Gaussian Processes for non-linear regression and prediction problems

Søren Frimann

September 2, 2011

1 Introduction

Gathering and interpretation of data is a key aspect in science, and creating good models is often a crucial point in the analysis – this however often turns out to be a very tricky problem. The classical approach to modelling is to choose a parametric model and run some kind of optimizer to maximize the likelihood of the model. For complicated models that might suffer from a large amount of narrow local minima more sophisticated (and time consuming) approaches such as *Markov Chain Monte Carlo* might be chosen.

Gaussian Processes Regression (GPR) is a paradigm in *Supervised Learning*, that makes you able to solve non-linear regression and prediction problems without having to resort to parametric models which are not very flexible with regard to the data (the data has to be well represented by the parametric model), and might be difficult to get to converge. Figure 1 shows a typical example of a regression problem that can be solved by GPR. Given a set of noisy observations, $\mathcal{D} = (\mathbf{x}, \mathbf{y})$, assumed to be drawn from an underlying *latent function*, $f(x)$, we wish to be able to estimate the function value, at all other x positions. The classical approach would be to try a number of different parametric models, and then choosing the one that gives the highest probability (I avoid saying likelihood since more complex models will often be able to fit the data better, but not generally be the best choice). GPR is another approach where one does not assume the data to follow a specific parametric model, instead it enables the data to "speak for itself" – as we will see later one will still have to make assumptions about the data, but these assumptions operate at a level above the data.

The outline of the report is the follwing: In section 2 I will briefly go through the idea and theory behind *Gaussian Processes* (GPs), while simultaneously touch upon the subject of *Bayesian Inference*, since this is the framework in which we'll be working. In section 3 I outline how to use the software I've written for the project, and describe the most essential code. In section 4

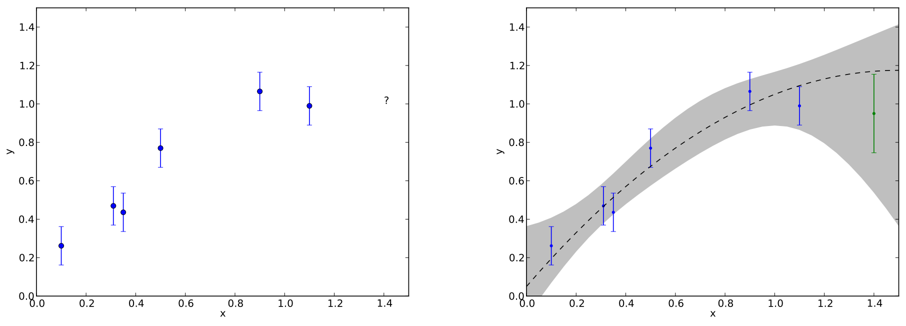

Figure 1: The classical regression problem. Left: Given six noisy data point we are interested in estimating a seventh at $x_\star = 1.4$. Right: The solution to the problem. The shaded area show the 2σ confidence interval for the value of the underlying function, while the green point show the predicted function value and 1σ uncertainty of the underlying function at $x_\star = 1.4$. The dashed line shows the actual function from which the observations were drawn in the first place.

I'll test the software on synthetic data to see how it performs, before applying it observations of Sunspots in section 5. In section 6 I'll discuss and conclude on the project.

All programming for this project was done in *Python*. The main purpose of this project has been to investigate and implement GPR, and I've therefore allowed myself to use some high-level routines already built into Python (or more specifically numpy and scipy). These include routines to do *Cholesky Decomposition*, and drawing samples from a *multivariate Gaussian Distribution*, but they also include routines to do *Downhill Simplex* optimization, and for solving linear systems of equations. I chose to go with the library routines, rather than the ones developed during the course, partly due to considerations of speed, and partly due to bugs that might linger in my own code, and make implementation of the Gaussian Processes more difficult.

2 Theory

A multivariate Gaussian distribution is fully specified by its mean vector, $/boldsymbol\mu$, and covariance matrix, Σ, such that when we draw a sample from the distribution we get a vector, $\mathbf{f} \sim \mathcal{N}(\mu, \Sigma)$. A Gaussian Process can be thought of as a continuous generalization of a multivariate Gaussian distribution. Formally:

> **Definition:** A Gaussian Process is a collection of random variables, any finite number of which have a joint Gaussian distribution [3]

In manner analougous to Gaussian distributions Gaussian Processes are fully specified by a mean function, $m(x)$, and a covariance function, $k(x,x')$, so that we can draw a sample, $f(x) \sim \mathcal{GP}\left(m(x), k(x,x')\right)$ from the GP. Working with continuous variables is not feasible when doing numerical calculations, but we're saved by the *marginalization property* which for multivariate Gaussians is a simple matter of dropping the variables one is not interested in [6]. More specifically:

$$p(\mathbf{x},\mathbf{y}) = \mathcal{N}\left(\begin{bmatrix}\mathbf{a}\\ \mathbf{b}\end{bmatrix}, \begin{bmatrix}A & C\\ C^T & B\end{bmatrix}\right) \Longrightarrow p(\mathbf{x}) = \mathcal{N}\left(\mathbf{a}, A\right).$$

This property means that it doesn't matter that we only look at a finite number of points in the Gaussian Process, since marginalizing out the rest won't change it!

As mentioned above you are not free of making some assumptions about the data when doing GPR. More specifically you have to imagine that your observations, $\mathcal{D} = (\mathbf{x},\mathbf{y})$, can be drawn from a multivariate Gaussian distribution with mean, $m(\mathbf{x})$, and covariance matrix, K, defined by the *Gram Matrix* [5] of the covariance function, $k(x,x')$. The results then ultimately depends on the choice of mean and covariance function for the GP. The mean function can be whatever function you like (often it is simply assumed to be zero), while the covariance function has to be positive semidefinite [3, Chapter 4]. A widely used covariance function is the *Squared Exponential*:

$$k(x,x') = \sigma_f^2 \exp\left(-\frac{(x-x')^2}{2\ell^2}\right). \tag{1}$$

This covariance function is *stationary*, that is it only depends on the distance between x and x' not the actual inputs themselves, which ensures that the samples drawn from the GP will be smooth (high covariance between points lying close to one another). $sigma_f$ and ℓ are the so-called *hyperparameters* (the set of them usually written as $\boldsymbol{\theta}$). Note that both $sigma_f$ and ℓ are both examples of what is called *scale parameters* (naturally positive parameters). To prevent problems during *optimization* later on we always take the *natural logarithm* of all scale parameters before giving them into the program. Depending on the training set at hand, different values of the hyperparameters, or different covariance functions altogether might be appropriate. Other examples of covariance functions, and their properties can be found in [3, Chapter 4], but other examples are also given in sections 4 and 5.

2.1 Bayesian Inference

In the following, I will briefly go through the framework of *Bayesian Inference*. This is pretty basic and covered in most elementary texts on the subject – a good introductory text is [1]. Specific information relating to Gaussian Processes is covered in [3, Chapter 2], and it is from here that most of the results in this section are quoted.

When working with GPR we will generally assume that we have a set of noisy observations, $\mathbf{y}$, that can be described as:

$$y_i = f(x_i) + \epsilon_i,$$

where $f(x)$ is the underlying latent function, and $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$ is *independent identically distributed* (IID) Gaussian noise with zero mean and variance, σ_n^2. Using these assumptions we can calculate the *likelihood* which will be a Gaussian distribution:

$$p(\mathbf{y}|\mathbf{x}, \mathbf{f}) = \mathcal{N}(\mathbf{f}, \sigma_n^2 I). \tag{2}$$

Here $\mathbf{f}$ is a vector of our best estimates for the value of the latent function, $f(x)$, at the same inputs as the targets, $\mathbf{y}$. I is the Identity matrix. We also define a prior probability for $\mathbf{f}$, encoding what we believe to be known about the system beforehand. This prior is assumed to be a multivariate Gaussian:

$$p(\mathbf{f}|\boldsymbol{\theta}, M_i) = \mathcal{N}(\mathbf{m}, K). \tag{3}$$

K and $\mathbf{m}$ are the covariance matrix and mean vector, constructed respectively from the covariance and mean function of the GP. As such the prior will be conditioned on the choice of hyperparameters, $\boldsymbol{\theta}$ and model (covariance function), M_i. We can get the posterior distribution by using *Bayes' Theorem*:

$$p(\mathbf{f}|\mathbf{y}, \mathbf{x}, \boldsymbol{\theta}, M_i) = \frac{p(\mathbf{y}|\mathbf{x}, \mathbf{f})p(\mathbf{f}|\boldsymbol{\theta}, M_i)}{p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}, M_i)}, \tag{4}$$

where $p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}, M_i)$ is the *marginal likelihood* given by:

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}, M_i) = \int p(\mathbf{y}|\mathbf{x}, \mathbf{f})p(\mathbf{f}|\boldsymbol{\theta}, M_i)\, d\mathbf{f}. \tag{5}$$

In order to make predictions about the value of the latent function at test input, $x_\star$, outside the training input, we have to calculate the likelihood of the predicted function values, $\mathbf{f}_\star \equiv f(\mathbf{x}_\star)$, multiply with the posterior probability in (4), and finally marginalize with over $\mathbf{f}$:

$$p(\mathbf{f}_\star|\mathbf{x}_\star, \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}, M_i) = \int p(\mathbf{f}_\star|\mathbf{x}_\star, \mathbf{y})p(\mathbf{f}|\mathbf{y}, \mathbf{x}, \boldsymbol{\theta}, M_i)\, d\mathbf{f}. \tag{6}$$

This procedure is equivalent to take the average of all possible parameters values weighed by the posterior probability.

Above we assumed that both likelihood and prior probability were multivariate Gaussians. Because of this one is able to evaluate all relevant integrals analytically. The main results of this section are (both from [3, Chapter 2]):

$$\begin{aligned} p(\mathbf{f}_\star|\mathbf{x}_\star, \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}, M_i) =& \mathcal{N}\Big(K_\star^T \left[K + \sigma_n^2 I\right]^{-1}(\mathbf{y} - \mathbf{m}), K(\mathbf{x}_\star, \mathbf{x}_\star) - \\ &- K_\star^T \left[K + \sigma_n^2 I\right]^{-1} K_\star\Big), \end{aligned} \tag{7}$$

for the predictive distribution and:

$$\log p(\mathbf{y}|\mathbf{x},\boldsymbol{\theta},M_i) = -\frac{1}{2}(\mathbf{y}-\mathbf{m})^T\left[K+\sigma_n^2 I\right]^{-1}(\mathbf{y}-\mathbf{m}) - \\ -\frac{1}{2}\log\left|K+\sigma_n^2 I\right| - \frac{n}{2}\log 2\pi, \tag{8}$$

for the log marginal likelihood. $K = K(\mathbf{x},\mathbf{x})$, and $K_\star = K(\mathbf{x},\mathbf{x}_\star)$, are covariance matrices constructed from the covariance function. We use the logarithm of the marginal likelihood since, for numerical reasons, this is the one we're going to use in the code. It is worth mentioning that the marginal likelihood depends on the hyperparameters though the covariance matrix, K, and that it contains a built-in *Occam's Razor* that effectively lowers the marginal likelihood for models that more complex than needed to describe the dataset well. This nice property is one of the reasons why Bayesian Inference has begun to regain much popularity in recent years, since it's a property one don't automatically get when using a classical approach. An example on how Occam's Razor work with the marginal likelihood can be seen in [3, Chapter 5], while an example of (7) is shown in Figure 2.

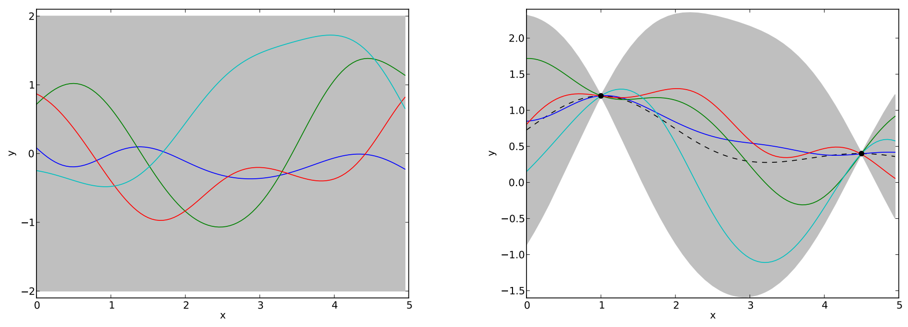

Figure 2: Left: Four samples drawn from the prior distribution (3), using a covariance matrix generated from a Squared exponential covariance function (1). The shaded area gives the 2σ uncertainty of the prior distribution. Right: Same four samples this time drawn from the predictive distribution (7), conditioned on two noiseless data points (in the case of no noise GPR essentially becomes an interpolation scheme known as *Kriging* [2, Chapter 3]). The uncertainty is again 2σ, and the dashed line gives the mean prediction.

Adaptation of the hyperparameters can also be done through Bayes' Theorem:

$$p(\boldsymbol{\theta}|\mathbf{y},\mathbf{x},M_i) = \frac{p(\mathbf{y}|\mathbf{x},\boldsymbol{\theta},M_i)p(\boldsymbol{\theta}|M_i)}{p(\mathbf{y}|\mathbf{x},M_i)}, \tag{9}$$

which gives the posterior of the hyperparameters, note that the likelihood for the hyperparameters is the same as the marginal likelihood in (8). In order to get the correct predictive description one multiply (7) with the above equation (9) and marginalizes with respect to $\boldsymbol{\theta}$. This is the correct Bayesian way to adapt the hyperparameters – unfortunately the integrals that appear here are not generally analytically tractable, and one has to resort to some kind of numerical integration scheme. Another way is to simply employ the "classical" approach and optimize (8) with respect to the hyperparameters (eventually starting optimization from a few different points in parameter space to check the convergence). According to [3] this is okay for the hyperparameters most of the time, since the location of the maximum likelihood is usually well localized in hyperparameter space. In this project we will employ the classical approach for the hyperparameters, using the downhill simplex algorithm built into scipy.

3 Description of Software

In this section I'll describe how to use the software that I developed, its limitations and a bit about the algoritms employed. The code is contained in two files **covar.py** and **gp.py**. **covar.py** just contains a selection of covariance functions and a helper function used to calculate covariance matrices, while **gp.py** contains all the real code.

The main function in **gp.py** is simply called **gp**, and takes up to five arguments and one keyword argument. There are three ways to call **gp**:

fprior,fcovar = **gp**(hypdict,covar,x,tol=1e-5)
nlml = **gp**(hypdict,covar,x,y,tol=1e-5)
f$_\star$,ferror,fcovar,alpha,nlml = **gp**(hypdict,covar,x,y,x$_\star$,tol=1e-5).

Going through the inputs, hypdict is a dictionary with the hyperparameters and will be discussed later. covar is the covariance function to be used (usually picked from **covar.py**). x is the training inputs, and y is the training targets. x$_\star$ is the prediction inputs. tol is a keyword giving the tolerance of the solution, more about that later. If **gp** is called with three arguments it assumes that one wants to draw samples from the prior, and returns the predicted mean (which is always constant in my implementation), and covariance matrix for the GP – this input is used when drawing samples from GP prior. If **gp** is called with four inputs it assumes that one is training the GP and it only returns the *negative log marginal likelihood*. It's make negative so that it'll work with the downhill simplex optimizer. Finally if **gp** is called with five arguments it assumes we're doing predictions and returns the mean estimate of the GP, the error estimate, the full covariance matrix, a parameter, alpha, which is useful when working with composite covariance functions, and again the negative log marginal likelihood.

As mentioned above hypdict is a dictionary in which we give hyperparameters of the GP. It always has the format:

hypdict = {'hyp': list, 'mean': float, 'sigma': float}.

'hyp' contains a list of the hyperparameters of the chosen covariance function, 'mean' gives the mean value of the GP, which in my implementation has to be a constant value, while 'sigma' gives the white noise component of the GP. The algorithm used for finding the mean prediction and covariance of the GP is identical to Algorithm 2.1 on page 19 in [3] and I won't print it here. Instead of computing the inverse of $K + \sigma_n^2 I$ directly, the algorithm uses the fact that covariance matrices are both *symmetric* and *positive definite*, and computes the *Cholesky Decomposition* [4] for the covariance matrix. The algorithm then uses the Cholesky decomposition to solve several system of equations to get the desired results. The main reason for using the Cholesky Decomposition is that it is considered more numerically stable than straight matrix inversion according to [3]. Some numerical instability however remains in the code, which is why I've introduced the the tol keyword. This simply adds a small value (default 10^{-5}) to the diagonal of the covariance matrix to do away with floating point errors that might otherwise make the Cholesky Decomposition impossible.

Another function inside **gp.py** is **draw_sample** which is used to draw random samples from the GP. It can be called in either of two ways:

sample = **draw_sample**(hypdict,covar,x,ns=1,seed=None)
sample = **draw_sample**(hypdict,covar,x,y,$x_\star$,ns=1,seed=None)

The inputs are the same as for **gp** (in fact **draw_sample** calls **gp** from inside), and depending on the number of inputs the samples are drawn from either a prior or conditioned GP. the ns keyword gives the number of samples (default is 1), while the seed keyword determines the seed of the pseudo-random number generator (default is None). Note that if sigma $\neq 0$ in the hypdict dictionary, a noisy sample will be drawn.

The last function inside **gp.py** is **hyper_optimize**, which as the name suggests tries to optimize the hyperparameters. It is called the following way:

hypdict,nlml = **hyper_optimize**(hypdict0,covar,x,y,mask).

hypdict0 is the first guess of the values of the hyperparameters, while mask is a dictionary with the same form as hypdict0, but with values equal to True or False depending on whether one wishes to optimize this parameter or not. An example would be

mask = {'hyp': True, 'mean': False, 'sigma': True}.

In this example **hyper_optimize** would not try to optimize the mean of the GP. This masking property is useful if one don't want to optimize either sigma (as would be the case when doing interpolation, where sigma is zero by definition) or mean. In the current version of the code one cannot single out hyperparameters inside the actual covariance function that one wishes not to optimize. **hyper_optimize** uses the downhill simplex algorithm, **fmin**, built into scipy which is not really optimal for the task. The reason for this is that matrix inversion is a $\mathcal{O}(n^3)$ task, and is carried out for every function call. A better approach would be a gradient optimizer which means that one will only have to invert the matrix once, hereby reducing the computational overhead. In an eventual future expansion of the code both the implementation of masking of individual hyperparameters, and optimization using a gradient method would have high priority.

4 Tests

In this section I'll go through some of the tests that I applied to check the validity of my code. All figures in this section as well as Figure 1 and 2 were generated by running **figures.py**.

We've already seen a couple of tests of the code in Figure 1 and 2. In Figure 1 we drew six noisy samples from an underlying function chosen to be a 2nd order polynomial. Applying GPR to the problem using a squared exponential covariance function was seen to give good results. Figure 2 was done by using **draw_sample** to draw random samples from first the GP prior and afterwards the GP conditioned on the two data points.

It is illustrative to see what would happen if we extend the range of the x-axis in Figure 1 to see what happens far away from the data. We do that in Figure 3, which is completely equivalent to Figure 1 except for the wider range. It is seen that when we far away from the data the GP falls back toward its prior distribution. This is not strange taking into consideration that the data was modelled using a Squared Exponential covariance function, which only has high covariance close to other data point.

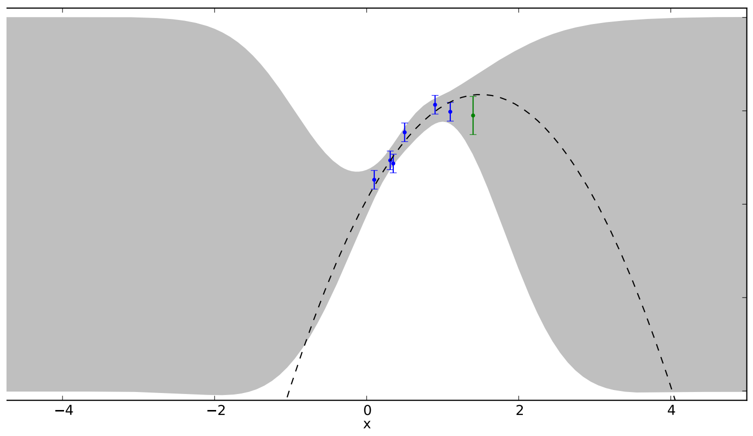

Figure 3: Same as Figure 1, but with wider range.

Another illustrative case to consider is what happens when one varies the hyperparameters. In Figure 4 I've shown data generated from a GP with a squared exponential covariance function with hyperparameters $(\ell, \sigma_f, \sigma_n) = (1, 1, 0.1)$, along with the predictions. In the following two plots I've shown the predictions with hyperparamters being respectively $(0.3, 0.5, 0.0005)$ and $(3, 1, 0.9)$. It's very clearly seen that neither case describes the data as well, as the first one.

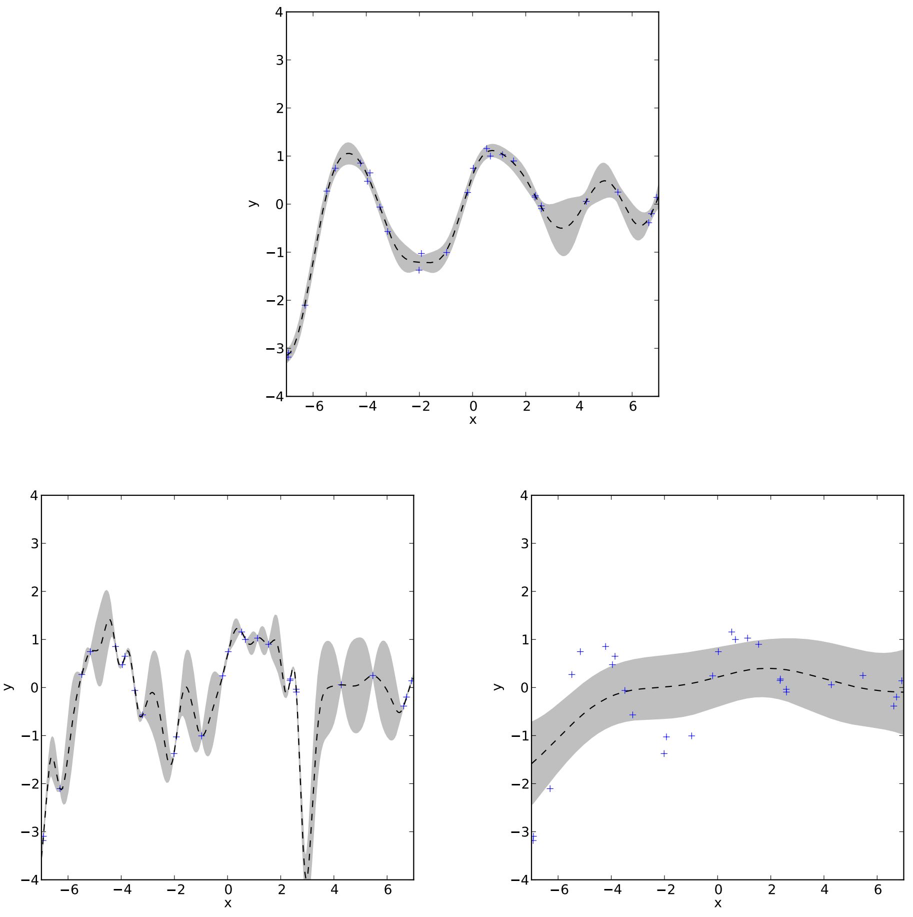

Figure 4: Top: Data drawn from a GP with hyperparameters $(\ell, \sigma_f, \sigma_n) = (1, 1, 0.1)$, and predictions using the same hyperparameters. Bottom Left: Same data but predictions using hyperparameters $(0.3, 0.5, 0.0005)$. Bottom Right: Same data but predictions using hyperparamters $(3, 1, 0.9)$

By using the property that sums and products of covariance functions are also valid covariance functions [3, Chapter 4], one can create composite covariance functions for data that cannot be described well by single covariance functions. An example of this is shown in Figure 5 where we have observations consisting of an oscillatory signal and a rising trend. We choose to model this by a sum of a squared exponential covariance function (1) and a periodic covariance function:

$$k(x, x') = \sigma_f^2 \exp\left(-\frac{2\sin^2\left(\pi\frac{x-x'}{P}\right)}{\ell^2}\right). \tag{10}$$

The periodic covariance function gives high covariance to points lying at a distance of P from eachother, and ℓ is a smoothness hyperparameter to determine the "intermediate" covariance in between the periodic peaks of covariance. All the hyperparameters for this covariance function are scale parameters.

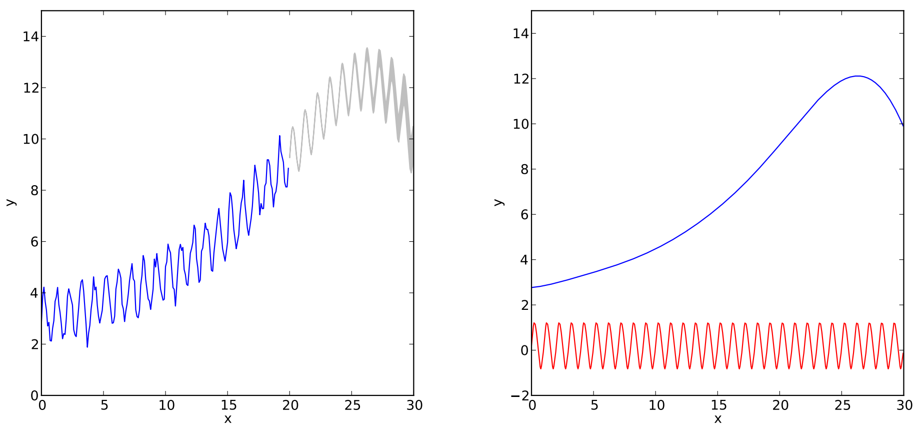

Figure 5: Left: The data and predictions ahead in time. Looking at the data the predictions looks reasonable for a while until they start dropping although there's nothing in the data to suggest this. Right: Predictions of the Squared Exponential, and periodic term.

The composite covariance function then ends up having the appearance:

$$k(x, x') = \theta_1^2 \exp\left(-\frac{(x-x')^2}{2\theta_2^2}\right) + \theta_3^2 \exp\left(-\frac{2\sin^2\left(\pi\frac{x-x'}{\theta_4}\right)}{\theta_5^2}\right) + \sigma_n \delta_{x,x'},$$

where I've included the noise hyperparameter in the function for clarity. Running the above covariance function though **hyper_optimize** gives the result $(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \sigma_n) = (3.66, 12.25, 1.00, 0.51, 1.27, 0.21)$. The log marginal likelihood for the model is -5.39. Note that the predictions initially look very good but that they begin to drop after a short while, although there's no indication of this in the data. This is simply where GPR hits it's limit, and goes to show that you should always be careful when interpreting model results, nomatter how they were created.

When you have a composite covariance function that can be described as a sum of several distinct covariance functions it's worth noting that the predictions for the distinct covariance functions can be separated. Recall from (7) that the mean prediction of the latent function, $\mathbf{f}_\star$ can be written:

$$\bar{\mathbf{f}}_\star = K_\star^T \left[K + \sigma_n^2 I\right]^{-1} (\mathbf{y} - \mathbf{m}).$$

Say that your covariance function can be written as a sum of two other covariance functions so that $k = k_1 + k_2$. Then it's possible to expand the above equation:

$$\begin{aligned}\bar{\mathbf{f}}_\star &= \left(K_{\star,1}^T + K_{\star,2}^T\right) \left[K_1 + K_2 + \sigma_n^2 I\right]^{-1} (\mathbf{y} - \mathbf{m}) \\ &= K_{\star,1}^T \left[K_1 + K_2 + \sigma_n^2 I\right]^{-1} (\mathbf{y} - \mathbf{m}) + K_{\star,2}^T \left[K_1 + K_2 + \sigma_n^2 I\right]^{-1} (\mathbf{y} - \mathbf{m}),\end{aligned}$$

where the first term will be the predictions belonging to k_1, and the second term predictions belonging to k_2. We show this property to the right in Figure 5, where the blue line is the mean predictions of the Squared Exponential covariance function, and the red line is the mean predictions of the periodic covariance function.

5 Example

6 Discussion & Conclusion

Bibliography

[1] Phil Gregory. *Bayesian Logical Data Analysis for the Physical Sciences: A Comparative Approach with Mathematica Support.* Cambridge University Press, 2005.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, 2007.

[3] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning series).* The MIT Press, 2006.

[4] Wikipedia. Cholesky decomposition — wikipedia, the free encyclopedia, 2011. [Online; accessed 28-August-2011].

[5] Wikipedia. Gramian matrix — wikipedia, the free encyclopedia, 2011. [Online; accessed 29-August-2011].

[6] Wikipedia. Multivariate normal distribution — wikipedia, the free encyclopedia, 2011. [Online; accessed 24-August-2011].