5. Gaussian likelihoods

Simon Vaughan [*]

July 8, 2016

In this section we discuss how to compute the likelihood function and how to use it. See Chapter 6 of *Scientific Inference* for a general introduction to likelihood functions.

The Gaussian Process model

We wish to study some time-variable phenomenon, and we have available measurements of one or more properties at a finite number of times. For example, the brightness of a variable star. We imagine the property is continuous in time, $f(t)$. We assume this is a *Gaussian Process* (GP) which means any finite collection of points have a (multivariate) Gaussian distribution.

$$\mathbf{f} \sim N(\boldsymbol{\mu}, K) \tag{1}$$

.

Where $\mathbf{f} = \{f_i | i = 1, 2, \ldots, n\}$ at times $\mathbf{t} = \{t_i | i = 1, 2, \ldots, n\}$. The mean is given by $\boldsymbol{\mu}_i = \mu(t_i)$ and the covariance is given by $K_{ij} = K(t_i, t_j)$.

The mean and covariance could be functions of time, e.g. $\mu(t)$. But we will always assume these do not change, this means the GP is *stationary.* In fact, we always assume $\boldsymbol{\mu}_i = \mu$ (some constant for all i) and $K(t_i, t_j)$ is a function of $|t_i - t_j|$.

We will only ever observe a finite number of points, or need to predict at a finite number of points (which could be arbitrarily dense), we can therefore use the mathematics of the (multivariate) Gaussian distribution to model the continuous Gaussian Process.

In real life we never get to know $f(t_i)$ because our measurements always have finite errors. What we measure is $\mathbf{y} = \{y_i | i = 1, 2, \ldots, n\}$ where

$$y(t_i) = f(t_i) + \epsilon_i \tag{2}$$

where ϵ_i are independently distributed random variates with mean zero and standard deviation σ_i: $\epsilon \sim N(0, \sigma_i^2)$. The distribution of $\mathbf{y}$ is also Gaussian

$$\mathbf{y} \sim N(\boldsymbol{\mu}, C) \tag{3}$$

.

The mean does not change (because $E[\epsilon] = 0$) but the covariance matrix has the error variances added.

$$\begin{aligned} C &= K + N \quad \text{where} \quad N = diag(\boldsymbol{\sigma}^2) \\ C_{ij} &= K(t_i, t_j) + \sigma_i^2 \delta_{ij} \end{aligned} \tag{4}$$

[*]Email: sav2@le.ac.uk

Eqn 29-30 of Rybicki & Press (1992) derive this expression.

The Gaussian likelihood

We begin with a *generative* model, a function or procedure that defines the probability distribution of some data $\mathbf{y}$ (n-dimensional column vector). If we think the data were generated by a Gaussian Process (GP), then $\mathbf{y}$ has a Gaussian distribution:

$$p(\mathbf{y}|\boldsymbol{\mu}, C) = \frac{1}{(2\pi)^{n/2}|C|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^T C^{-1}(\mathbf{y}-\boldsymbol{\mu})\right) \tag{5}$$

For given $\boldsymbol{\mu}$ and C, this is a probability density function (pdf) and obeys all the rules of probability theory. E.g. it is non-negative everywhere and integrates to unity. Using this we can generate random data.

But when we have the data already, $\mathbf{y} = \mathbf{y}_{obs}$, it makes no sense to talk of the probability of the data. The data are as they are, and have probability zero of being different! However, we can use equation 5 as a function of $\boldsymbol{\mu}$ and Σ for fixed $\mathbf{y}$. In this case it no longer obeys the rules of probability, e.g. the integral over all $\boldsymbol{\mu}$ and Σ values will generally not be unity. In this case it is called the *likelihood* of the data, and is a function of the *parameters*.

The *probability* (of the data) and the likelihood (of the parameters) are different – the equation is the same, but we use it differently: the probability is a function of the data given fixed parameters, the likelihood is a function of the parameters given fixed data.

Once we have some data, we may adjust the parameters of the model to find the values that maximise the likelihood. These are the maximum likelihood estimates (MLEs) for the parameters. This is not the only way to estimate parameters; we will discuss Bayesian methods in a later section.

The likelihood for a GP

For our purposes we shall assume the mean value of the GP is constant for all time, so $\boldsymbol{\mu} = \mu$. And the matrix Σ is formed from a valid autocovariance function $K(t_i, t_j)$, which is a function of $|t_i - t_j|$ only. This function may be as simple as

$$K(\tau|A, l) = A \exp\left(-\frac{|\tau|}{l}\right) \tag{6}$$

where $\tau = t_i - t_j$, and A and l are the parameters that specify the function. Or in matrix form

$$K_{ij} = K(t_i, t_j) = A \exp\left(-\frac{|t_i - t_j|}{l}\right) \tag{7}$$

But we also allow for the random errors $\boldsymbol{\sigma} = \{\sigma_i | i = 1, 2, \ldots, n\}$ on the measurements y_i. We do this be adding to the covariance matrix

$$\begin{aligned} C &= K + \nu N \\ C_{ij} &= K(t_i, t_j) + \nu \sigma_i^2 \delta_{ij} \end{aligned} \tag{8}$$

Notice we have included a constant factor ν here. If the given errors (σ_i) are correct then $\nu = 1$. But often the given errors are over- or under-estimated. We can adjust (or fit) ν to allow for this.

In R we can add to the diagonal elements quite simply, e.g.

```
# add the error matrix N[i.j] = diag(dy^2) for i=j
C <- K + nu * diag(dy*dy)
```

where `K` is the noise-free $n \times n$ matrix and `dy` is an n-vector containing the errors σ_i.

With this choice of covariance function, the Gaussian likelihood is a function of the four parameters μ, ν, A and l. We usually collect these together into a parameter vector $\boldsymbol{\theta} = \{\mu, \nu, A, l\}$. We can then write the probability of the data as $p(\mathbf{y}|\boldsymbol{\theta})$ and the likelihood as $L(\boldsymbol{\theta})$. Strictly, we should probably also note that this depends on the times $\mathbf{t}$, which are fixed in the data.

But there is no reason to restrict the ACV to a function as simple as equation 7, with only two parameters (or four if you include μ and ν). The function $K(\tau|\boldsymbol{\theta})$ could be arbitrarily complicated (here we have explicity included $\boldsymbol{\theta}$ in the arguments to the function).

For several reasons – not least of which is ease of computation – we usually work with the log likelihood function.

$$\begin{aligned} l(\boldsymbol{\theta}) &= \ln L(\boldsymbol{\theta}) = \ln p(\mathbf{y}|\boldsymbol{\mu}, C) \\ &= -\frac{n}{2}\ln(2\pi) - \frac{1}{2}\ln|C| - \frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^T C^{-1}(\mathbf{y}-\boldsymbol{\mu}) \end{aligned} \tag{9}$$

Efficiently computing the likelihood

We wish to evaluate equation 9 as quickly as possible. In order to find the MLE of the parameters $\boldsymbol{\theta}$ we will need to use an iterative optimisation algorithm (such as the Nelder-Mead simplex method). This will require many log likelihood evaluations, so we need to make the computation as efficient as possible.

There are three parts to the right side of equation 9. The first is $-(n/2)\ln(2\pi)$ which is simple to compute and is constant for a given data vector $\mathbf{y}$. E.g.

```
n <- length(y)
l.1 <- -(n/2) * log(2*pi)
```

where `y` is the n-vector of data $\mathbf{y}$. Note that the R function `log()` is the natural logarithm.

The second part of the log likelihood is $-(1/2)\ln|C|$. At first sight this would seem to require us to find the determinant of C. This can be done with e.g. `det()`. However, this is computationally demanding for large matrices. The `det()` function uses an LU decomposition of the target matrix, which requires $O(n^3)$ operations, so is slow for large n. We need to consider any alternative that gives a speed improvement.

One approach – the one discussed by Rasmussen & Williams (2006) in their algorithm 2.1 – is to use the Cholesky decomposition of the covariance matrix. If we know the (lower) Cholesky decomposition of Σ, i.e. $LL^T = C$, then we can use

$$|C| = \prod_{i=1}^{n} L_{ii}^2, \tag{10}$$

so

$$\frac{1}{2}\ln|C| = \sum_{i=1}^{n} \ln L_{ii}. \tag{11}$$

In R we can do this using the `chol()` function to compute the Cholesky decomposition, and the `t()` function to transpose a matrix (because `chol` computes the upper triangular Cholesky matrix). E.g.

```
L <- t( chol(C) )
l.2 <- -sum( log( diag(L) ) )
```

(We did not have to transpose the matrix here, but it will be useful to have L not L^T for the next step.)

The third part involves the quadratic form $\frac{1}{2}(\mathbf{y}-\mu)^T C^{-1}(\mathbf{y}-\mu)$ which involves the finding the inverse of C. We can compute the inverse directly using the `solve()` function. But computing a matrix inverse is another very slow process, so any alternative method that offers a speed improvement should be considered, ideally making use of the Cholesky matrix `L` we have already computed. We can write this as

$$\begin{aligned} Q &= (\mathbf{y}-\mu)^T C^{-1}(\mathbf{y}-\mu) \\ &= (\mathbf{y}')^T C^{-1}\mathbf{y}' \qquad (12) \\ &= \mathbf{z}^T\mathbf{z} \qquad (13) \end{aligned}$$

where

$$\mathbf{z} = (L^{-1})(\mathbf{y}-\mu) = L^{-1}\mathbf{y}' \qquad (14)$$

$$\mathbf{y}' = \mathbf{y}-\mu \qquad (15)$$

This works because

$$\begin{aligned} \mathbf{z}^T\mathbf{z} &= \left(L^{-1}\mathbf{y}'\right)^T\left(L^{-1}\mathbf{y}'\right) \\ &= (\mathbf{y}')^T(L^{-1})^T(L^{-1})\mathbf{y}' \\ &= (\mathbf{y}')^T C^{-1}\mathbf{y}' \qquad (16) \end{aligned}$$

since $(L^{-1})^T(L^{-1}) = C^{-1}$. In R we can use

```
y <- y - mu
z <- solve(L, y)
l.3 <- -0.5 * (z %*% z)[1]
```

(Note: the operation `z %*% z` computes the inner product of `z` with itself, i.e. $\mathbf{z}^T\mathbf{z}$. As this uses matrix operators, the result is also a matrix, but a 1×1 matrix. To convert the result to a scalar we simply extract the first element using the `[1]`.)

We do not need to explicitly compute any inverse matrices. The line `z <- solve(L, y)` finds the vector $\mathbf{z}$ such that $\mathbf{y} = L\mathbf{z}$. In other words, $\mathbf{z} = L^{-1}\mathbf{y}$.

The final log likelihood is the sum of these three parts.

```
loglike <- l.1 + l.2 + l.3
```

The Cholesky decomposition is also an $O(n^3)$ computation. But it is usually faster than either the direct inverse or determinant calculation, and can be used to compute both of these. The result is a factor ~few speed up in the calculation of the log likelihood.

However, we still have the problem that the computing time scales as n^3, and the memory usage scales as n^2 (as all the matrices such as `C` and `L` are $n \times n$), and so we will struggle when $n \geq 10^3$.

An R function

We can combine all these ideas into a single function which takes as input a vector of parameters (including the mean μ, and error scale factor ν), and the data. The data can be combined in *data frame* like this

```
dat <- data.frame(t = t.obs, y = y.obs, dy = errors)
```

where `t.obs`, `y.obs` and `errors` are the n-vectors of time, measurements and errors. Now all the information about the data is in a single object, which is a $n \times 3$ array with named columns.

Also, if we are making repeated calls to the log likelihood function, we can save computer time by pre-computing the matrix $\tau_{ij} = |t_i - t_j|$. For a given dataset this is constant. If we compute it once and store the result, we can reuse it each time we need to compute the log likelihood.

```
loglike <- function(theta, tau=NULL, dat) {

# -----------------------------------------------------------
# Inputs:
#   theta - vector of parameters for covariance function
#            the first element is the mean value mu
#   tau   - N*N array of lags at which to compute ACF
#   dat   - an n * 3 data frame/array, 3 columns
#            give the times, measurement and errors of
#            the n data points.
#
# Value:
#  loglike - scalar value of log[likelihood(theta)]
# -----------------------------------------------------------

# check arguments
  if (missing(theta)) {stop('** Missing theta input.')}
  if (!all(is.finite(theta))) {stop('** Non-finite values in theta.')}
  if (missing(dat)) {stop('** Missing dat input')}
  if (missing(dat$y)) {stop('** Missing dat$y.')}

# length of data vector(s)
   n <- length(dat$y)

# if there are no errors, and the dat£dy column is
# missing, make a column of zeroes.
```

```
  if (missing(dat$dy)) {
    dat$dy <- array(0, n)
  }

# if n * n array tau is not present then make one
  if (is.null(tau)) {
    tau <- abs( outer(dat$t, dat$t, "-") )
  }

# make sure y and tau have matching lengths
   if (ncol(tau) != n) {
    stop('** y and tau do not have matching dimensions')
  }

# first, extract the mean and error scale from the parameter vector theta,
# then remove them from the theta
  mu <- theta[1]
  nu <- theta[1]
  theta <- theta[c(-1, -2)]

# now subtract the mean from the data: y <- (y - mu)
  y <- dat$y - mu

# compute the covariance matrix C as C[i,j] = ACV(tau[i,j])
# using the remaining parameters
  K <- acv(theta, tau)

# check there aren't any missing values
  if (!all(is.finite(K))) {
    cat('Non-finite values in model covariance matrix.')
    return(NULL)
  }

# add the error matrix diag(dy^2)
  C <- K + nu*diag(dy*dy)

# compute the easy (constant) part of the log likelihood.
  l.1 <- -(n/2) * log(2*pi)

# find the Cholesky decomposition (lower left)
  L <- t( chol(C) )

# first, compute the log|C| term
  l.2 <- -sum( log( diag(L) ) )

# then compute the quadratic form -(1/2) * (y-mu)^T C^-1 (y-mu)
  z <- solve(L, y)
  l.3 <- -0.5 * (z %*% z)[1]

# combine all three terms for give the log[likelihood]
  loglike <- l.1 + l.2 + l.3
  return(loglike)
}
# ------------------------------------------------------------
```

Notice how there are comments before every few lines of code, explaining what every few lines is meant to do. At the start there is an explanation of the inputs and outputs – this makes is easier to check how to use the function properly. And there are several lines to check that the necessary inputs are present and in the right format. Checks like this help catch the most common causes of errors (using the function incorrectly!). We also use lines line `if (!all(is.finite(K)))` {...} to check that the matrix K doesn't have any non-finite values (such as `Inf`, `NA` or `NULL`) before proceeding with the calculation.