CHAPTER 18

Gaussian Processes

The supervised machine learning algorithms that we have seen have generally tried to fit a parametrised function to a set of training data in order to minimise an error function. This function is then used to generalise to previously unseen data. Some of the differences between the methods have been the set of model functions that the algorithm can use to represent the data; for example, the linear models of Chapter 3 and the piecewise constant splines of Chapter 5. However, if we do not know anything about the underlying process that generated the data, then choosing an appropriate model is often a trial-and-error process.

As a very simple example, Figure 18.1 shows a few datapoints. If we assumed that these were drawn from a single Gaussian distribution then we would have two parameters to fit (the mean and standard deviation) in order to get the best match that we could, as shown in the middle figure. However, choosing a different distribution (here, a Weibull distribution, which also has two parameters):

$$f(x;k,\lambda) = \begin{cases} \frac{k}{\lambda}\left(\frac{x}{\lambda}^{k-1} e^{-\left(\frac{x}{\lambda}\right)^k}\right), & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{18.1}$$

gives a better fit, as shown on the right (where the dashed line is the Weibull distribution and the solid line is the Gaussian). For the Gaussian $\mu = 0.7$ and $\sigma^2 = 0.25$, while for the Weibull $k = 2$ and $\lambda = 1$.

One possible solution to this problem is to let the optimisation process search over

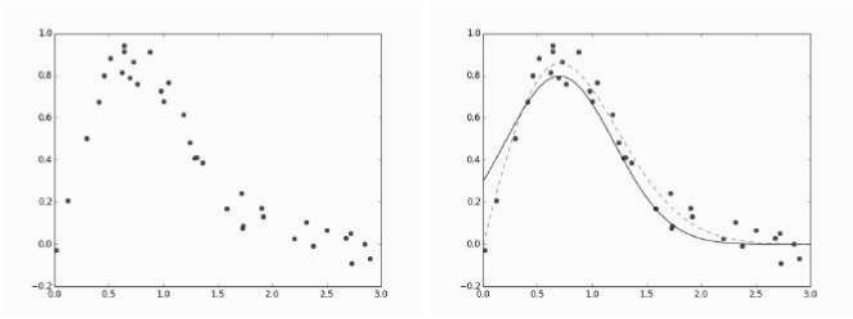

FIGURE 18.1 *Left:* a set of datapoints, *right:* two possible fits to that data, using a Gaussian (solid line) and Weibull distribution (dashed line). It can be seen that the Weibull distribution fits the data better, although both are a fairly good fit.

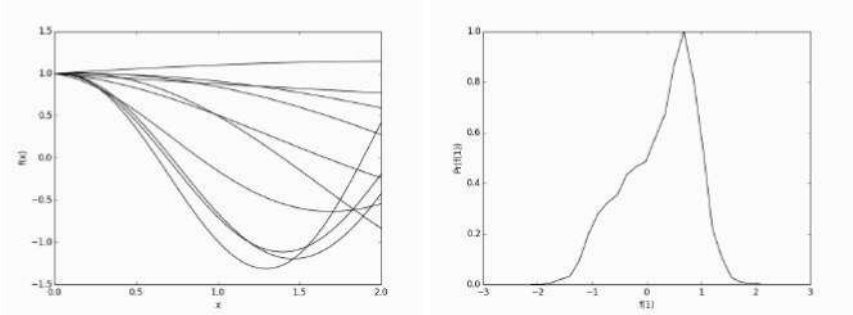

FIGURE 18.2 *Left:* 10 samples from the stochastic process $f(x) = \exp(ax)\cos(bx)$ with a and b drawn from Gaussian distributions. *Right:* The probability distribution of $f(1)$ based on 10,000 samples of $f(x)$.

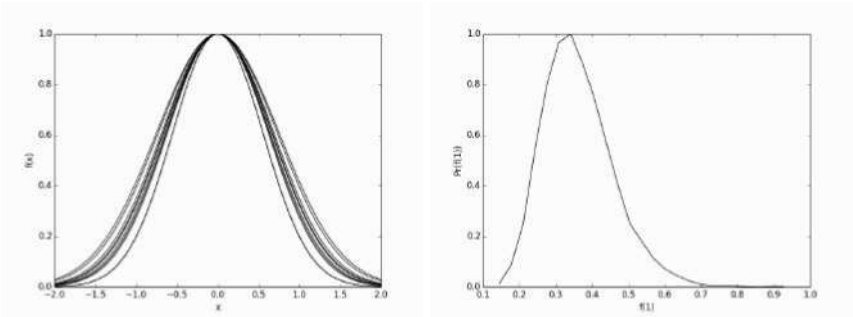

FIGURE 18.3 *Left:* 10 samples from the stochastic process $f(x) = \exp(-ax^2)$ with $a > 0$ drawn from a Gaussian distribution. *Right:* The probability distribution of $f(1)$ based on 10,000 samples of $f(x)$.

different models as well as the parameters of the model. To do this, we need to generalise the idea of a probability distribution to something that we can optimise over. This is known as a stochastic process, and it is simply a collection of random variables put together: instead of having a set of parameters that specify a probability distribution (such as the mean and covariance matrix for a multivariate Gaussian), we have a set of functions and a distribution over that set of functions. Figure 18.2 shows an example of a set of samples from the stochastic process $f(x) = \exp(ax)\cos(bx)$ with a drawn from a Gaussian with mean 0 and variance 0.25, and b from a Gaussian with mean 1 and variance 1, together with the probability distribution of $f(1)$ (computed from a set of 10,000 samples of $f(x)$).

Dealing with general stochastic processes is very difficult because combining the random variables is generally hard. However, if we restrict the process in such a way that all of the random variables have a Gaussian distribution, and the joint distribution over any (finite) subset of the variables is also Gaussian, then this Gaussian process (GP) is much easier to deal with. In order to see that it is still very powerful, Figure 18.3 shows a set of samples from $f(x) = \exp(-ax^2)$ with a drawn from a Gaussian distribution with mean 1 and standard deviation 0.25. It can be seen that the probability distribution of $f(1)$ is not a Gaussian.

The way to think about modelling with a Gaussian process is that we put a probability distribution over the space of functions and sample from that. A function is a mapping

from some (possibly multi-dimensional) input $\mathbf{x}$ to $f(\mathbf{x})$, so to specify the function we could just list the value of $f(x)$ for every value of x, which would be an infinitely long vector. One sample would consist of a specification of this vector. However, because everything is Gaussian, just as we specify a Gaussian distribution with the mean and covariance matrix, we can specify a Gaussian process by the mean function and a covariance function.

A complete specification of a particular function would, as has already been remarked, require an infinitely long vector. However, it turns out that Gaussian processes are very well behaved, so that considering only finite sets of points gives exactly the same inference result as would the compete integral (for more on this, see the references in the Further Reading section).

There have been various versions of Gaussian processes around for a very long time, known as kriging after one inventor, and Kolmogorov–Wiener prediction after two more. In more than one dimension it is technically a Gaussian random field, which was the focus of Section 16.2.

In fact, Gaussian processes are just smoothers, fitting a smooth curve through a set of datapoints. It seems amazing that such a simple process can be so powerful, but regression problems do all pretty much boil down to finding a smooth function that passes through the data. More surprisingly, we will see later in the chapter that Gaussian processes can also solve classification problems, which are harder to view in this way. Regardless of how it is viewed, it is time to start working out how to use one.

18.1 GAUSSIAN PROCESS REGRESSION

As was mentioned previously, the GP is specified by the mean and covariance functions. In fact, it is usual to subtract off the mean first, so that the mean function is identically zero. In this case, the GP is completely described as a function $G(k(\mathbf{x}, \mathbf{x}'))$ that models some underlying function $f(\mathbf{x})$, where covariance function $k(\mathbf{x}, \mathbf{x}')$ gives us the expected covariance matrix between the values of f at $\mathbf{x}$ and $\mathbf{x}'$. The random variables that define the GP are used to provide an estimate of $f(\mathbf{x})$ for each input $\mathbf{x}$.

This is where we have to put some prior work in: the covariance function needs to be specified, and this is what provides the expressive power of the GP. This covariance function is the same thing as the kernel, which we explored in Chapter 8, and there are strong links between SVMs and GPs; for more details see the references in the Further Reading section.

Taking a hint from SVMs, then, we will start with a covariance matrix that has the form of the RBF kernel:

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left(-\frac{1}{2l^2}|\mathbf{x} - \mathbf{x}'|^2\right). \tag{18.2}$$

In GPs, for some reason, this is normally known as the squared exponential covariance matrix rather than the RBF. For a set of input vectors it enables us to specify a matrix of covariances $\mathbf{K}$ where the element at place (i, j) in the matrix is $K_{ij} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$.

There are two parameters in this covariance function: σ_f and l, and we shall consider them shortly. First, though, we will work out how to use the GP to predict the value of $f^* = f(\mathbf{x}^*)$ for some values of $\mathbf{x}^*$ based on a training set of values $f(\mathbf{x})$.

As is usual for supervised learning, the training set consists of a set of N labelled examples $(\mathbf{x}_i, t_i), i = 1..N$. Since this is a GP, the joint density $P(t^*, \mathbf{t}_N)$ is a Gaussian (where the notation is meant to imply that t^* is a single test point, while $\mathbf{t}_N$ is the whole set of training target labels) and so is this conditional distribution:

$$P(t^*|\mathbf{t}_N) = P(t^*, \mathbf{t}_N)/P(\mathbf{t}_N). \tag{18.3}$$

The covariance matrix for the joint distribution is $\mathbf{K}_{N+1}$, which has size $(N+1)\times(N+1)$, and can be partitioned in the following way:

$$\mathbf{K}_{N+1} = \begin{pmatrix} \begin{bmatrix} & \mathbf{K}_N & \\ \mathbf{k}^{*T} & & \end{bmatrix} & \begin{bmatrix} \mathbf{k}^* \\ [k^{**}] \end{bmatrix} \end{pmatrix} \tag{18.4}$$

where $\mathbf{K}_N$ is the covariance matrix for the training data, $\mathbf{k}^*$ is the covariance matrix between the test points $\mathbf{x}^*$ and the training data (which also appears in transposed form), and k^{**} is the covariance between the points in the test set (which will be a single scalar value when building $\mathbf{K}_{N+1}$ from $\mathbf{K}_N$). If there are N pieces of training data and n test points, then the sizes of these parts are $N \times N$, $N \times n$, and $n \times n$, respectively. We will drop the size subscript from $\mathbf{K}$ from now on, and use it to denote the covariance matrix of the training data ($\mathbf{K}_N$) and use the notation introduced in Equation (18.4).

The joint distribution of the training and test data ($p(\mathbf{t}, t^*)$ is the Gaussian distribution with zero mean and the extended covariance matrix shown in Equation (18.4). We know the values of the observations for the test data, so we only want to produce samples that match the observables at these points. We could do this by choosing random samples and throwing them away if they don't match, but this would be very slow, since very few of the samples would match.

Fortunately, we can condition the joint distribution on the training data, which gives us the posterior distribution as:

$$P(t^*|\mathbf{t}, \mathbf{x}, \mathbf{x}^*) \propto \mathcal{N}\left(\mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{t}, k^{**} - \mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{k}^*\right), \tag{18.5}$$

where $\mathcal{N}(m, \Sigma)$ denotes a Gaussian distribution with mean m and covariance Σ.

There is one important thing to notice, which is the requirement to invert the $N \times N$ matrix $\mathbf{K}$, which is an expensive operation, and not necessarily a numerically stable one. The good news is that only the covariance matrix of the training data needs to be inverted, and so this only has to be done once. However, if there is a lot of training data then this is still an expensive $\mathcal{O}(N^3)$ operation, and it requires that the matrix is (numerically) invertible.

18.1.1 Adding Noise

The top-left plot in Figure 18.4 shows the mean and plus/minus two standard deviations of the posterior distribution for the squared exponential kernel with the five datapoints marked as the training data. In that plot, you can see that the variance at the training data is zero, which is fine if you don't believe that your training data has any noise. However, this is, of course, very unlikely. The usual way to add noise into any GP is to assume that it is independent, identically distributed Gaussian noise and so include an extra parameter into the covariance matrix, so that instead of using $\mathbf{K}$ we use $\mathbf{K}+\sigma_n^2\mathbf{I}$, where $\mathbf{I}$ is the $N \times N$ identity matrix. Noise is only added to the covariance for the training data. Together, the parameters of the kernel, including σ_n, are known as hyperparameters.

The posterior distribution is then:

$$P(t^*|\mathbf{t}, \mathbf{x}, \mathbf{x}^*) \propto \mathcal{N}\left(\mathbf{k}^{*T}(\mathbf{K} + \sigma_n\mathbf{I})^{-1}\mathbf{t}, k^{**} - \mathbf{k}^{*T}(\mathbf{K} + \sigma_n\mathbf{I})^{-1}\mathbf{k}^*\right). \tag{18.6}$$

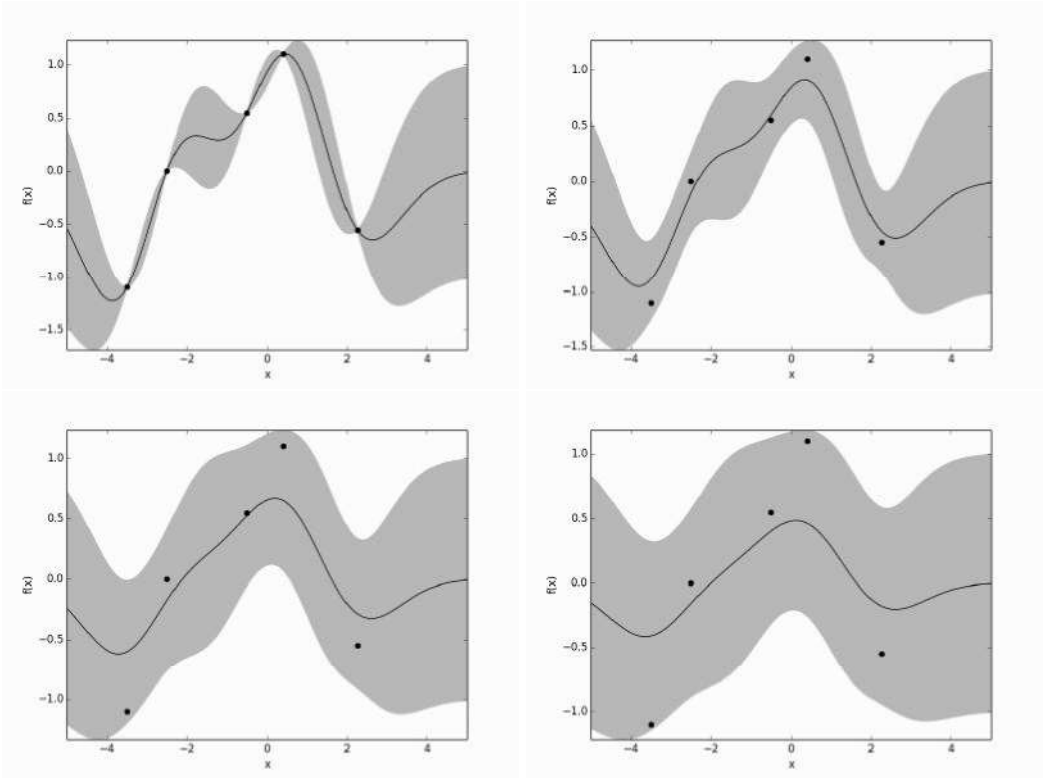

FIGURE 18.4 The effects of adding noise to the estimate of the covariance in the training data with the squared exponential kernel. Each plot shows the mean and 2 standard deviation error bars for a Gaussian process fitted to the five datapoints marked with dots. *Top left:* $\sigma_n = 0.0$, *top right:* $\sigma_n = 0.2$, *bottom left:* $\sigma_n = 0.4$, *bottom right:* $\sigma_n = 0.6$

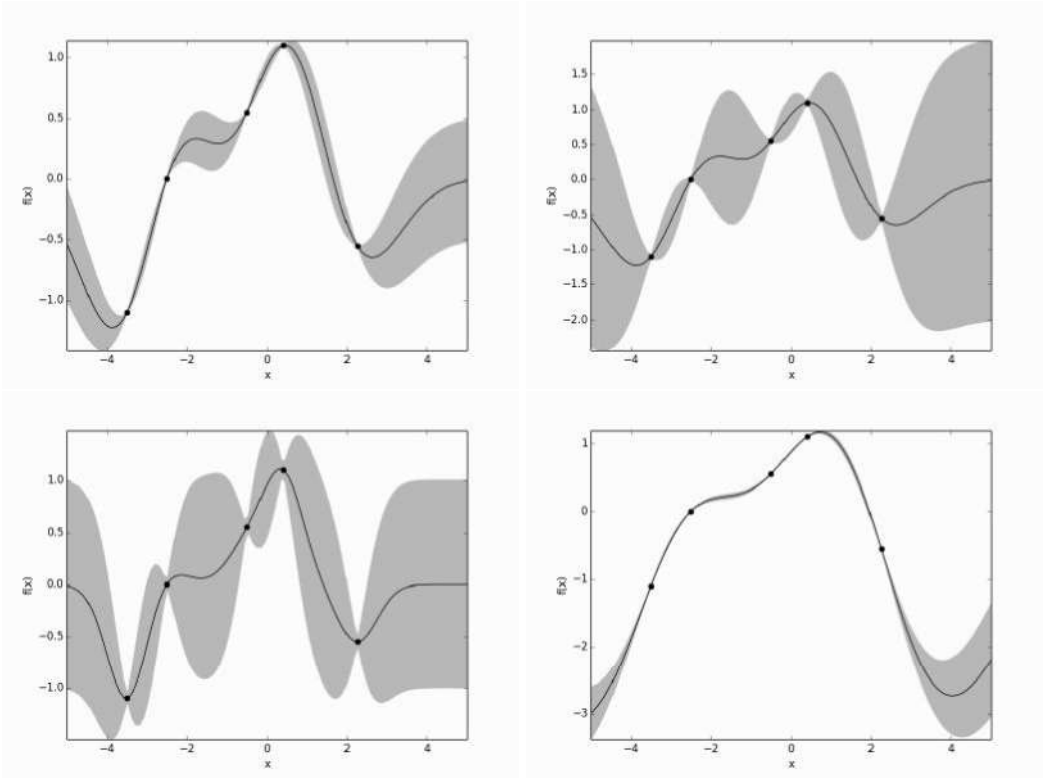

FIGURE 18.5 The effects of the other two parameters in the squared exponential kernel (compare to the top-left plot of Figure 18.4). Each plot shows the mean and 2 standard deviation error bars for a Gaussian process fitted to the five datapoints marked with dots. The parameters of the kernels were: *Top left:* $\sigma_f = 0.25, l = 1.0, \sigma_n = 0.0$, *top right:* $\sigma_f = 1.0, l = 1.0, \sigma_n = 0.0$, *bottom left:* $\sigma_f = 0.5, l = 0.5, \sigma_n = 0.0$, *bottom right:* $\sigma_f = 0.5, l = 2.0, \sigma_n = 0.0$.

The other three plots in Figure 18.4 show the effect that adding increasing amounts of observation noise makes.

Since we have considered the role of one of the hyperparameters, this is also a good place to consider the role of σ_f and l. Figure 18.5 shows the effects of changing these parameters for the same data as in Figure 18.4. It can be seen that modifying the signal variance σ_f^2 simply controls the overall variance of the function, while the length scale l changes the degree of smoothing, trading it off against how well the curve matches the training data.

Of the two parameters it is the l factor that is of most interest. It acts as a length scale, which says something about how quickly the function changes as the inputs vary. Figure 18.6 shows GP regression with similar data, except that in the plots on the second row, the x values of the points have been brought closer together. On the left, $l = 1.0$, while on the right $l = 0.5$. It can be seen that the top left and bottom right plots, where the length scale 'matches' the distances in the data, the fit looks smoother.

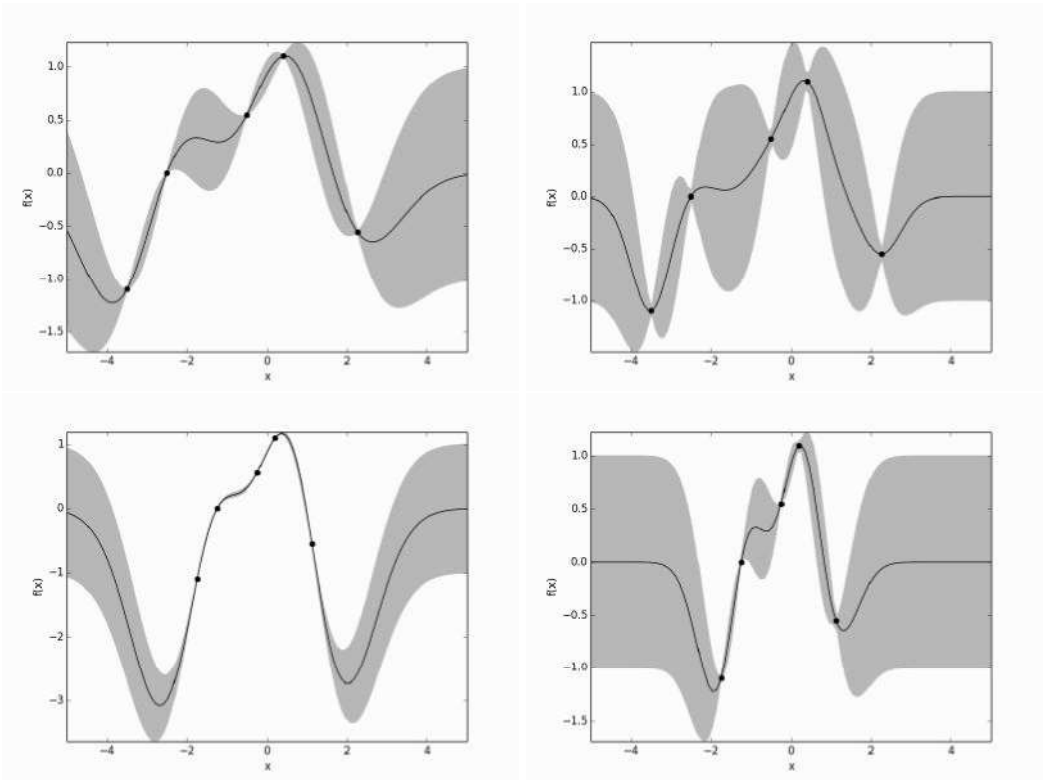

FIGURE 18.6 The effects of changing the length scale in GP regression. The top row shows one dataset, while the second row shows the same dataset, but with the points brought closer together. The length scale is the same for the plots above each other, being $l = 1.0$ on the left and $l = 0.5$ on the right.

18.1.2 Implementation

We have seen everything that we need to compute a basic Gaussian process regression program: we compute the covariance matrix of the training data, and also the covariances between the training and test data, and the test data alone. Then we compute the mean and covariance of the posterior distribution and sample from it. This results in the following algorithm:

Gaussian Process Regression

- For given training data $(\mathbf{X}, \mathbf{t})$, test data $\mathbf{x}^*$, covariance function $k()$, and hyperparameters $\boldsymbol{\theta} = (\sigma_f^2, l\sigma_n^2)$:
  - compute the covariance matrix $\mathbf{K} = k(\mathbf{X}, \mathbf{X}) + \sigma_n \mathbf{I}$ for hyperparameters $\boldsymbol{\theta}$
  - compute the covariance matrix $\mathbf{k}^* = k(\mathbf{X}, \mathbf{x}^*)$
  - compute the covariance matrix $k^{**} = k(\mathbf{x}^*, \mathbf{x}^*)$
  - the mean of the process is $\mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{t}$
  - the covariance is $k^{**} - \mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{k}^*$

However, before implementing it, there are a few numerical problems that need to be dealt with, as inverting the matrix $(\mathbf{K}+\sigma_n\mathbf{I})$ is not always stable, as it can have eigenvalues that are very close to 0.

Since we know that $\mathbf{K}$ is symmetric and positive definite, there are more stable ways to perform the inversion. The key is what is known as the Cholesky decomposition, which decomposes a real-valued matrix $\mathbf{K}$ into the product $\mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is a lower triangular matrix that only has non-zeros entries on and below the leading diagonal. There are two benefits to this, first that it is relatively cheap to calculate the inverse of a lower triangular matrix (and the inverse of the original matrix is $\mathbf{K}^{-1} = \mathbf{L}^{-T}\mathbf{L}^{-1}$, where $\mathbf{L}^{-T} = (\mathbf{L}^{-1})^T$), and secondly that it provides a very quick and easy way to solve linear systems $\mathbf{A}\mathbf{x} = \mathbf{b}$.

In fact, these two benefits are both parts of the same thing, since the inverse of a matrix $\mathbf{A}$ is the matrix $\mathbf{B}$ for which $\mathbf{A}\mathbf{B} = \mathbf{I}$, and we can solve this column-by-column as $\mathbf{A}\mathbf{B}_i = \mathbf{I}_i$ (where the subscript is an index for the ith column of the matrix).

To solve $\mathbf{L}\mathbf{L}^T\mathbf{x} = \mathbf{t}$ it is simply a matter of forward substitution to find the $\mathbf{z}$ that solves $\mathbf{L}\mathbf{z} = \mathbf{t}$ followed by back-substitution to find the $\mathbf{x}$ that solves $\mathbf{L}^T\mathbf{x} = \mathbf{z}$.

The cost of these operations is $\mathcal{O}(n^3)$ for the Cholesky decomposition and $\mathcal{O}(n^2)$ for the solve, and the whole thing is numerically very stable. NumPy provides implementations of both of these computations in the `np.linalg` module, and so the whole computation of the mean (`f`) and covariance (`V`) can be written as:

```
L = np.linalg.cholesky(k)
beta = np.linalg.solve(L.transpose(), np.linalg.solve(L,t))
kstar = kernel(data,xstar,theta,wantderiv=False,measnoise=0)
f = np.dot(kstar.transpose(), beta)
v = np.linalg.solve(L,kstar)
V =                    kernel(xstar,xstar,theta,wantderiv=False,measnoise=0)-
np.dot(v.transpose(),
),v)
```

The computation of `V` uses $\mathbf{v}^T\mathbf{v}$, where $\mathbf{Lv} = \mathbf{k}^*$ and to see that this does indeed match the covariance in Equation (18.6) requires a little bit of algebra:

$$\begin{aligned}\mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{k}^* &= (\mathbf{Lv})^T\mathbf{K}^{-1}\mathbf{Lv} \\ &= \mathbf{v}^T\mathbf{L}^T(\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{Lv} \\ &= \mathbf{v}^T\mathbf{L}^T\mathbf{L}^{-T}\mathbf{L}^{-1}\mathbf{Lv} \\ &= \mathbf{v}^T\mathbf{v}\end{aligned}$$

Comparing the code to Equation (18.6) you might also notice that the mean can be written in a slightly different way as:

$$m(\mathbf{x}, \mathbf{x}^*) = \sum_i \beta_i k(\mathbf{x}_i, \mathbf{x}^*), \tag{18.7}$$

where β_i is the ith part of $\beta = (\mathbf{K} + \sigma_n^2\mathbf{I})^{-1}\mathbf{t}$. This suggests that we can consider GP regression as the sum of a set of basis functions positioned on the training data; indeed for the squared exponential covariance matrix, we have produced precisely an RBF method; see Chapter 5. In that chapter we could modify the weights that specified the locations of the RBFs, but here we can't, but we can modify the weights that connect them to the outputs. Seen in this way, this GP is basically a linear neural network.

So providing that the hyperparameters are chosen to match the data, using a GP for regression is very simple. Now we are ready to do some learning to modify the parameters based on the data in order to improve the fit of the GP.

18.1.3 Learning the Parameters

The squared exponential covariance matrix (Equation (18.2)) has three hyperparameters (σ_f, σ_n, l) that need to be selected, and we have already seen that they can have a significant effect on the shape of the resulting output curve, so that finding the correct values is very important. In the next section we will also see that with more complex covariance matrices there are many more hyperparameters to choose, and so finding an automatic method of choosing the hyperparameters is clearly important if GPs are going to be useful.

If the set of hyperparameters are labelled as $\boldsymbol{\theta}$ then the ideal solution to this problem would be to set up some kind of prior distribution over the hyperparameters and then integrate them out in order to maximise the probability of the output targets:

$$P(t^*|\mathbf{x}, \mathbf{t}, \mathbf{x}^*) = \int P(t^*|\mathbf{x}, \mathbf{t}, \mathbf{x}^*, \boldsymbol{\theta})P(\boldsymbol{\theta}|\mathbf{x}, \mathbf{t})d\boldsymbol{\theta}. \tag{18.8}$$

This integral is very rarely tractable, but we can compute the posterior probability of $\boldsymbol{\theta}$ (which is the marginal likelihood times $P(\boldsymbol{\theta})$). The log of the marginal likelihood (also known as the evidence for the hyperparameters, which marginalises over the function values) is:

$$\log P(\mathbf{t}|\mathbf{x}, \boldsymbol{\theta}) = -\frac{1}{2}\mathbf{t}^T(\mathbf{K} + \sigma_n^2\mathbf{I})^{-1}\mathbf{t} - \frac{1}{2}\log|\mathbf{K} + \sigma_n^2\mathbf{I}| - \frac{N}{2}\log 2\pi. \tag{18.9}$$

In order to derive this equation you need to remember that the product of two Gaussians is also Gaussian (up to normalisation) and then write out the equation of a multivariate Gaussian and take the logarithm.

We now want to minimise this log likelihood, which we can do by using our favourite

gradient descent solver from Chapter 9 (for example, conjugate gradients from Section 9.3), providing that we first compute the gradient of it with respect to each of the hyperparameters. We will write $\mathbf{Q} = (\mathbf{K} + \sigma_n^2\mathbf{I})$ and then recall that $\mathbf{Q}$ is a function of all of the hyperparameters $\boldsymbol{\theta}_i$. Amazingly, these derivatives have a very nice form, as can be seen with the use of two matrix identities (where $\frac{\partial Q}{\partial \boldsymbol{\theta}}$ is simply the element-by-element derivative of the matrix):

$$\frac{\partial \mathbf{Q}^{-1}}{\partial \boldsymbol{\theta}} = -\mathbf{Q}^{-1}\frac{\partial \mathbf{Q}}{\partial \boldsymbol{\theta}}\mathbf{Q}^{-1} \tag{18.10}$$

$$\frac{\partial \log |\mathbf{Q}|}{\partial \boldsymbol{\theta}} = \text{trace}\left(\mathbf{Q}^{-1}\frac{\partial \mathbf{Q}}{\partial \boldsymbol{\theta}}\right). \tag{18.11}$$

Then:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \log P(\mathbf{t}|\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{2}\mathbf{t}^T\mathbf{Q}^{-1}\frac{\partial Q}{\partial \boldsymbol{\theta}}\mathbf{Q}^{-1}\mathbf{t} - \frac{1}{2}\text{trace}\left(\mathbf{Q}^{-1}\frac{\partial \mathbf{Q}}{\partial \boldsymbol{\theta}}\right). \tag{18.12}$$

Now, all that is required is to actually perform the computations of the derivatives of the covariance with respect to each hyperparameter, and then optimise the log likelihood using the conjugate gradient solver.

It will make things slightly easier if we change the way that the hyperparameters are presented a little bit. Note that all of the hyperparameters are positive numbers (since they are all squared in Equation (18.2). We can also make them positive by taking the exponential of each of them, and since the derivative of an exponential is just the exponential, this can make things a little clearer. Further, we will effectively work with $1/\sigma_l$ since it also makes the computation easier.

For the squared exponential kernel (where there is a slight notation abuse in the use of the identity matrix $\mathbf{I}$ in the last term):

$$k(\mathbf{x}, \mathbf{x}') = \exp(\sigma_f)\exp\left(-\frac{1}{2}\exp(\sigma_l)|\mathbf{x} - \mathbf{x}'|^2\right) + \exp(\sigma_n)\mathbf{I} \tag{18.13}$$

$$= k' + \exp(\sigma_n)\mathbf{I} \tag{18.14}$$

these are nice and easy to compute:

$$\frac{\partial k}{\partial \sigma_f} = k' \tag{18.15}$$

$$\frac{\partial k}{\partial \sigma l} = k' \times \left(-\frac{1}{2}\exp(\sigma_l)|\mathbf{x} - \mathbf{x}'|^2\right) \tag{18.16}$$

$$\frac{\partial k}{\partial \sigma_n} = \exp(\sigma_n)\mathbf{I} \tag{18.17}$$

Note that the term inside the bracket in $\frac{\partial k}{\partial \sigma l}$ is precisely the one that has already been computed for the exponential calculation.

18.1.4 Implementation

The basic algorithm is very simple again, which is to call the conjugate gradient optimiser to minimise the log likelihood, providing it with the computations of the gradients with

respect to the parameters. The SciPy optimiser was used in Section 9.3 and the syntax is no different here:

```
result = so.fmin_cg(logPosterior, theta, fprime=gradLogPosterior,
args=[(X,y)]
], gtol=1e-4,maxiter=5,disp=1)
```

where possible implementations of the log likelihood and gradient functions are:

```
def logPosterior(theta,args):
                data,t = args
                k = kernel2(data,data,theta,wantderiv=False)
                L = np.linalg.cholesky(k)
                beta = np.linalg.solve(L.transpose(), np.linalg.solve(L,t))
                logp = -0.5*np.dot(t.transpose(),beta) - np.sum(np.log(np.
                diag(L))) - np.shape(data)[0] /2. * np.log(2*np.pi)
                return -logp

def gradLogPosterior(theta,args):
                data,t = args
                theta = np.squeeze(theta)
                d = len(theta)
                K = kernel2(data,data,theta,wantderiv=True)

                L = np.linalg.cholesky(np.squeeze(K[:,:,0]))
                invk = np.linalg.solve(L.transpose(),np.linalg.solve(L,np.
                eye(np.shape(data)[0])))

                dlogpdtheta = np.zeros(d)
                for d in range(1,len(theta)+1):
                        dlogpdtheta[d-1] = 0.5*np.dot(t.transpose(), np.dot(
                        invk, np.dot(np.squeeze(K[:,:,d]), np.dot(invk,t)))) 
                        - 0.5*np.trace(np.dot(invk,np.squee
ze(K[:,:,d])))

                return -dlogpdtheta
```

In terms of implementation, the only thing that we have not covered yet is how to compute the covariance matrix, but there is nothing complex about that: the function takes in two sets of datapoints and returns the covariance matrix, and possibly the gradients as well (which is the `wantd` switch). One possible way to do this for the squared exponential kernel is:

```
def kernel(data1,data2,theta,wantderiv=True,measnoise=1.):
        # Squared exponential
        theta = np.squeeze(theta)
        theta = np.exp(theta)
        if np.ndim(data1) == 1:
                d1 = np.shape(data1)[0]
                n = 1
        else:
                (d1,n) = np.shape(data1)

        d2 = np.shape(data2)[0]
        sumxy = np.zeros((d1,d2))
        for d in range(n):
                D1 = np.transpose([data1[:,d]]) * np.ones((d1,d2))
                D2 = [data2[:,d]] * np.ones((d1,d2))
                sumxy += (D1-D2)**2*theta[d+1]

        k = theta[0] * np.exp(-0.5*sumxy)

        if wantderiv:
                K = np.zeros((d1,d2,len(theta)+1))
                K[:,:,0] = k + measnoise*theta[2]*np.eye(d1,d2)
                K[:,:,1] = k
                K[:,:,2] = -0.5*k*sumxy
                K[:,:,3] = theta[2]*np.eye(d1,d2)
                return K
        else:
                return k + measnoise*theta[2]*np.eye(d1,d2)
```

Figure 18.7 shows an example of a Gaussian process before and after optimisation, with initially random hyperparameters. Before the optimisation process the log likelihood of the data under the model, based on random initialisation of the hyperparameters, was around 60, whereas afterwards it was around 16. It can be seen that the model fits the data much better after the optimisation process.

18.1.5 Choosing a (set of) Covariance Functions

Like any other kernel, the choice of covariance function is crucial to successful prediction. This is the modelling part of GP learning, and it is entirely human-dependent: you choose appropriate covariance functions and then the algorithm learns their parameters. All that is required is that the functions must generate positive-definite (or actually, non-negative definite) covariance matrices. There is a fairly large choice of typical kernels for GPs, but given that our only restriction is that they must be positive-definite, it is possible to add and multiply kernels as we saw in Section 8.2 using Mercer's theorem. The upshot of this is that you can string together a whole set of covariance functions that represent different parts of what you believe the data is doing. So for example, if you think that there are two different squared exponential processes, but with different length scales, you could include two versions of the kernel, and optimise the two different length scales.

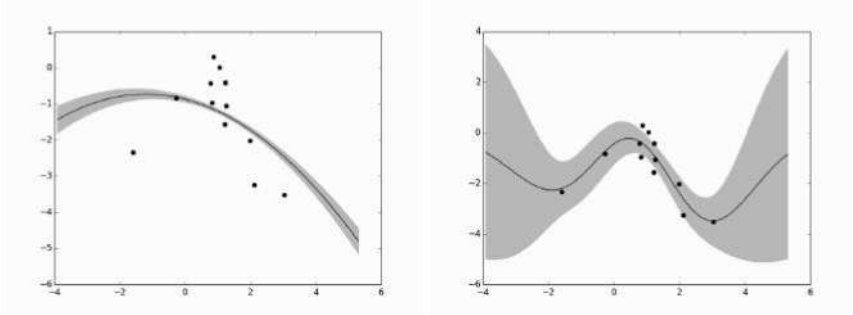

FIGURE 18.7 *Left:* the data and the model based on random parameters, *right:* the fitted model.

A few commonly used covariance functions are:

Constant $k(\mathbf{x}, \mathbf{x}') = e^{\sigma}$

Linear $k(\mathbf{x}, \mathbf{x}') = \sum_{d=1}^{D} e^{\sigma_d} \mathbf{x}_d \mathbf{x}'_d$

Squared Exponential $k(\mathbf{x}, \mathbf{x}') = e^{\sigma_f} \exp\left(-\frac{1}{2}\exp(\sigma_l)(\mathbf{x} - \mathbf{x}')^2\right)$

Ornstein–Uhlenbeck $k(\mathbf{x}, \mathbf{x}') = \exp\left(-\exp(\sigma_l)|\mathbf{x} - \mathbf{x}'|\right)$

Matérn $k(\mathbf{x}, \mathbf{x}') = \frac{1}{2^{\sigma_\nu - 1}\Gamma(\sigma_\nu)} \left(\frac{\sqrt{2\sigma_\nu}}{l}(\mathbf{x} - \mathbf{x}')\right)^{\nu} K_\nu \left(\frac{\sqrt{2\sigma_\nu}}{l}(\mathbf{x} - \mathbf{x}')\right)$,
where K_{σ_ν} is a modified Bessel function and Γ is the gamma function.

Periodic $k(\mathbf{x}, \mathbf{x}') = \exp\left(-2\exp(\sigma_l)\sin^2(\sigma_\nu \pi(\mathbf{x} - \mathbf{x}'))\right)$

Rational Quadratic $k(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{1}{2\sigma_\alpha}\exp(\sigma_l)(\mathbf{x} - \mathbf{x}')^2\right)^{-\sigma_\alpha}$

18.2 GAUSSIAN PROCESS CLASSIFICATION

While it is possible to perform multi-class classification with a Gaussian process, we will consider only two classes, labelled as +1 and -1. The task of the process is then to model the probability that input $\mathbf{x}$ belongs to class 1, which means that the output should be a value between 0 and 1 (inclusive) like all good probabilities. We will arrange this in the same way that we did it for neurons: by squashing it using the logistic function $P(t^* = 1|a) = \sigma(a) = 1/(1 + \exp(-a))$, where a is the output of the regression GP, and a little care is needed since we are now using $\sigma(\cdot)$ to denote the logistic function, as well as σ_n to denote a hyperparameter and even σ^2 as the variance. Since there are two classes $P(t^* = -1|a) = 1 - P(t^* = 1|a)$, and so we can write $p(t^*|a) = \sigma(t^* f(x^*))$. So GP classification consists of finding a GP prior over $f(\mathbf{x})$ (known as the latent function) and then putting this through the logistic function to find a prior on the predicted class, which is:

$$p(t^* = 1|\mathbf{x}, \mathbf{t}, \mathbf{x}^*) = \int \sigma(f(\mathbf{x}^*)) p(f(\mathbf{x}^*)|\mathbf{x}, \mathbf{t}, \mathbf{x}^*) df(\mathbf{x}^*). \tag{18.18}$$

This is a 1D integral, and so it can be computed numerically, but unfortunately the likelihood function $p(f(\mathbf{x}^*)|\mathbf{x}, \mathbf{t}, \mathbf{x}^*)$ is not a Gaussian function and so computing that term is

intractable. This means that some form of approximation is needed. There are several methods of doing these approximations, including using MCMC, but we will consider only the simplest version, which is known as Laplace's approximation. The references at the end of the chapter provide a list of places with more information about more advanced approximation methods.

18.2.1 The Laplace Approximation

Laplace's approximation is a way to approximate any integral of the form $\int \exp(f(\mathbf{x}))d\mathbf{x}$, which, of course, includes Gaussians. The basic idea is to find the global maximum of the function $f(\mathbf{x})$, which occurs at some $\mathbf{x}_0$. At this point the gradient of $f(\mathbf{x})$ (which is $\nabla f(\mathbf{x})$) is 0, and so the second-order Taylor expansion around $\mathbf{x}_0$ is (where $\nabla\nabla f(\cdot)$ is the Hessian matrix):

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \nabla\nabla f(\mathbf{x})(\mathbf{x} - \mathbf{x}_0). \tag{18.19}$$

Since the logarithm of a Gaussian is a quadratic function, this has a unique maximum, and so we replace $f(\mathbf{x})$ by $\log f(\mathbf{x})$ and then compute the exponential of this, which tells us that:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) \exp\left(\frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \nabla\nabla \log f(\mathbf{x})(\mathbf{x} - \mathbf{x}_0)\right). \tag{18.20}$$

Normalising this to make it a Gaussian distribution tells us that:

$$\begin{aligned} q(f(\mathbf{x})|\mathbf{x}, \mathbf{t}) &\propto \exp\left(-\frac{1}{2}(f(\mathbf{x}) - \hat{f}(\mathbf{x}))^T \mathbf{W}(f(\mathbf{x}) - \hat{f}(\mathbf{x}))\right) \\ &= \mathcal{N}(f(\mathbf{x})|f(\mathbf{x}_0, \mathbf{W}^{-1})), \end{aligned} \tag{18.21}$$

where $\mathbf{W} = -\nabla\nabla \log f(\mathbf{x})$.

In order to compute the Laplace approximation we need to find the value of $\mathbf{x}_0$ and then evaluate the Hessian matrix at that point. Identifying $\mathbf{x}_0$ can be done using the Newton–Raphson iteration, which finds an approximation to solutions of $f(x) = 0$ (in fact, here we want to find $f'(x) = 0$, but this doesn't change things much) by iterating the computation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{18.22}$$

until the changes are sufficiently small for the required accuracy.

18.2.2 Computing the Posterior

Returning to the actual computations that we need for the GP, we had reached the stage of approximating $p(f(\mathbf{x}^*)|\mathbf{x}, \mathbf{t}, \mathbf{x}^*)$. Using Bayes' rule we get that:

$$p(f(\mathbf{x})|\mathbf{x}, \mathbf{t}) = \frac{p(\mathbf{t}|f(\mathbf{x}))p(f(\mathbf{x})|\mathbf{x})}{p(\mathbf{t}|\mathbf{x})}. \tag{18.23}$$

We are in the lucky situation that the denominator is independent of $f()$, and so can be ignored for the optimisation. The first term in the numerator is:

$$p(\mathbf{t}|f(\mathbf{x})) = \prod_{i=1}^{N} \sigma(f(\mathbf{x}_i))^{t_n}(1-\sigma(f(\mathbf{x}_i))^{1-t_n}. \tag{18.24}$$

We will need to differentiate the log of this expression twice in order to use Equation (18.21):

$$\begin{aligned} \nabla \log p(\mathbf{t}|f(\mathbf{x})) &= \mathbf{t} - \sigma(f(\mathbf{x})) - \mathbf{K}^{-1}f(\mathbf{x}) && (18.25)\\ \nabla\nabla \log p(\mathbf{t}|f(\mathbf{x})) &= -\text{diag}(\sigma(f(\mathbf{x}))(1-\sigma(f(\mathbf{x})))) - \mathbf{K}^{-1}, && (18.26) \end{aligned}$$

where diag() puts the values along the diagonal of a zero matrix, and this term is the $\mathbf{W}$ matrix in Equation (18.21).

We now need to find the maximum of $\log p(\mathbf{t}|f(\mathbf{x}))$, for which we construct the Newton–Raphson iteration:

$$\begin{aligned} f(\mathbf{x})^{\text{new}} &= f(\mathbf{x}) - \nabla\nabla \log p(\mathbf{t}|f(\mathbf{x}))\\ &= f(\mathbf{x}) + (\mathbf{K}^{-1} + \mathbf{W})^{-1}(\nabla \log p(\mathbf{t}|f(\mathbf{x})) - \mathbf{K}^{-1}f(\mathbf{x}))\\ &= (\mathbf{K}^{-1} + \mathbf{W})^{-1}(\mathbf{W}f((x)) + \nabla \log p(\mathbf{t}|f(\mathbf{x}))). \end{aligned} \tag{18.27}$$

Thus, the Laplace approximation to the posterior probability is:

$$q(f(\mathbf{x})|\mathbf{x},\mathbf{t}) = \mathcal{N}(\hat{f}, (\mathbf{K}^{-1} + \mathbf{W})^{-1}). \tag{18.28}$$

Based on this, we can estimate the posterior mean and variance. For the mean, we need to use the fact that at the maximum of $\log p(\mathbf{t}|f(\mathbf{x}))$:

$$\hat{f}(\mathbf{x}) = \mathbf{K}(\nabla \log p(\mathbf{t}|\hat{f}(\mathbf{x}))), \tag{18.29}$$

and then the expressions for the mean and variance of the GP regression give us posterior distribution:

$$P(t^*|\mathbf{t},\mathbf{x},\mathbf{x}^*) \propto \mathcal{N}\left(\mathbf{k}^{*T}(\mathbf{t} - \sigma(f(\mathbf{x}))), k^{**} - \mathbf{k}^{*T}(\mathbf{K} + \mathbf{W}^{-1})^{-1}\mathbf{k}^*\right) \tag{18.30}$$

For the optimisation we will also need to calculate the log likelihood and the gradient of it with respect to each hyperparameter, just as we did for GP regression.

The log likelihood is:

$$\log p(\mathbf{t}|\mathbf{x},\boldsymbol{\theta}) = \int p(\mathbf{t}|f(\mathbf{x})p(f(\mathbf{x})|\boldsymbol{\theta})df(\mathbf{x}), \tag{18.31}$$

and so we again use the Laplace approximation to get:

$$\begin{aligned} \log p(\mathbf{t}|\mathbf{x},\boldsymbol{\theta}) &\approx \log q(\mathbf{t}|\mathbf{x},\boldsymbol{\theta})\\ &= \log p(\hat{f}(\mathbf{x})|\boldsymbol{\theta}) + \log p(\mathbf{t}|\hat{f}(\mathbf{x}) - \frac{1}{2}\log|\mathbf{W} + \mathbf{K}^{-1}| + \frac{N}{2}\log(2\pi). \end{aligned} \tag{18.32}$$

Differentiating this with respect to each of the hyperparameters will lead to two terms, since both $\hat{f}()$ and $\mathbf{K}$ depend on $\boldsymbol{\theta}$. The same matrix identities as for the regression case are

useful, and the first part, which is the explicit dependence upon any element of $\boldsymbol{\theta}$ is fairly similar to the regression case:

$$\frac{\partial}{\partial \boldsymbol{\theta}_j} \log p(\mathbf{t}|\boldsymbol{\theta})\bigg|_{\text{explicit}} = \frac{1}{2}\hat{f}(\mathbf{x})^T\mathbf{K}^{-1}\frac{\partial \mathbf{K}}{\partial \boldsymbol{\theta}_j}\mathbf{K}^{-1}\hat{f}(\mathbf{x}) - \frac{1}{2}\text{trace}\left((\mathbf{I}+\mathbf{KW})^{-1}\mathbf{W}\frac{\partial \mathbf{K}}{\partial \boldsymbol{\theta}_j}\right) \tag{18.33}$$

We can then use the chain rule to get the other parts: $\frac{\partial}{\partial \boldsymbol{\theta}_j} = \frac{\partial}{\partial \hat{f}}\frac{\partial \hat{f}}{\partial \boldsymbol{\theta}_j}$, where:

$$\frac{\partial \hat{f}}{\partial \boldsymbol{\theta}_j} = (\mathbf{I}+\mathbf{WK})^{-1}\frac{\partial \mathbf{K}}{\partial \boldsymbol{\theta}_j}(\mathbf{t}-\sigma(\hat{f}(\mathbf{x})), \tag{18.34}$$

and so we just need to compute:

$$\begin{aligned}&\frac{\partial}{\partial \hat{f}(\mathbf{x}_i)}\log|\mathbf{W}+\mathbf{K}^{-1}| \\ &= \left((\mathbf{I}+\mathbf{WK})^{-1}\mathbf{K}\right)_{ii}\sigma(\hat{f}(\mathbf{x}_i))(1-\sigma(\hat{f}(\mathbf{x}_i)))(1-2\sigma(\hat{f}(\mathbf{x}_i))\frac{\partial \hat{f}(\mathbf{x}_i)}{\partial \boldsymbol{\theta}_j}\end{aligned} \tag{18.35}$$

Note that this includes the third derivative of the $\sigma(\cdot)$ term!

Putting these three terms together gives us the whole gradient, ready for the conjugate gradient solver.

18.2.3 Implementation

The algorithm can be written out from the previous discussion, but as with the regression case, there are some tricks that can be used to improve the computational time and stability. The main one is that the matrix $(\mathbf{K}+\mathbf{W}^{-1})$ can be inverted using another matrix identity:

$$(\mathbf{K}+\mathbf{W}^{-1})^{-1} = \mathbf{K} - \mathbf{K}\mathbf{W}^{\frac{1}{2}}\mathbf{B}^{-1}\mathbf{W}^{\frac{1}{2}}\mathbf{K}, \tag{18.36}$$

where $\cdot^{\frac{1}{2}}$ means the element-wise square root and $\mathbf{B}$ is the symmetric positive definite matrix

$$B = \mathbf{I} + \mathbf{W}^{\frac{1}{2}}\mathbf{K}\mathbf{W}^{\frac{1}{2}}. \tag{18.37}$$

To make implementation easier, the algorithm is written out here in these computationally efficient terms:

Gaussian Process Classification

- **To find the maximum by Newton–Raphson iteration:**
  - compute the covariance matrix $\mathbf{K} = k(\mathbf{X},\mathbf{X}) + \sigma_n\mathbf{I}$ for hyperparameters $\boldsymbol{\theta}$
  - repeat until change < tolerance:
    - $W = -\nabla\nabla \log p(f(\mathbf{x}))$
    - $L = \text{cholesky}(\mathbf{I} + \mathbf{W}^{\frac{1}{2}}\mathbf{K}\mathbf{W}^{\frac{1}{2}})$
    - update f using Equation (18.27), with Equation (18.36) giving the form of the inverse matrix
    - change = oldf - f

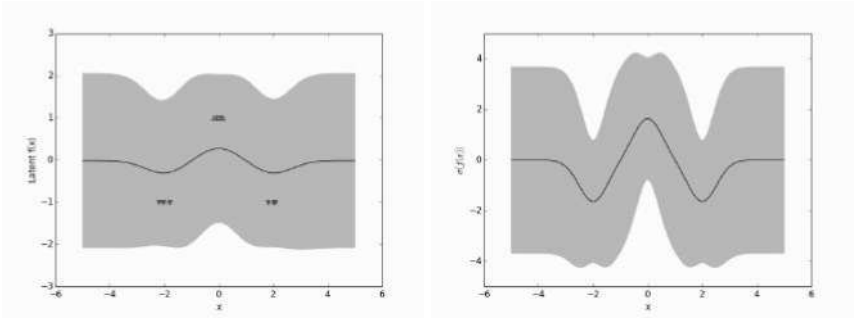

FIGURE 18.8 Gaussian process classification for a very simple dataset (shown plotted in the figure on the left). The latent function can be seen on the left, and the output of the logistic function on the right.

- **To make a prediction:**
  - compute the covariance matrix $\mathbf{k}^* = k(\mathbf{x}^*, \mathbf{X})$
  - compute the covariance matrix $k^{**} = k(\mathbf{x}^*, \mathbf{x}^*)$
  - compute the maximum f^* using the Newton–Raphson iteration algorithm
  - the mean of the process is $\mathbf{k}^* \nabla \log p(f(\mathbf{x}))$
  - solve $\mathbf{L}\mathbf{v} = \mathbf{W}^{\frac{1}{2}}\mathbf{k}^*$ for $\mathbf{v}$
  - the variance is $k^{**} - \mathbf{v}^T\mathbf{v}$
- **To compute the log likelihood and gradient:**
  - compute log likelihood using Equation (18.31)
  - compute $\mathbf{R} = \mathbf{W}^{\frac{1}{2}}\mathbf{B}^{-1}\mathbf{W}^{\frac{1}{2}}$, where $\mathbf{B}$ is defined in Equation (18.37).
  - compute $\mathbf{s}_2 = \frac{\partial}{\partial \hat{f}(\mathbf{x})} \log q$ using Equation (18.35)
  - for each hyperparameter $\boldsymbol{\theta}_j$:
    - compute gradients of covariance matrix with respect to $\boldsymbol{\theta}_j$
    - compute explicit gradient $s_1 = \frac{\partial}{\partial \boldsymbol{\theta}_j} \log p(\mathbf{t}|\boldsymbol{\theta})$ using Equation (18.33)
    - compute $\mathbf{s}_2 = \frac{\partial \hat{f}}{\partial \boldsymbol{\theta}_j}$ using Equation (18.34)
    - full gradient of log likelihood for $\boldsymbol{\theta}_j$ is $s_1 + \mathbf{s}_2^T\mathbf{s}_3$

Figure 18.8 shows a very simple example of Gaussian process classification. The data consists of a few points at around $x = -2$ and $x = +2$ that belong to one class and a few at around $x = 0$ that belong to the other class.

It is possible to do multi-class classification with GPs. The basic idea is to use a separate latent function for each class (so that the function $f(\mathbf{x})$ gets c times longer for c classes), looking like:

$$(f_1^{C_1}, f_2^{C_1}, \ldots f_n^{C_1}, f_1^{C_2}, f_2^{C_2}, \ldots f_n^{C_2}, \ldots f_1^{C_c}, f_2^{C_c}, \ldots f_n^{C_c}). \tag{18.38}$$

The target vector has to be the same dimension, so it will contain a row of n 1s where the

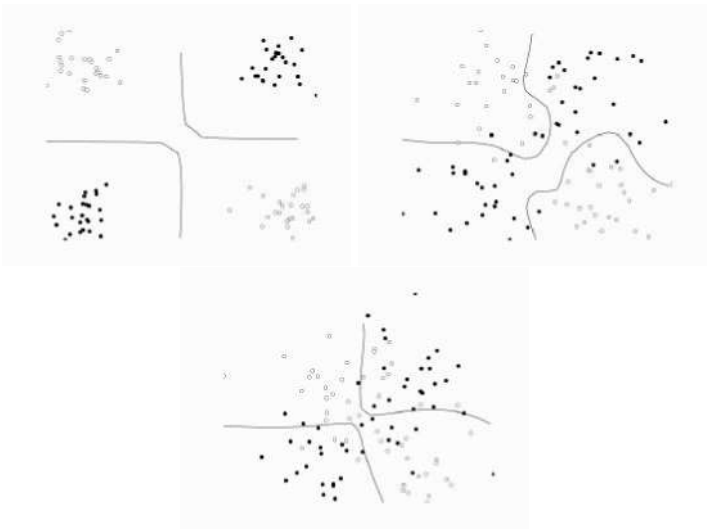

FIGURE 18.9 Gaussian process classification on the modified XOR dataset, with standard deviations $\sigma = 0.1$ (*left*), $\sigma = 0.3$ (middle), $\sigma = 0.4$ (*right*). *The line gives the* $p = 0.5$ *decision boundary.*

f_i for the correct target class are, and 0 everywhere else. The covariance function will then be represented by a set of blocks of the individual covariance matrices. It is also necessary to use soft-max instead of the logistic function to do the 'squashing' of the regression output, which changes the derivatives in the computation of the log likelihood and its gradients. For further details on this, see the references in the Further Reading section.

There has been a lot more work on Gaussian processes over the past 10 years, including far more sophisticated optimisation methods, better ways to perform multi-class classification, and a better understanding of the links between Gaussian processes and neural networks, splines, and many other topics, but they are beyond our scope here: for more information, consult the references in the Further Reading section.

For a fairly simple idea, Gaussian processes do tend to work very well on a wide range of topics, and the way that the covariance function explicitly encodes the correlations that can be seen in the data means that the user has a lot of control. Even in the simple treatment here we have put quite a lot of effort into making the computations numerically stable and relatively fast. However, there is much more that can be done, including methods for approximation to speed things up significantly. Again, the Further Reading section provides more detail.

FURTHER READING

There is a very readable book dedicated solely to Gaussian Processes, which is:

- Carl Edward Rasmussen and Christopher K.I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, USA, 2006.

Another summary that may be useful is:

- D. MacKay. Neural networks and machine learning. *NATO ASI Series, Series F, Computer and Systems Sciences*, 168:133–166, 1998.

and GPs are also covered in:

- D.J.C. MacKay. (Chapter 45) *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, Cambridge, UK, 2003.
- C.M. Bishop. (Section 6.4) *Pattern Recognition and Machine Learning*. Springer, Berlin, Germany, 2006.

PRACTICE QUESTIONS

Problem 18.1 The current implementation only has the squared exponential kernel in. Implement some more of those listed in Section 18.1.5 and experiment with them, particularly with the Palmerston North ozone layer dataset that we saw in Section 4.4.4. You might find the example in Section 5.4.3 of Rasmussen and Williams helpful.

Problem 18.2 Compare the optimisation results with using other optimisers, such as BFGS.

Problem 18.3 A simple version of multiclass classification uses one-against-all classifiers as we did with the SVM. Implement that and see how well it works on the iris dataset.