteration counts on output, sometimes that report can misrepresent the actual number of objective function calls.

Example Optimization

```
eps <- sqrt(.Machine$double.eps)
n <- 8
X <- matrix(seq(0, 2*pi, length=n), ncol=1)
X <- rbind(X, X)
n <- nrow(X)
y <- 5*sin(X) + rnorm(n, sd=1)
D <- distance(X)
####################
counter <- 0
g <- optimize(nlg, interval=c(eps, var(y)), D=D, Y=y)$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))
```

Example Optimization

```
XX <- matrix(seq(-0.5, 2*pi + 0.5, length=100), ncol=1)
DX <- distance(XX, X)
DXX <- distance(XX)
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)
```

Derivative-based hyperparameter optimization

- It can be unsatisfying to brute-force an optimization for a hyperparameter like g, even though $1d$ solving with optimize is often superior to cleverer methods. Can we improve upon the number of evaluations?
- Differentiating $l(g)$ involves pushing the chain rule through the inverse of covariance matrix $\boldsymbol{K}_n$ and its determinant, which is where hyperparameter g is involved. The following identities, which are framed for an arbitrary parameter ϕ, will come in handy.

$$\frac{\partial \boldsymbol{K}_n^{-1}}{\partial \phi} = -\boldsymbol{K}_n^{-1} \frac{\partial \boldsymbol{K}_n}{\partial \phi} \boldsymbol{K}_n$$

and

$$\frac{\partial \log |\boldsymbol{K}_n|}{\partial \phi} = tr\left\{ \boldsymbol{K}_n^{-1} \frac{\partial \boldsymbol{K}_n}{\partial \phi} \right\}$$

Derivative-based hyperparameter optimization

The chain rule, and a single application of each of the identities above, gives

$$\begin{aligned} l'(g) &= -\frac{n}{2}\frac{\boldsymbol{Y}_n^T \frac{\partial \boldsymbol{K}_n^{-1}}{\partial g} \boldsymbol{Y}_n}{\boldsymbol{Y}_n^T \boldsymbol{K}_n^{-1} \boldsymbol{Y}_n} - \frac{1}{2}\frac{\partial \log \boldsymbol{K}_n}{\partial g} && (3) \\ &= \frac{n}{2}\frac{\boldsymbol{Y}_n^T (\boldsymbol{K}_n^{-1})^2 \boldsymbol{Y}_n}{\boldsymbol{Y}_n^T \boldsymbol{K}_n^{-1} \boldsymbol{Y}_n} - \frac{1}{2} tr(\boldsymbol{K}_n^{-1}) && (4) \end{aligned}$$

UNIVERSITY OF Cincinnati

Using the gradient of $-\log(l(g))$ in code

```
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)
```

Lengthscale: rate of decay of correlation

How about modulating the rate of decay of spatial correlation in terms of distance? Surely unadulterated Euclidean distance isn't equally suited to all data. Consider the following generalization, known as the **isotropic Gaussian family**.

$$C(\boldsymbol{x}, \boldsymbol{x}') = \exp\{-\frac{||\boldsymbol{x} - \boldsymbol{x}'||^2}{\theta}\}.$$

- Isotropic Gaussian correlation functions are indexed by a scalar hyperparameter θ, called the characteristic lengthscale.
- Sometimes this is shortened to lengthscale, or θ may be referred to as a range parameter, especially in geostatistics. When $\theta = 1$ we get back our inverse exponentiated squared Euclidean distance-based correlation as a special case.
- Isotropy means that correlation decays radially; Gaussian suggests inverse exponentiated squared Euclidean distance.

Inference on θ

- How to perform inference for θ?
- Should our GP have a slow decay of correlation in space, leading to visually smooth/slowly changing surfaces, or a fast one looking more wiggly?
- Like with nugget g, embedding θ deep within coordinates of a covariance matrix thwarts analytic maximization of log likelihood. Yet again like g, numerical methods are rather straightforward. In fact the setup is identical except now we have two unknown hyperparameters.

Algorithm on otpim for θ and g

Consider brute-force optimization without derivatives. The R function *nl* is identical to *nlg* except argument par takes in a two-vector whose first coordinate is θ and second is g. Only two lines differ, and those are indicated by comments in the code below.

```
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)
 }
```

Algorithm on otpim for θ and g

```
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
```

Too Many Evaluations of GP

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

## function gradient   actual
##       13       13       65
$
```

We're searching in two input dimensions, and a rule of thumb is that it takes two evaluations in each dimension to build a tangent plane to approximate a derivative. So if 13 function evaluations are reported, it'd take about $2 \times 2 \longrightarrow 4 \times 13 = 52$ additional runs to approximate derivatives, which agrees with our "by-hand" counter.

How can we improve upon those counts?

- Reducing the number of evaluations should speed up computation time. It might not be a big deal now, but as n gets bigger the repeated cubic cost of matrix inverses and determinants really adds up.
- What if we take derivatives with respect to θ and combine with those for g to form a gradient? That requires $\dot{\boldsymbol{K}}_n = \frac{\partial \boldsymbol{K}_n}{\partial \theta}$, to plug into inverse and determinant derivative identities. The diagonal is zero because the exponent is zero no matter what θ is. Off-diagonal entries of $\dot{\boldsymbol{K}}_n$ work out as follows.

$$K_\theta(\boldsymbol{x}, \boldsymbol{x}') = \exp\left\{ -\frac{||\boldsymbol{x} - \boldsymbol{x}'||^2}{\theta} \right\}$$

we have

$$\frac{\partial K_\theta(\boldsymbol{x}_i, \boldsymbol{x}_j)}{\partial \theta} = K_\theta(\boldsymbol{x}_i, \boldsymbol{x}_j) \frac{||\boldsymbol{x} - \boldsymbol{x}'||^2}{\theta}$$

Gradient with respect to θ

A slightly more compact way to write the same thing would be $\dot{\boldsymbol{K}}_n = \boldsymbol{K}_n \circ Distn/\theta^2$ where $\circ$ is a component-wise, Hadamard product, and Distn contains a matrix of squared Euclidean distances – our D in the code. An identical application of the chain rule for the nugget, but this time for θ, gives

$$l'(\theta) = \frac{\partial l(\theta, g)}{\partial \theta} = \frac{n}{2} \frac{\boldsymbol{Y}_n^T (\boldsymbol{K}_n^{-1} \dot{\boldsymbol{K}}_n \boldsymbol{K}_n^{-1}) \boldsymbol{Y}_n}{\boldsymbol{Y}_n^T \boldsymbol{K}_n^{-1} \boldsymbol{Y}_n} - \frac{1}{2} tr(\boldsymbol{K}_n^{-1} \dot{\boldsymbol{K}}_n) \quad (5)$$

A vector collecting the two sets of derivatives forms the gradient of $l(\theta, g)$, a joint log likelihood with τ^2 concentrated out. R code below implements the negative of that gradient for the purposes of MLE calculation with *optim* minimization.

Gradient Function

```
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))
 }
```

Optimization with Gradiant Information

```
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)

#############################################
########### Efficiency Check ################
#############################################

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

##       function gradient actual
## grad        10       10     10
#$# brute        13       13     65
```

That's way better. No only does our actual "by-hand" count of evaluations match what's reported on output from *optim*, but it can be an order of magnitude lower, roughly, compared to what we had before. A factor of five-to-ten savings is definitely worth the extra effort to derive and code up a gradient.

Back to the Prediction

```
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)
```

Anisotropic modeling

Lets assume a nonlinear surfacein five input coordinates with mean and variance 1,

$$EY(\boldsymbol{x}) = 10sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 - 5x_5,$$

```
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))
 }
```

Simulate from the above

Code below uses fried to generate an LHS training–testing partition (see, e.g., Figure 4.9) with $n = 200$ and $n' = 1000$ observations, respectively. Such a partition could represent one instance in the "bakeoff" described by Algorithm 4.1.

```
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)])
```

RMSE to see our fit

The code above extracts two types of Y-values for use in out-of-sample testing. De-noised yytrue values facilitate comparison with root mean-squared error (RMSE),

$$\sqrt{\frac{1}{n'} \sum_{i=1}^{n'} (y_i - \mu(\boldsymbol{x}_i))^2}.$$

Notice that RMSE is square-root Mahalanobis distance calculated with an identity covariance matrix. Noisy out-of-sample evaluations yy can be used for comparison by proper score, combining both mean accuracy and estimates of covariance.

Anisotropic modeling

Estimate the hyper-parameters (the same way we have done already):

```
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

## $par
## [1] 2.533216 0.005201
##
## $value
## [1] 683.5
##
## $counts
## function gradient
##       33       33
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

Prediction

Estimate the hyper-parameters (the same way we have done already):

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


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

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

Comparison to a non-parametric regression

How about comparing to MARS? That seems natural considering these data were created as a showcase for that very method. MARS implementations can be found in the mda (Leisch, Hornik, and Ripley 2017) and earth (Milborrow 2019) packages on CRAN.

```
install.packages("mda")
library(mda)
fit.mars <- mars(X, y)
p.mars <- predict(fit.mars, XX)
rmse <- c(rmse, mars=sqrt(mean((yytrue - p.mars)^2)))
rmse
## gpiso  mars
## 1.107 1.518
```

Anisotropic Covariance function

How about the following generalization for the correlation?

$$C_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}') = \exp\{-\sum_{k=1}^{m} \frac{(\boldsymbol{x}_k - \boldsymbol{x}'_k)^2}{\theta_k}\}$$

Here we're using a vectorized lengthscale parameter $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_m)$, allowing strength of correlation to be modulated separately by distance in each input coordinate. This family of correlation functions is called the separable or anisotropic Gaussian. Separable because the sum is a product when taken outside the exponent, implying independence in each coordinate direction. Anisotopic because, except in the special case where all θ_k are equal, decay of correlation is not radial. Remember that the mechanism generating our data has covaraince matrix:

$$\boldsymbol{\Sigma}(\boldsymbol{\theta}, \tau^2, g, n) = \tau^2(\boldsymbol{C}_{\boldsymbol{\theta},n}(\boldsymbol{x}, \boldsymbol{x}') + g\mathbb{I}_n)$$

Inference for a vectorized parameter How does one perform inference for such a vectorized parameter? Simple; just expand log likelihood and derivative functions to work with vectorized $\boldsymbol{\theta}$. Thinking about implementation: a for loop in the gradient function can iterate over coordinates, wherein each iteration we plug:

$$\frac{\partial K_\theta(\boldsymbol{x}_i, \boldsymbol{x}_j)}{\partial \theta_k} = K_\theta(\boldsymbol{x}_i, \boldsymbol{x}_j) \frac{||x_{ik} - x_{jk}||^2}{\theta_k}$$

into our formula for $l'(\theta_k)$ in Eq. from the isotropic covariance function above, which is otherwise unchanged.

Likelihood Function

```
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)
 }
```

Optimization

```
###############################
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
## function gradient   actual
##       66       66     1122
toc - tic
```

Gradient Function

```
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))
 }
```

Gradient Function

```
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)
```

```
##       function gradient actual
## grad       135      135    135
## brute       66       66   1122
```

```
toc - tic
```

Gradient Function

```
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

##  gpiso   mars  gpsep
## 1.1071 1.5176 0.6512
```