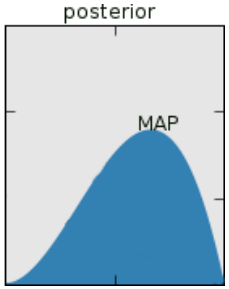
Gaussian processes

$$
\begin{aligned}
P(\text{Head}|\mathcal{D}) &= \int P(\text{head}, b|\mathcal{D})\ db \\
&= \int P(\text{head}|b, \mathcal{D})\ P(b|\mathcal{D})\ db \\
&= \int P(\text{head}|b)\ P(b|\mathcal{D})\ db \\
&= \int b\ P(b|\mathcal{D})\ db
\end{aligned}
$$



In any model (eg. a neural net) we have parameters θ (eg. weights) we start off unsure about (prior: $P(\theta)$), but which the data $\mathcal{D}$ do inform ($\longrightarrow P(\theta\ \mathcal{D})$, the "posterior").

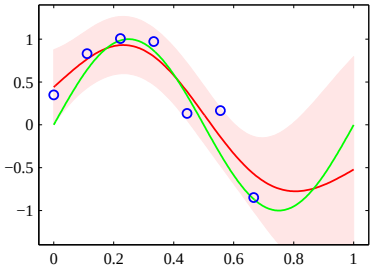
- minimize the error? [argmax of the likelihood]
- minimize the error + penalty? [argmax of the posterior, "MAP"]
- don't optimize at all: just integrate them out ["Bayesian"]

Ways to do the Bayesian integrals

- exactly, analytically (is possible for the above), or
- brute force, up to some discretization error (fine here, but cursed...), or
- approximate it via sampling (eg. Rejection Sampling fine since hypothesis space is low dimensional. Else MCMC).
- analytically, *for a simpler distribution* - find a simpler distribution that approximates the true posterior, and integrate over *that* instead ("variational" methods). It's a great idea, but we won't pursue it here - tends to be case-specific, requires mathematical sophistication.

a "Gaussian process"

What if we just assumed the posterior is always Gaussian, like this perhaps:



Lighter line (green) is ground truth, and circles are the data. Darker line (red) is the mean of the posterior.
The shaded area represents "error bars", ie. $\sqrt{\ }$ of the variance.

1D Gaussian

A 1D Gaussian distribution with zero mean is

$$p(y) = \frac{1}{Z} \exp(-\frac{y^2}{2\sigma^2})$$

which can be written as

$$= \frac{1}{Z} \exp(-\tfrac{1}{2}\ y\, C^{-1}\ y)$$

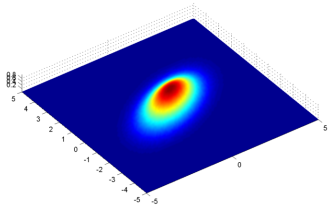
where C is the **variance**, σ^2.

a zero-mean, 2D Gaussian

$$p(\mathbf{y}) = \frac{1}{Z}\exp(-\tfrac{1}{2}\,\mathbf{y}^T\,\mathbf{C}^{-1}\,\mathbf{y})$$

- $\mathbf{y}$ is a 2D vector now,
- T stands for "transpose",
- $\mathbf{C}$ is a **covariance matrix**,

$$\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$



- $\mathbf{C}$ can be any matrix that is *positive definite*, meaning that the scalar number given by $\mathbf{y}^T\mathbf{C}^{-1}\mathbf{y} \geq 0, \quad \forall \mathbf{y}$
- note that we need to *invert* $\mathbf{C}$ - this scales as n^3.

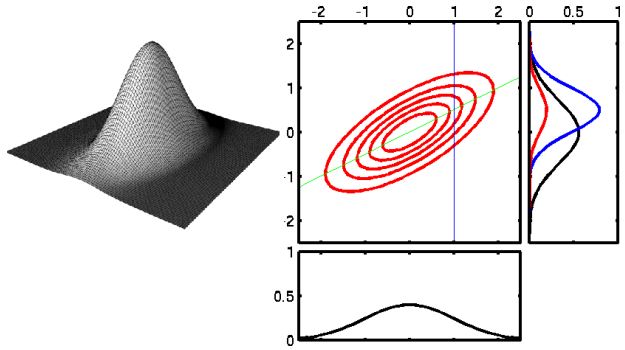
effect of the covariances

- if the off-diagonal terms are zero, there is no covariance between y_1 and y_2: knowing one tells you nothing at all about the other. So the blob is a spherical Gaussian in this case.

- non-zero off-diagonal terms mean the blob is squished or smeared out.

- **if** $C_{11} = C_{22}$, the variance of each dimension is the same, so the blob lies "on the diagonal"
- in that case, the off-diagonal term tells you how long and thin the distribution is.

recap: a unique property of the Gaussian



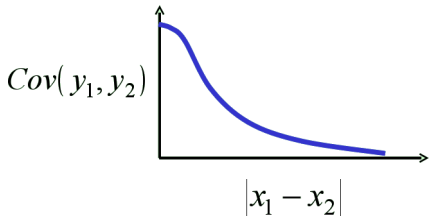
Going along any straight line through a multidimensional Gaussian yeilds a 1D Gaussian!

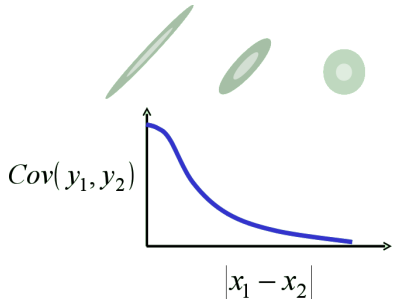
So for example $p(y_2|y_1)$ is (1D) Gaussian, for any particular y_1. (But the mean and variance do depend on y_1).

covariance as a function of distance

Let's say y_1 and y_2 are the outputs corresponding to two input vectors, $\mathbf{x}_1$ and $\mathbf{x}_2$. Say the distance between $\mathbf{x}_1$ and $\mathbf{x}_2$ is d_{12}.

- If d is very small then y_1 and y_2 covary strongly, so C_{12} in the matrix should be big.
- If d is large then y_1 and y_2 sould become independent, so C_{12} in the matrix should tend to zero.





We just need a couple of hyperparameters, controlling the shape of this:

- θ_{vert} for the vertical scale of variations,
- θ_{length} for the horizontal scale of variations: there should be one of these for each input dimension,
- θ_{noise} for the amount of additive noise.

Others are also possible, e.g. a linear trend, periodic trends, etc...

an example covariance function

We could parameterize this via a **covariance function** - some simple function of d. An example is this one:

$$C(y_1, y_2) = \theta_{vert} \exp(-\frac{|x_1 - x_2|^2}{2\theta_{length}^2})$$

Nb. this is just one option and is *not* where Gaussian processes get their name though...

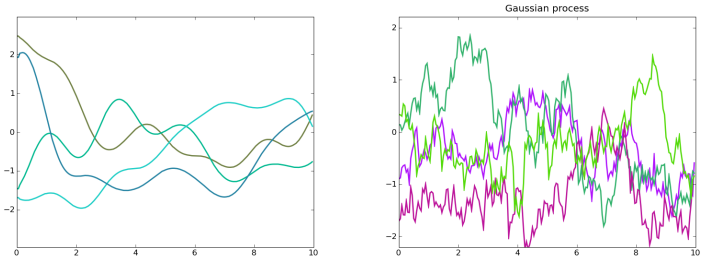
Notice it's symmetric: $C_{12} = C_{21}$. The two θ's are known as **hyperparameters**:

- θ_{vert} says how strongly we think y values should covary when the same input is given twice. *Example*: $\theta_{vert} = 1$
- θ_{length} controls how fast this dies away as d increases. *Example*: $\theta_{length} = 0.5$

We might also have a third hyperparameter, θ_{noise}, that specifies how much noise we think is corrupting the "true" values before we get to measure them as y.

Covariance function defines a prior over curves

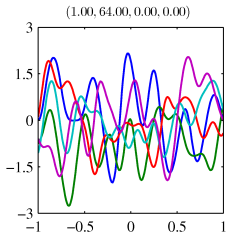
By specifying $C(x, x')$ via hyperparameters θ, we define a prior over the outputs given *any* set of inputs.



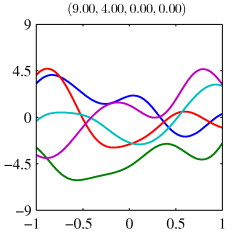
On the left are samples drawn using the most commonly used covariance function: a squared-exponential. Those on the right used an exponential covariance function: these are "Brownian motion" random walks.

effect of hyperparameters

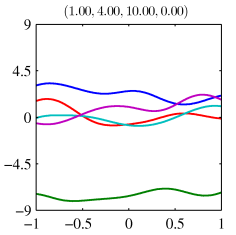
small θ_{length}



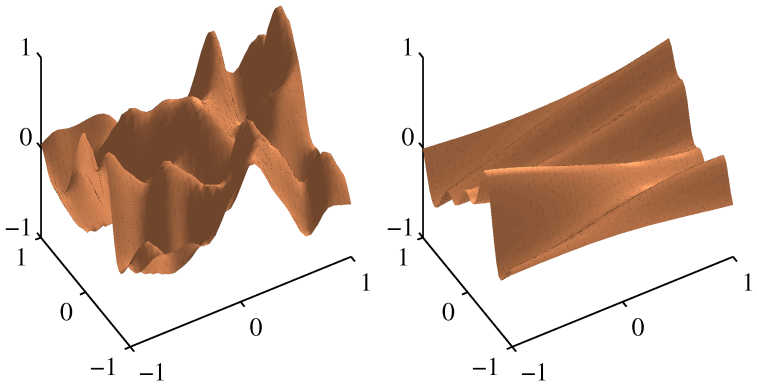
big θ_{vert}



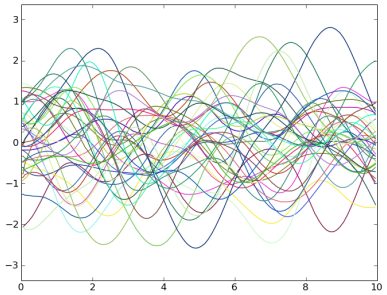
big θ_{noise}



different dimensions, different length scales



samples from Gaussian Process prior: lots...

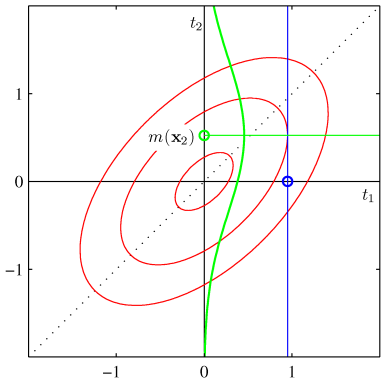


Doing this enough, you'd get a Gaussian-shaped histogram for any vertical slice.

Data: it's *like* we want to kill off all the curves that don't go through the data. That sounds impossible...

inference / regression / supervised learning

- Say we've observed $\mathbf{x}_1$ and its output $y_1 = t_1$. (t_1 is observed value, y is random variable).
- given some *new* input $\mathbf{x}_2$, we want to know the distribution $p(y_2|y_1 = t_1)$: a "slice" through the joint distribution along a line corresponding to the observed y_1 value.



doing it with more than 1 data point..!

Nothing we've said is intrinsically limited to 2 dimensional Gaussian distributions.

Consider a whole set of (say n) input $\mathbf{x}$'s, and their observed (y) values $t_{1\ldots n}$.

1. use the covariance function $C(x, x')$ to fill in the entries for a matrix using all the $\mathbf{x}_{1\ldots n}$ for which we know the output AND the new $\mathbf{x}'$ we're interested in. This generates an $(n+1)$-by-$(n+1)$ covariance matrix $\mathbf{C}$.
2. invert $\mathbf{C}$ to give a joint distribution for all the $n+1$ corresponding outputs, $\mathbf{y} = (y_1, \ldots, y_n, y')$, namely $p(\mathbf{y}) = \frac{1}{Z}\exp(-\frac{1}{2}\mathbf{y}\mathbf{C}^{-1}\mathbf{y}^T)$: this is an $n+1$ dimensional Gaussian distribution.
3. we *know* the y values for all but one of those dimensions, so condition on them to yeild a 1D Gaussian prediction (tedious linear algebra omitted).

the general result

With a whole set of (say n) input $\mathbf{x}$'s, and their observed (y) values $t_{1\ldots n}$, the result is:

Gaussian process prediction at $\mathbf{x}$:

$$\texttt{mean} = \mathbf{k}^T\mathbf{C}^{-1}\mathbf{t}$$
$$\texttt{variance} = \kappa - \mathbf{k}^T\mathbf{C}^{-1}\mathbf{k}$$

where

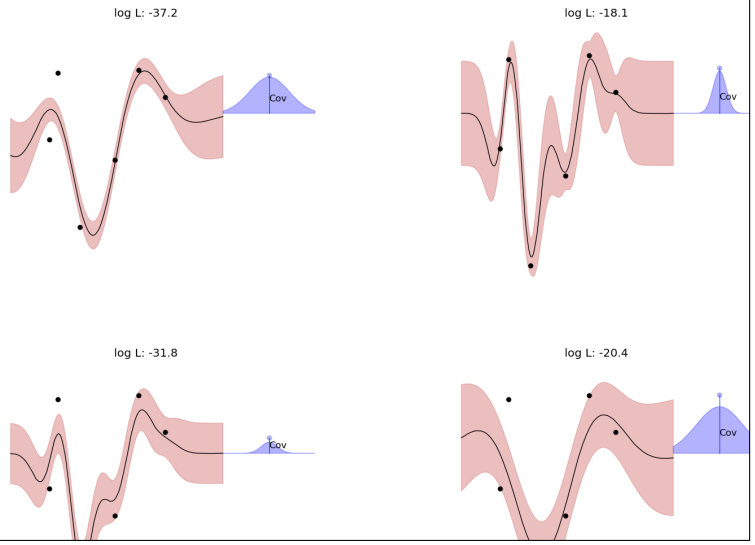
- $\mathbf{C}_{ij} = Cov(\mathbf{x}_i, \mathbf{x}_j)$
- $\mathbf{k}$ is the vector of individual covariances $\mathrm{k}_j = Cov(\mathbf{x}_j, \mathbf{x})$ between the new input $\mathbf{x}$ and each of those in the data set.
- κ is $Cov(x, x)$. In our case it's θ_{vert}.

See *MacKay* or *Bishop* for the details.

cost of matrix inversion

- we need to invert the covariance matrix, which is a square matrix of size n. Matrix inversion scales as n^3 in general, which limits the above Gaussian process calculation to smallish datasets (n up to about 1000).
- But there are a number of ways of coping with the n^3 scaling issue that allow one to deal with several thousand training items - e.g. I recently saw one with $n = 44000$.
- One can also use Gaussian processes to perform Bayesian classification (as opposed to regression), although the story isn't quite as clean and we're forced to fall back on MCMC techniques for part of the resulting calculations.

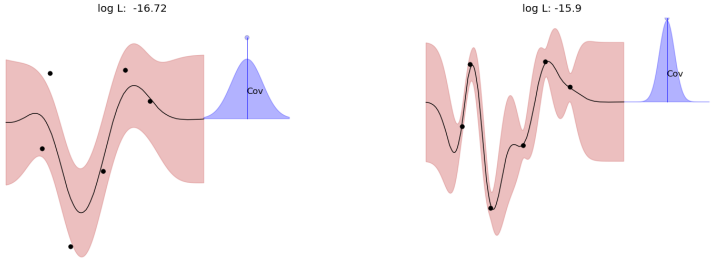
other hyperparameters



learned (MAP) hyperparameters

I used conjugate gradient[1] on $\log P(\theta|\mathcal{D})$ to optimize θ.

In this case there were 2 local optima:



[1] a smarter form of gradient ascent