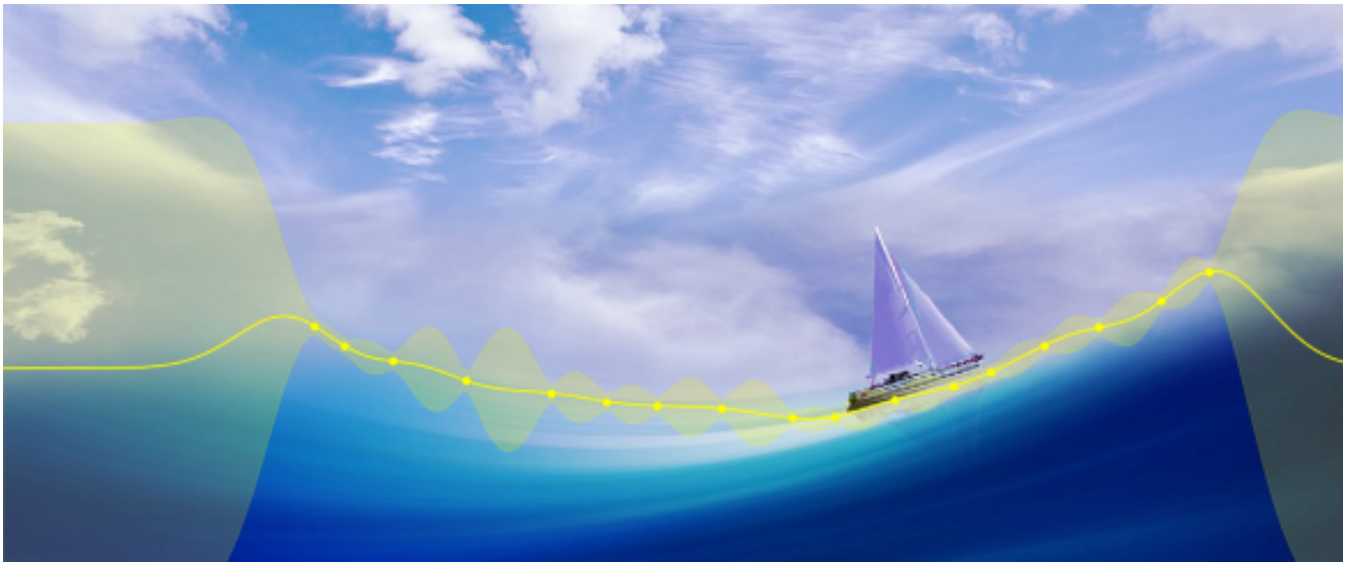
Understanding Gaussian Process, the Socratic Way

Wei Yi Dec 1, 2019 · 67 min read ★



Gaussian Process is a machine learning technique. You can use it to do regression, classification, among many other things. Being a Bayesian method, Gaussian Process makes predictions with uncertainty. For example, it will predict that tomorrow's stock price is $100, with a standard deviation of $30. Knowing uncertainty is important for applications such as algorithmic trading. I have designed trading strategies that made a fortune and strategies that lost a fortune. I understand first-hand how much uncertainty can do to you. So I decided to study the inner workings of Gaussian Process.

Socratic method, by asking questions and answering them.

I will use the following template to carry out the explanations:

1. The **Gaussian Process model** section defines the Gaussian Process prior and the likelihood. And it explains the model parameters in the prior and the likelihood.
2. The **Computing the posterior** section derives the posterior from the prior and the likelihood. And it describes how to make predictions using the posterior.
3. The **Parameter learning** section finds optimal concrete values for model parameters. The posterior from the previous section is a symbolic expression that mentions model parameters. After we found concrete values for model parameters, we can evaluate the posterior into a concrete number.

This article serves as the basic introduction to Gaussian Process; it helps you build up the first GP-sense in your head. In an application setting, you may come across two tricks that make a Gaussian Process model more practical:

- Variational Gaussian Process — What To Do When Things Are Not Gaussian introduces variational inference to allow us to use a non-Gaussian likelihood in a Gaussian Process model.
- Sparse and Variational Gaussian Process — What To Do When Data is Large introduces inducing variables to allow us to scale a Gaussian Process model to large datasets.

In my experience, understanding the basics in this article accounts for at least 50% of the conceptual burden. After you established sufficient GP-sense, it is much easier to understand advanced Gaussian Process models.

Some notations

Medium supports Unicode in text. This allows me to write many math subscript notations such as X_1 and X_n. But I could not write down some other subscripts. For example:

$$X_* \text{ or } X_{*1}$$

If some math notations render as question marks on your phone, please try to read this article from a computer. This is a known Unicode rendering issue.

Gaussian Process model

This section introduces the Gaussian Process model for regression. It solves a regression task.

The regression task

In a regression task, we have a set of training data points in pairs (X_1, Y_1), (X_2, Y_2), …, (X_n, Y_n), where X_i, Y_i, are real values. We use the symbol $\mathbb{R}$ to denote the set of all real values. And use (X, Y) to denote the training data, where X and Y are both vectors of length n.

The task is to find a function f that takes a real value and returns a real value, denoted as f: $\mathbb{R} \longmapsto \mathbb{R}$. This function should be close to the training data (X, Y). "close" means that if we plug in a X_i in f, we should get a value close to Y_i. This is a typical regression task in machine learning.

The mapping view to define functions

We are used to seeing a function with a body, like $f(x) = x+1$, with body "$x+1$". This body computes the function value from its input x. Many machine learning techniques find the body for functions. For example, linear regression finds the function body in the form of $f(x) = ax+b$.

But in math, we can define a function without anybody. A function is a mapping from its inputs to outputs. The body is merely a concise way to describe this mapping. Instead of using a body, we can define a function by specifying all its input-output mappings. For example, the following mappings define a function f: $\mathbb{R} \longmapsto \mathbb{R}$ with domain {1, 5}, and range {7, 4}:

$$f : \begin{bmatrix} X_1 = 1 \\ X_2 = 5 \end{bmatrix} \mapsto \begin{bmatrix} Y_1 = 7 \\ Y_2 = 4 \end{bmatrix}$$

We don't know what the body for this function is, but that does not stop us from defining it— we just need to write down the mapping from each input to its output. We

Mappings as functions in real life

You may ask, do we use this mapping view of functions in everyday life? Yes, we use it all the time. At school, we saw the trigonometric table:

Angle θ				
Degrees	Radians	$\sin\theta$	$\cos\theta$	$\tan\theta$
0	0	0	1	0
30	$\frac{\pi}{6}$	$\frac{1}{2}$	$\frac{\sqrt{3}}{2}$	$\frac{1}{\sqrt{3}}$
45	$\frac{\pi}{4}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$	1
60	$\frac{\pi}{3}$	$\frac{\sqrt{3}}{2}$	$\frac{1}{2}$	$\sqrt{3}$
90	$\frac{\pi}{2}$	1	0	undefined
180	π	0	−1	0
270	$\frac{3\pi}{2}$	−1	0	undefined
360	2π	0	1	0

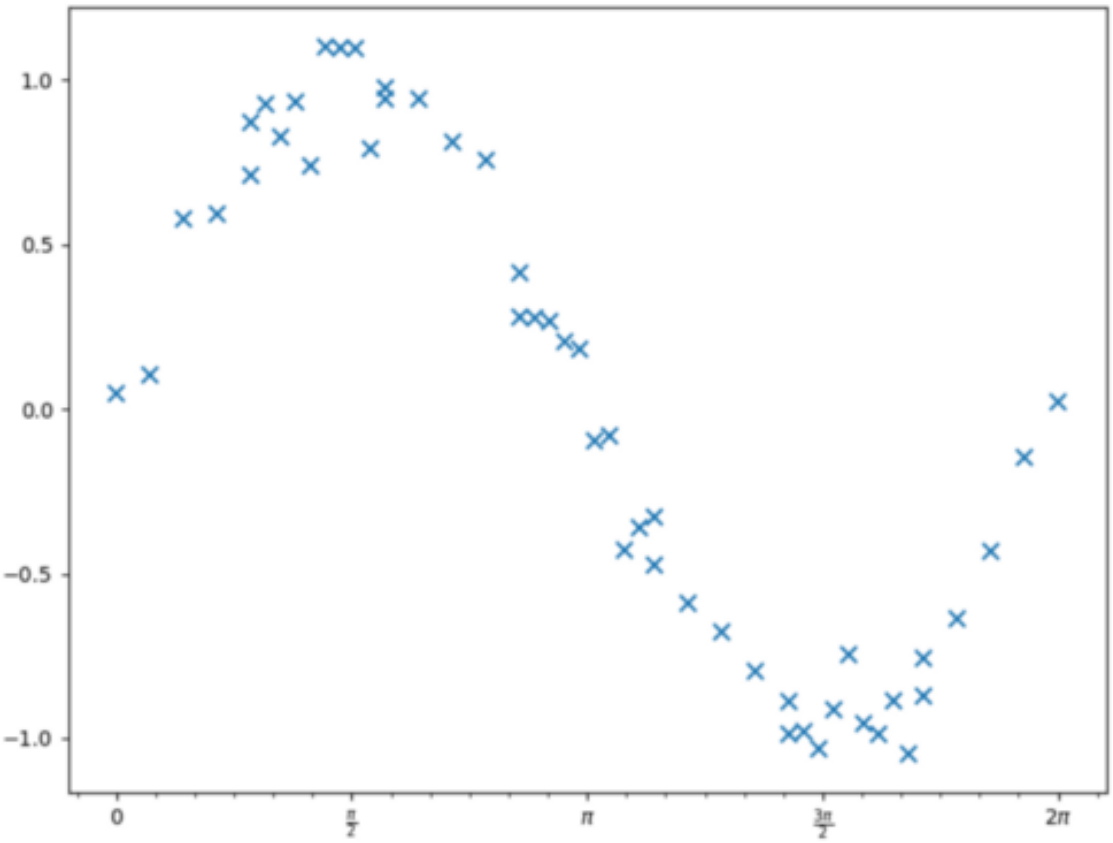
The third column of the table shows the sine function value at radius 0, π/2, and so on. Your elementary math book gives you this table instead of a function body to calculate the sine value for arbitrary degree because that formula is very complex — think about it, you don't really know how to compute the sine function value from an input, say π — you remember its value, that's an input-output mapping view.

The downside of this table is obvious, you don't know the value of sine at π/5, or 2.7π. The table only lists a subset input-output mappings for space reasons. That's where regression can help — it finds a full mapping.

In this article, we will use a Gaussian Process to learn a function that looks like a sine function. The mapping view to define functions is the intuition behind Gaussian Process.

First, let's generate some training data. We compute true sine function values at some x-axis locations between 0 and 2π and add noise to them. The x-axis locations are not

points. There are 50 points. I used this code to generate the training data. If you want to run the code, please clone the public understanding_gaussian_process Github repository and run the code with a Python 3 interpreter.



Training data points. Blue “x” markers are the data points. They are not equidistant.

Our regression task is to learn a function that explains this data well. We already know the true answer — the function is a sine function. We will see how well Gaussian Process can learn this function from the generated training data.

Sets of real numbers representing locations

The function f: $\mathbb{R} \mapsto \mathbb{R}$ has $\mathbb{R}$ as its domain, meaning it accepts any real value input. In our regression task, an input x is a location where we want f to approximate the value of $sin(x)$. We can split the domain $\mathbb{R}$ into three disjoint sets of locations:

- X is the set of locations that we have in our training data. X is a vector of length n:

$$\begin{bmatrix} \dots \\ X_n \end{bmatrix}$$

- X_* the set of test locations where we want to evaluate our underlying function f. Remember the goal of this regression task is to find f so we can get its values at test locations. X_* is finite because practically, in our lifetime, we can only evaluate a function at a finite number of test locations. X_* is a vector of length n_*:

$$\begin{bmatrix} X_{*1} \\ X_{*2} \\ X_{*3} \\ \dots \\ X_{*n_*} \end{bmatrix}$$

- X_o (the subscript is the letter "*o*" standing for *other,* not the number 0) is the set of locations other than X and X_* in $\mathbb{R}$. The set X_o has infinite length because, in the full real line, there are infinite numbers other than X and X_*. Write down X_o as a vector of infinite length n_o:

$$\begin{bmatrix} X_{o1} \\ X_{o2} \\ X_{o3} \\ \dots \\ X_{on_o} \end{bmatrix}$$

Random variables representing function values

After we defined the input to the function f, we move to define its output. For each input location x, we introduce a random variable $f(x)$ that represents the *possible* output of the function f at this location. You can understand $f(x)$ like this:

1. The random variable $f(x)$ has its distribution, we have not defined this distribution yet, but spoiler alert, it will be Gaussian. We use this distribution to model the possible values and their appearing probabilities that f can take at location x. A Gaussian distribution is completely parameterized by its mean and variance.

variance of *f(x)* controls how much this sample can deviate from the mean of *f(x)*. Note the word "eventually", as we will introduce another random variable to sample observations from. The reason will be clear in the likelihood section.

You can see this modeling fits perfectly with how we generated our training data *(X, Y)*. Because we generated *Y* by first getting the true sine values (that is the mean) at locations *X*, and added noise (which simulates the variance).

Note *f(x)* is just the name of the random variable at location *x*. Please do not interpret *f(x)* as a function application of calling the function *f* with argument *x*. I decided to include the location information in random variable names to make the math easier to understand.

Since we are modelling a function with an infinite number of inputs, we need to introduce an infinite number of random variables to represent its output, one for each input location. Corresponding to the three parts of locations we defined before, we organize these random variables into three corresponding disjoint sets:

- *f(X),* a vector of random variables of length *n*,
- *f(X_*)*, a vector of random variables of length *n_**.
- $f(X_o)$, a vector of random variables of infinite length n_o.

Function values are dependent on each other

Now that we have introduced random variables for function values, we need to assume that they are dependent on each other. This is a **very important** assumption. Without it, we cannot continue.

This is because if instead, we assume these random variables are independent of each other, then knowing the observation data *Y* for random variable *f(X)* at training location *X* will not give any information about the possible random variable values *f(X_*)* at test locations *X_**. This implies that we will not be able to learn a regression function that can describe values at test locations.

So we need a way to describe the dependency relationships among random variables. In Gaussian Process, we use the multivariate Gaussian distribution over the the random variables *f(X)*, *f(X_*)* and $f(X_o)$ to define their correlations, as well as their means.

$$\begin{bmatrix} f(X) \\ f(X_*) \\ f(X_O) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(X) \\ m(X_*) \\ m(X_O) \end{bmatrix}, \begin{bmatrix} k(X,X) & k(X,X_*) & k(X,X_O) \\ k(X,X_*)^\top, & k(X_*,X_*) & k(X_*,X_O) \\ k(X,X_O)^\top & k(X_*,X_O)^\top & k(X_O,X_O) \end{bmatrix} \right)$$

mean vector — covariance matrix

The above is the template of a multivariate Gaussian distribution. To fully define this distribution, we need to define:

The mean function *m* that gives the mean of those random variables. We use *m(x)* to express the mean, also known as the expected value of the random variable *f(x)* at location *x*. Since *x* represents a set of locations, *m(x)* is in the form of vectors, containing mean values for set of random variables. So you will have:

1. the mean vector *m(X)* for random variable *f(X)* when *x=X*,
2. the mean vector *m(X_*)* for random variable *f(X_*)* when *x=X_**,
3. the mean vector $m(X_o)$ for random variable $f(X_o)$ when *x=O*.

Note *m(x)* is not a random variable, it is a function that takes *x* as its argument and returns the mean values for the corresponding random variables at location *x*.

We also need to define the covariance function *k* appearing in the covariance matrix. We call *k* a **kernel** function. We use *k* to define the covariance between every two random variables *f(x)* and *f(x′)* at location *x* and *x′*. You will understand that choosing *k* is the key in Gaussian Process.

The mean function m

The mean function defines the expected value for each random variable in the vector $[f(X), f(X_*), f(X_o)]^T$. Here "^{T}" stands for the transpose operator, which turns a column vector into a row vector so we can write it in a text line. The mean function *m* has domain $\mathbb{R}$ and range $\mathbb{R}$, that is, $m: \mathbb{R} \mapsto \mathbb{R}$. When we apply *m* to each of the above locations, we got a mean value vector:

$$\begin{bmatrix} \\ m(X_o) \end{bmatrix}$$

What should m look like? If we have domain knowledge about the expected values of the function f at every location, we can encode this knowledge in m. For example, if you are modeling the temperature at given time X of a server room and you know the mean temperature because you set the air conditioner temperature to 20 degrees, you can set $m(X) = 20$. But in most cases, we have no idea. So we define the mean function to be a zero function:

$$m(x) = 0$$

This function returns 0 no matter what value x is. You may wonder why a zero mean function is OK, since real data may not have zero mean. Well, you can always normalize your data so they have zero mean. In most cases, this works. But from time to time, you do need to think harder about a non-zero mean function to derive a better model.

Note unlike $f(X)$, $m(X)$ is not a random variable, it is a function that takes X as its argument and returns a real vector. We usually set $m(X) = 0$.

The kernel function k

Each entry in the covariance matrix represents the covariance between a pair of random variables $f(x)$ and $f(x')$. x and x' can come from either the location set X or X_*. We need to define a function $k(x, x')$ to return the covariance between $f(x)$ and $f(x')$. How should k look like?

From our training data, we notice if two locations x and x' are close by, their corresponding function values are similar. This is how smooth function behaves. And we want to model this effect of similar function values when their evaluation locations are close-by.

Remember that we use random variable $f(x)$ and $f(x')$ to model the possible values of the function f at location x and x'. In the language of random variables, similar means to have high positive covariance. This is because when $f(x)$ and $f(x')$ have high positive

values at close-by locations make up points in a smooth curve.

Many functions of two arguments x and x' satisfy our requirement to return a larger positive value when x and x' are close-by and return a smaller value when they are far away. And we focus on the following one:

In this formula, *exp* is the exponential function. l is called the lengthscale, and σ^2 the signal variance. l and σ^2 are scalars. They are **model parameters**. We don't know the values for these model parameters; we will use parameter learning to find good values for them. This kernel function is called the squared exponential kernel because of its structure.

Please note that the kernel function only mentions x and x'. It does not mention function values Y at all. This is because to decide the distance between two locations, knowing x and x' is enough. Don't worry, we will use Y when we introduce the likelihood.

We can verify that this kernel function satisfies our requirement:

1. the exponential function and σ^2 are both positive, so the covariance is always positive.
2. k has its maximum σ^2 when x is equal to x', so their distance is the smallest 0. The exponential function evaluates to 1, so the kernel function evaluates to σ^2.
3. as x and x' drift apart, the kernel function decreases and reaches its minimum 0 when x and x' are infinitely apart.

Now we can define the elements inside the covariance matrix:

We need to use k to define the nine block entries inside this covariance matrix. Let's work on $k(X, X)$ as an example, and the rest entries are similar.

Let's denote $k(X, X)$ to be the $n \times n$ matrix where we apply the function k to every pair of locations in X. And we place the pairs in the following exact order:

Similarly, we can define the $n \times n\_*$ matrix $k(X, X\_*)$. Due to the definition of the kernel function, $k(x, x') = k(x', x)$, the covariance matrix is symmetric. So we have the covariance $k(X\_*, X) = k(X, X\_*)^T$. And we define the $n\_* \times n\_*$ matrix $k(X\_*, X\_*)$.

We can continue to write down the five entries related to X_o, such as $k(X, X_o)$ and $k(X\_*, X_o)$. Those entries will be matrices of infinite dimension because the length of X_o is infinite. Writing down matrices of infinite dimensions does not seem to be an interesting task. Manipulating them also does not seem easy.

The good news is, we don't have to deal with those infinite matrices. This is because, in practice, we are only interested in the parts related to X and X_*. And these parts are finite. So we only need the parts from the joint probability density highlighted with red boxes below:

The Gaussian marginalization rule tells us that the highlighted parts also form a multivariate Gaussian distribution. This distribution is over the random variables $f(X)$ and $f(X\_*)$ with mean and covariance matrix exactly as highlighted in the red boxes. So we have the following finite distribution to work with from now on:

Alternatively, we can represent this distribution using its probability density function:

In the above formula, I use K to represent the whole covariance matrix to save space. $det(K)$ means the determinant of the covariance matrix K, K^{-1} means the inverse of K. The power to the constant 2π is $(n+n\_*)/2$ because the length of $[f(X), f(X\_*)]^T$ is $n+n\_*$ — since the GP prior, which is a $n+n\_*$ dimensional Gaussian distribution, dedicates an individual dimension to each data point, including training data points (whose number is n) and testing data points (whose number is n_*).

When writing formulas, the probability density function is quite long, so I often use the notation $\mathcal{N}(a; \mu, \Sigma)$ to represent a Gaussian probability density function for random variable a with mean μ and covariance Σ.

The Gaussian Process prior (GP prior)

We call the above multivariate Gaussian distribution the **Gaussian Process prior (the GP prior)**. It represents our assumption of how the values of the underlying function that we are modeling are jointly distributed.

GP prior is a distribution over functions

People say the GP prior is a distribution over functions. To understand why we need to understand the following two points:

1. The random variable vector $[f(X), f(X\_*)]^T$ represents a function. This is where we need the mapping view to define functions.
2. The multivariate Gaussian distribution over these random variables gives different probabilities for different functions.

The GP prior is a multivariate Gaussian distribution over the random variable vector:

Note that the set X and X_* are fixed because in practice X is given as part of the training data, it is fixed. X_* contains the test locations; it also fixed — we know where we want to evaluate the function.

So the above random variable vector represents the function values at location X and X_*. From the mapping view to define functions, these random variables represent the following function.

Since X and X_* are fixed, this function can be represented just by the random variable vector $[f(X), f(X\_*)]^T$. Now it is easy to see that the GP prior describes a distribution over functions. Let's again write down the probability density function of the GP prior by assuming a zero mean function and use K to represent the covariance matrix to save space:

Let's write down two example functions over the same fixed $X=\{1\}$ and $X\_*=\{2\}$. The first function f_1 is:

The second function f_2 is:

If we plug the vector $[3, 4]^T$ for f_1 and $[5, 6]^T$ for f_2 into the above probability density function, we get two scalar numbers $prob_1$ and $prob_2$:

The following figure is an illustration of this probability density function. I plotted the location of function f_1 and function f_2 at the x-axis, and their corresponding probability number at the y-axis. The bell-shaped curve represents the full probability density.

Illustration of distribution over functions, with highlighted function f_1 and f_2

inputs, and returns probability numbers.

Note that:

1. In the above figure, the x-axis represents function value vectors. All these functions have the same domain (x-axis locations, 1 and 2 in our example).
2. This figure is just an illustration. In reality, it is not necessary that the vector for f_1 is on the left, and f_2 is on the right. And it is not necessary that this distribution over functions looks like a bell curve so much. And it is not necessary that $prob_1$ is larger than $prob_2$.

Sampling from the GP prior

Since the GP prior is a multivariate Gaussian distribution, we can sample from it. I generated 600 equally spaced values between 0 and 2π to form my sampling locations. So my GP prior is a 600-dimensional multivariate Gaussian distribution. I used a zero mean function and set the lengthscale $l=1$ and the signal variance $\sigma^2=1$. The following figure shows 50 samples drawn from this GP prior. I used this code to sample from the GP prior.

Please ignore the orange arrow for the moment. In the figure, each curve consists of 600 data points.

In the figure, I gave some curves a more opaque colour and some a more transparent colour to demonstrate that some functions are more likely, according to the prior, to be drawn than others. The more likely a function is, the more opaque colour I used to displayed its curve.

Why different curves have different probabilities? This is because each curve (which is made of 600 points) is a sample drawn from a 600-dimensional Gaussian distribution. The probability density function of this multivariate Gaussian distribution defines how likely a sample happens in a draw. If a function is closer to the mean of the prior distribution, the probability of it being sampled is higher.

From the above figure, we can also see that even if we set the mean function $m(x)$ to zero (so that at each x-axis location, the corresponding random variable has zero mean), the drawn samples (whose values are shown at y-axis) are not necessarily zero. Just like when you draw a sample from a univariate Gaussian distribution with zero mean, the sample may not equal to zero.

Just by pure luck, I noticed that within these 50 sampled functions, there is one, the orange curve with an orange arrow pointing to it. This curve is quite close to the sine function that we want to find. When luck strikes, let's take advantage of it to make an important point. This orange curve illustrates the essence of Bayesian learning: we design the prior distribution to contain a large set of candidate functions, then use training data to re-weight these candidates.

We can understand this "re-weighting" in the following sense:

1. We design the GP prior not by thinking about the training data, instead, by thinking about mathematical convenience of the distributions we use and what functions it can describe. In our context, we chose the Gaussian distribution because it has good properties, and we chose the squared exponential kernel to model smooth functions. This means the prior may not explain the training data well, as shown by the above figure that the prior has high probabilities for functions that do not even resemble a sine wave.

prior. The Bayes rule outputs the posterior distribution. The posterior is a distribution over the same set of functions as the prior, but it associates different probabilities to those functions. The posterior gives functions that are close to the training data set much higher probabilities than functions that are far away from the training data set.

You can see the effect of the Bayes rule by sampling functions from the posterior (later I will show how the posterior is computed) in the following figure. It plots 50 sampled functions from the posterior:

Sampled functions from the posterior

These sampled functions are very close to our training data points, which are marked by blue crosses. The high opaqueness of these curves indicates their high probabilities from the posterior distribution. This suggests that the posterior can explain the training data much better than the prior — through high probabilities, the posterior distinguishes out the functions that resemble the training data points. And you don't see curves are far away from the sine wave anymore because their possibilities in the posterior are so small that they don't get sampled easily.

infinite set of functions. The Bayes rule needs to go through these infinite number of functions and re-weight them. How can a process of going through infinite things terminate? Let's take a look at the Bayes rule:

We start from a candidate function *f(X)* with some probability from the prior, that's *p(f(X))*, which is probability number between 0 and 1. The Bayes rule scales this probability by a factor, written as a fraction. This scaling factor is proportional to the the likelihood of our data given the candidate function, that's the numerator, normalized by the average likelihood, averaged over all possible (and infinite number of) candidate functions, that's the denominator.

The integration is the part in the Bayes rule that goes through the infinite set of possible functions. We already understand that integration is an infinite sum computed in a finite step. That's why the computation of the Bayes rule terminates.

Note I should rename the name *f(X)* inside the integration to, say, *g(X)*:

This version of the Bayes rule is clearer in math. In math, one name can only mean one thing — 2 always means the number 2, it doesn't mean 3 here and 4 there. So in the above formula, *f(X)* is the function for which we want to calculate the posterior probability; *g(X)* is the integration variable, integration over all possible *n*-dimensional real vectors. They are two different things, so we give them different names.

However, in the first version of the Bayes rule, we use *f(X)* to mean both the function for which we want to calculate the posterior probability and the integration variable. This is mathematically confusing, but unfortunately it is how people usually write.

orange curve at the 600 fixed x-axis locations form a vector of 600 real numbers. This vector should evaluate to a higher probability in our posterior distribution (which by the way, is a multivariate Gaussian distribution as well, as you shall see a bit later).

Previously, we said that the Bayes rule gives higher probabilities to functions that are close to the training data. Let's now think how the posterior achieves this. It must give higher probabilities to real number vectors (representing function values) that are close to the training observations Y. So, the means of the posterior at training location X must not be zero; they should be closer to Y, right? Just like if you want a univariate Gaussian distribution to give high probability to a number, say 6, then this Gaussian distribution should have a mean closer to 6, and hopefully, the variance is not so wide.

For now, it is enough to intuitively understand that the Bayes rule takes the GP prior, consults the training data, and computes the posterior. Later in the *Computing the posterior* section, you will see how this understanding is reflected in math.

This essence of Bayesian learning suggests that when you design your prior:

1. You don't want your prior to be too broad, otherwise finding the subset can be harder.
2. You also don't want your prior to be too restrictive, as it may exclude the true function that we want to model.

This view of Bayesian learning as re-weighting things from the prior using data is the reason why we want to look at the samples from our prior — we want to make sure our designed prior permits the functions that we want to model. For example, by looking at another set of samples where I change the lengthscale l to 0.1:

Samples from the GP prior with lengthscale 0.1, and signal variance σ²=1

We immediately know that this prior is not good for modeling the sine function. Because the samples are two wiggly and do not look like the sine wave.

In practice, you draw samples from the prior to see if the look and feel of those sampled functions are similar to the actual training data you have. Samples serve as a sanity check. However, quite often you will find it is hard to decide if a sample is similar to your training data. In such cases, accept that sanity check using samples has its limitations, and move on.

The above two figures also reveal that the value of the lengthscale l is important. It dramatically affects the shape of the functions that the prior models.

Compared to the effect of the lengthscale, the signal variance parameter σ^2 only scales the functions in the y-axis. For example, if σ^2 is *10* instead of 1, samples from the GP prior look like the following:

The look and feel of those functions are similar to the previous figure, only the values at y-axis are scaled to have a larger range.

The fixed and vary-able part of the GP prior

Our GP prior has the multivariate Gaussian structure, and it has the lengthscale l and the signal variance σ^2 as parameters. We fixed the multivariate Gaussian structure by specifying its mean $m(x)$ and covariance $k(x, x)$ components. But we left the concrete values for l *and* σ^2 unspecified.

Both the multivariate Gaussian structure and the model parameter values contribute to defining the set of functions that our prior includes. In the parameter learning process, we vary the model parameter values to see which values result in a posterior that best explains the training data. But during parameter learning, we keep the Gaussian structure of the prior unchanged. This is because if we change the structure of the prior, we won't be talking about a Gaussian Process model anymore, the model may be something quite different, with very different properties.

Likelihood and marginal likelihood

The GP prior only mentions X. It is the likelihood that connects the random variables from the GP prior to the actual observations Y. From the prior and the likelihood, we can compute the posterior. And from the posterior, we make predictions.

Likelihood

The likelihood is the probability of observing Y given the random variables $f(X)$ and $f(X\_*)$ from the prior. To talk about the probability of observing Y, we need to introduce a new set of random variables. For each location x in X, introduce a new random variable $y(x)$. We assume our observation at location x is a sample of this random variable $y(x)$. There are n observations in Y, so we have a random variable vector $y(X)$ of length n.

Gaussian with mean $f(X)$ and covariance $\eta^2 I_n$, where η^2 is a scalar model parameter, called noise variance, and I_n is the identity matrix with size $n \times n$. In formula:

I use the notation $\mathcal{N}(y(X); f(X), \eta^2 I_n)$ to denote the multivariate Gaussian probability density function for the random variable $y(X)$, and this probability density has mean $f(X)$ and covariance matrix $\eta^2 I_n$.

An equivalent way to describe the same random variable $y(X)$ is to use the Gaussian linear transformation form:

This formula expresses $y(X)$ as a linear transformation from the random variable $f(X)$ with added Gaussian noise ε. The transformation matrix is none other than the $n \times n$ identity matrix I_n. Again, these two ways of defining $y(X)$ are equivalent.

The formula of the likelihood is the probability density function for the random variable $y(X)$ given $f(X)$:

The likelihood is a function of three arguments: $y(X)$, $f(X)$, and the model parameter $\{l, \sigma^2, \eta^2\}$, and we treat the whole set of model parameters as a single argument.

Why do we model $y(X)$ this way? Because:

1. We used $f(X)$ to model the possible function values at location X, it is natural to use it as the mean of our observational random variable $y(X)$.
2. We want to model that our observation Y is coming from $f(X)$ but somehow corrupted by random noise. So we introduced the noise variance $\eta^2 I_n$. The corruption at each location x is independent of each other, that's why $\eta^2 I_n$ is a

You may wonder, $f(X)$ is already a random variable with its own variance, defined by $k(X, X)$, why do we want to introduce another level of variance $\eta^2 I_n$?

The answer is: you don't have to. It is YOUR modeling. You can model it however you want. You can model observations Y to be samples from $f(X)$ without introducing $y(X)$. This is called a noise-free Gaussian Process (here, page 15).

But it is more common to introduce observation noise. Because then we have a better separation of concerns: $f(X)$ and its variance focuses on modeling the underlying system dynamics and its inherent uncertainty. And $y(X)$ and its variance focus on the uncertainty that occurs during observation. This is a closer match to real systems.

You may also wonder: why the noise needs to be independent at each location in X? Because if the noise is not independent, for example, if the noise gets bigger as locations in X gets bigger in a linear fashion, then it is not noise anymore. It is a characteristic of the system and we should model it in the prior.

You may also wonder, isn't likelihood usually the probability density of the observation random variable given **all** the random variables introduced in the prior? In other words, you would suggest the likelihood to be:

And why I defined the likelihood as $p(y(X)|f(X))$ without the random variable $f(X\_*)$? The answer is: you are right. The likelihood should be the formula that you suggested. But your formula is the same as mine. This is because I modeled the observational random variables as $y(X) = I_n f(X) + \varepsilon$, so $y(X)$ only depends on $f(X)$, and not on $f(X\_*)$. And I would argue this modeling makes sense because it would be strange to model $y(X)$ to depend on random variable $f(X\_*)$ at test locations X_*.

Since $y(X)$ does not depend on $f(X\_*)$, we can remove this irrelevant mention of $f(X\_*)$ in the condition, so the likelihood $p(y(X)|f(X), f(X\_*))$ becomes $p(y(X)|f(X))$. If you want to know more about how to remove a random variable from a probability density,

According to our model, the actual observation Y is a sample from the random variable $y(X)$. So $f(X)$ and $f(X\_*)$ become random variables whose samples that we will never observe. In probability theory, we call $f(X)$ and $f(X\_*)$ *latent random variables*.

Marginal likelihood

We already mentioned that the observation random variable $y(X)$ can be defined in the Gaussian linear transformation way:

From this formula, we can apply the rule of Gaussian linear transformation to derive the probability density function for $y(X)$ without mentioning $f(X)$.

First, we can get the distribution of the random variable $f(X)$ from the GP prior by applying the multivariate Gaussian marginalization rule:

We've used this rule once before to get rid of anything related to X_o, such as $f(X_o)$ and $y(X_o)$ in the section *The kernel function k*. Here we use the same rule to get rid of $f(X\_*)$ from the prior to get the distribution for only $f(X)$. So:

Then we can apply the Gaussian linear transformation rule (see my article Demystifying Tensorflow Time Series: Local Linear Trend for details of this rule) to get the distribution for the random variable $y(X)$. Here is the multivariate Gaussian linear transformation rule:

If the multivariate random variable a is from the following Gaussian distribution:

And random variable b is a linear transformation from a with A being the transformation matrix and with added Gaussian noise η:

Then the distribution of b is:

The distribution of b uses quantities from the distribution of a but does not mention the random variable a. This is an important point.

The multivariate Gaussian linear transformation is definitely worth your time to remember, it will pop up in many, many places in machine learning. For example, you need it to understand the Kalman filter algorithm, you also need it to reason about uncertainty in least squares linear regression.

For us, a is $f(X)$, b is $y(X)$, A is I_n and η is ε. We apply the linear transformation rule to derive the distribution of $y(X)$:

Its probability density function is:

This probability density function is the famous **marginal likelihood**. Unlike the probability density function of the likelihood, the above marginal likelihood density

you have a young kid at home, this kind of question occurs naturally)?

Since we marginalized the random variable *f(X)* out from the likelihood formula, we call the result marginal likelihood.

The marginal likelihood probability density function has two arguments, *y(X)* and the model parameters set. Knowing which unknowns/arguments that a function is important. Because later we will use the marginal likelihood formula as the objective function for parameter learning. So we need to make sure that the marginal likelihood is a function only with the model parameters as arguments.

Although the marginal likelihood has two arguments, *y(X)* and the model parameter set, we have observations *Y* for *y(X)*. We can plug-in *Y* into the position of *y(X)* to get a function with only one argument — the model parameter set.

You may wonder, don't we usually derive the marginal likelihood by integrating the latent random variable *f(X)* out, like this:

You are right, and if you compute this integration (I will provide another article to carry out this computation), the result is the same as the above when we applied the Gaussian linear transformation rule.

The advantage of applying the Gaussian linear transformation rule is that it is so much simpler. The downside is that we can only use the Gaussian linear transformation rule when everything is Gaussian. The above derivation of using integration, however, is

Computing the posterior

In a Bayesian method, we need to compute the posterior from the prior and the likelihood. The posterior is over the same set of random variables introduced in the prior.

Our GP prior consists of two latent random variable vectors $f(X)$ and $f(X\_*)$, but we are only interested in finding the posterior $p(f(X\_*)|y(X))$ because we need it to make predictions at test locations X_*. We can optionally, and trivially compute the posterior for $p(f(X)|y(X))$, by first computing $p(f(X\_*)|y(X))$, and then plugging X into X_*.

To compute the posterior $p(f(X\_*)|y(X))$, we rely on the fact that both $f(X\_*)$ and y(X) are multivariate Gaussian random variables, and we know the distribution for both.

The distribution of $f(X\_*)$ is derived by applying the multivariate Gaussian marginalization rule to the GP prior to get rid of things related to $f(X)$.

This is the third time we apply the multivariate Gaussian marginalization rule:

And in the *Likelihood and marginal likelihood* section, we have derived the distribution for $y(X)$, that's the marginal likelihood:

Since both $f(X\_*)$ and $y(X)$ are Gaussian, we can write down their joint distribution (the reason why we want to write down the joint distribution will be clear a few paragraphs down) as follows, provided we can compute the covariance between $f(X\_*)$ and $y(X)$, denoted by $Cov(f(X\_*), y(X))$:

Computing $Cov(f(X\_*), y(X))$ is simple if we remember the definition of covariance:

Line (2) shows the definition of the covariance between $f(X\_*)$ and $y(X)$, given their mean $m(X\_*)$ and $m(X)$.

Line (3) plugs in the definition of $y(X) = f(X) + \varepsilon$. Here we ignored the identity matrix I_n in front of $f(X)$ as I_n does not change the vector $f(X)$.

Line (4) re-organizes terms so we can split the single expectation into two expectations at line (5) by using the linearity property of the expectation: the expectation of a sum is equal to the sum of the two expectations of the operands.

Line (6) recognizes that the first expectation is the definition of the covariance between $f(X\_*)$ and $f(X)$. And line (7) plugs in this known quantity $k(X\_*, X)$.

Line (8) recognizes that the expectation between any random variable and ε is 0 because ε is independent of all other random variables.

So now we can have a fully specified joint distribution between $f(X\_*)$ and $y(X)$:

Now the posterior $p(f(X\_*)|y(X))$ is just one step away — we apply the multivariate Gaussian conditional rule to compute it. The reason why we want to write down the joint distribution between $f(X\_*)$ and $y(X)$ is that we want to apply this rule.

The conditional rule for multivariate Gaussian is:

The rule gives you the formula of $p(x|y)$ from the joint $p(x, y)$ when $p(x, y)$ is a multivariate Gaussian. Let's match terms in this rule to the terms in our joint distribution between $f(X\_*)$ and $y(X)$:

After applying this rule, we have the distribution for the posterior:

with the posterior mean and posterior covariance:

on.

But before I forget, I need to point out that these steps of:

1. Organizing our Gaussian latent random variables and non-latent observation random variables into a joint Gaussian distribution;
2. Applying the multivariate Gaussian conditional rule to derive the posterior probability density.

These two steps are a recurring pattern in Bayesian learning. Here we use it to compute the posterior for our Gaussian process model. Another famous example is the Kalman filter algorithm. See my other article Demystifying Tensorflow Time Series: Local Linear Trend, the *Kalman filter* section, and especially the summary of the Kalman filter algorithm.

You may wonder, why don't we use the usual Bayes rule to compute the posterior:

We can, and I will provide an article about it. The resulting posterior is the same as here, but the derivations are way more complicated than what we have here. The reason is that when everything is Gaussian, using the properties and rules from multivariate Gaussian is a shortcut to the Bayes rule.

Now let's investigate the structure of the posterior mean and posterior covariance.

Structure of the posterior mean and covariance

each component matrix in red (remember n is the length of $f(X)$, or the number of training data points. n_* is the length of $f(X\_*)$, or the number of testing locations):

The posterior mean and covariance are *symbolic* expressions — they are mathematical expressions that mention the model parameters $\{l, \sigma^2, \eta^2\}$. We don't know the values for those model parameters. So even after we plug in the known quantities: training location X, testing location X_* and observation Y into $y(X)$, we still have a symbolic expression. Only after we used parameter learning to find optimal values for our model parameters, we can evaluate these symbolic expressions into concrete real values.

Let's begin with the posterior mean:

To ease our investigation, we assume the mean function m is 0, so $m(X)=0$, and $m(X\_*)=0$. Now let me expand this formula. In fact, quite often you can help yourself understand a linear algebra expression by expanding the vectors and matrices.

Line (1) is the formula for μ_* that we derived before.

Line (2) replaces $m(X\_*)$ and $m(X)$ by 0 to ease our investigation.

Line (3) plugs part of our observation data Y into $y(X)$.

Line (4) expands the definition of $k(X\_*, X)$.

Line (5) expands Y into the actual vector.

The formula at line (5) reveals crucial information:

1. the mean value at a single test location, say x_*1, is a weighted sum of all the observations Y. The weights are defined by the kernel between the test location x_*1 and all training locations in X. Even with a zero mean function from the prior, the posterior mean is not necessarily zero. This formula shows what we mentioned before — the posterior re-weights functions from the prior such that they agree more with the training data. And this re-weighting results in a non-zero-meaned posterior.
2. The part in the middle $(k(X, X) + \eta^2 I_n)^{-1}$ does not mention x_*1, so it is a constant with respect to x_*1. You can imagine that the power $^{-1}$ puts the term $k(X, X) + \eta^2 I_n$ into the denominator of a fraction, where the numerator is $k(x\_*1, X)$ multiplied by Y. Then you can understand that the posterior mean is indeed a weighted sum of observation Y, with weights $k(x\_*1, X)$. And this weighted sum is normalized by $(k(X, X) + \eta^2 I_n)^{-1}$.

I implemented Gaussian Process in this code. And I used this code to plot the posterior mean and variance. The code for the GP is a straightforward translation of the above formulas into NumPy syntax. When reading it, pay attention to the use of assert

remember that. If you want to know more about writing assertions to help you develop machine learning code, this is a good read: Reducing Tensorflow Debugging Time by 90 Percent.

Let's see how the posterior mean behaves. The following is a very important figure. I used 50 equally spaced locations between 0 and 2π as our testing location X_* and computed the posterior mean and variance at those 50 locations. In this figure, the red dots represent the posterior mean at these test locations. The red curve connecting all the red dots is the matplotlib.plot effect. The blue bands represent the posterior variances, which are the values at the main diagonal of the posterior covariance matrix. I will explain the posterior covariance in the next section. The training data points are marked with blue cross "x" markers. I used lengthscale $l=0.4$, signal variance $\sigma=1$, and noise variance $\sigma=0.01$.

Posterior mean and variance for test locations. Blue crosses are training points, red dots are testing points.

We focus on the posterior mean first. The first impression is that the red curve connecting all the posterior means is kinda similar to the sine wave. Not perfect, but

To see the "**weighted** sum of Y" interpretation behind the posterior mean, in the following figure, I highlighted a single test location with a big red dot. And I draw the training data points with blue markers whose sizes are proportional to their weights in the posterior mean for the big red dot.

Now you can verify that the closer a training point to the highlighted test point, the larger the weights of the corresponding observations, so the blue markers are bigger. And the blue marker sizes decrease as training points get farther away from the highlighted testing point.

Mapping view to define functions revisited

After understanding that the posterior means is a weighted sum of observations Y, now we understand why the mapping view to define functions is important for Gaussian Process.

Gaussian Process does not find a function body that only needs a new x and returns a y in the traditional sense of $f(x)=ax+b$, like what linear regression gives you. Instead, it gives you a **mapping** between every test location x_* to a function mean value

an infinite number of test locations, in practice, you only need to make a finite number of predictions, so you only need to compute a finite number of such mappings.

We realize this difference is quite interesting: with linear regression, a so-called **parametric model**, after you used the training data to decide the values (this is called parameter learning) for parameter a and b in $f(x) = ax+b$, you can throw away the training data. Because making predictions on new x does not need the training data. But with Gaussian Process, a so-called **non-parametric** model, even after parameter learning, you still need to keep the training data, because the training data appears in the posterior mean and variance, and Gaussian Process relies on the posterior to make predictions.

You may wonder if Gaussian Process is space-efficient to make predictions. It is not. Previously we've already said you usually need more space to write down a function defined through input-output mappings. That was from the math point of view. Now let's see this space non-efficiency from the computer science point of view:

- In a parametric model, such as linear regression in form of $f(x) = ax+b$. After parameter learning to find the values for the parameters a and b, you only need the space to store these two numbers to make predictions for a new test location x_*. In other words, after parameter learning, you can throw away the training data.
- In a non-parametric model, such as Gaussian Process, after parameter learning to find the values for the model parameters, you still need to keep all the training data (X, Y) because both the model parameters and the training data appear in the posterior. So if (X, Y) consists of a million data points, get ready to allocate memory for them.

The posterior covariance

Now let's move to the posterior covariance matrix:

It is a matrix describing the covariance between every pair of random variables in the posterior of $f(X\_*) | y(X)$:

Note that entries in this matrix are not $k(X\_*i, X\_*j)$ because this is the posterior covariance, and $k(X\_*i, X\_*j)$ are used to define the prior. So I use the notation $Cov(X\_*i, X\_*j)$ for entries in the posterior covariance matrix.

Entries in the main diagonal are variances of each random variable in $f(X\_*)$ — covariance between a random variable and itself is called variance.

Let me show the figure with the posterior mean and variance again below.

Posterior mean and variance for test locations. Blue crosses are training points, red dots are testing points.

In this figure, the blue bands represent a 95% confidence interval. Remind ourselves that we generated non-equal-spaced training points. So some regions have more training points and some fewer. The posterior variance represents the model's

model is more certain about where the function value should appear at test locations. In regions where there is no training data nearby, the variance is large. In other words, the model is quite uncertain. This makes common sense. And this is why people say "training data explains the uncertainty away".

Now let's see how this common sense is reflected in math. But first, you may wonder, the posterior variance is a variance, it should be non-negative. Is the posterior mean formula always non-negative?

Yes, it is. And let's convince ourselves in a case where there is a single training data point and a single test data point. Since there is a single test point, there is only a single random variable in $f(X^*)$. The posterior covariance matrix becomes the posterior variance. We further assume there is no observation noise, in other words, $\eta^2=0$, so $\eta^2 I_n=0$. So the posterior variance becomes:

Line (1) is the posterior covariance formula with no observation noise, so there is no $\eta^2 I_n$ in the matrix inversion. Note that in this formula, all the k terms evaluates to a scalar number because we have a single training data point and a single test point.

Line (2) simplifies the formula because the transpose of a scalar is itself.

Line (4) expands and reduces $k(X\_^*, X\_^*)$.

Line (5) expands the definition of k.

Line (6) shows that the posterior covariance is non-negative because the input of the exponential function is a negative quantity, so the exponential function has a maximum value 1. So the whole formula evaluates to a non-negative value.

Training data explain away uncertainty

Also from the above formula, line (6), we can see why the posterior covariance becomes smaller when a testing point is closer to training points. As $X\_^*$ getting closer to X, the squared distance between the two approaches 0; the exponential evaluates to this maximum value 1. So the overall posterior variance evaluates to its minimum value.

On the other hand, when $X\_^*$ gets very far away from X, the exponential evaluates to 0, so the overall posterior variance evaluates to its maximum value σ^2. This is an interesting behavior, meaning that the posterior variance is bounded by σ^2, it won't go infinitely large — it will max-out eventually. You will see a picture showing this max-out behavior later.

Posterior re-weights functions from the prior

Let's revisit the view we developed in the *Sampling from the GP prior* section: the posterior gives higher probabilities to functions that are close to training data.

To validate this, we investigate when we set $X\_^*$ to X, what the posterior $p(f(X\_^*)|y(X))$ looks like. That is, we want to see what is the posterior distribution for functions at locations equals to X, the training locations. What we hope to see is that the posterior gives the function $X \longmapsto Y$ a high probability, where X and Y are our training data.

Posterior mean

Let's continue to use zero mean function and assume there is no observation noise, so $\eta^2 I_n = 0$. And we set $X\_^* = X$. The posterior mean formula becomes:

Line (1) is the original posterior mean formula.

Line (2) uses zero mean and plugs in training data: replace X_* with X and replaces $y(X)$ with Y.

Line (3) shows that the posterior mean is exactly the same as our observation data Y. Note here $k(X, X)$ and $k(X, X)^{-1}$ cancels because we assume $\eta^2 I_n = 0$. When $\eta^2 I_n \neq 0$, these two terms will not cancel each other, so the posterior mean will not be equal to Y.

Posterior covariance

The posterior covariance formula becomes:

Line (1) is the original posterior covariance formula.

Line (2) plugs in training data: replace X_* with X. Here, similar to the case in the posterior mean $k(X, X)$ and $k(X, X)^{-1}$ cancels because we assume $\eta^2 I_n = 0$. But they will not cancel each other if $\eta^2 I_n \neq 0$.

Line (3) simplifies the formula.

Line (4) shows that since $k(X, X)$ is a symmetric matrix, it equals to its transpose.

Line (5) simplifies the formula to show that the posterior covariance at training data locations X is 0.

density function, you will get the highest probability because a Gaussian probability density function has its maximum at the location of the mean. In fact, you cannot plug Y into this posterior distribution as with zero covariance, the posterior distribution is not a valid Gaussian distribution anymore. It is merely a mean vector which equals to Y and a zero covariance matrix, which you can interpret as selecting a single function that passes through all the training data points.

Making predictions

The posterior distribution $p(f(X\_*)|y(X))$ is for the latent variable $f(X\_*)$. But remember that we model the observations as a linear transformation from our latent variable with added Gaussian noise. So the posterior for the observation random variable $y(X\_*)|y(X)$ is:

where

Since we know the distribution for $f(X\_*)|y(X)$ from the previous section, that is the posterior, by applying the multivariate Gaussian linear transformation rule, we can derive the distribution for $y(X\_*)|y(X)$:

This is the distribution we need to predict function values at test locations X_*. Similar to the posterior mean and variance for $f(X\_*)|y(X)$, the posterior mean and variance for $y(X\_*)|y(X)$ are also functions with a single argument, the test locations X_*. The prediction is in the form of a Gaussian distribution with mean and variance. With these two quantities, you can plot the mean curve and the confidence interval.

random variables *f(X_*)*.

Gaussian Process models uncertainty in a principled way

The above formula for making predictions reveals a very important thing: a Gaussian Process model makes predictions in the form of a Gaussian distribution. Its mean tells us the average or expected value of a prediction, and its variance tells us the uncertainty that the model has for this prediction.

The uncertainty (measured by covariance matrix) of the prior and the uncertainty of the likelihood propagate into the uncertainty of the posterior through the multivariate Gaussian conditional rule (and equivalently the Bayes rule) we used to derive the posterior. This is why we say Gaussian Process models uncertainty in a principled way.

You may wonder, is this uncertainty subjective? Meaning we subjectively decided to use multivariate Gaussian distributions for both the GP prior and the likelihood, and the uncertainty comes from the uncertainty of these two distributions. If we were to choose a different distribution (say, we choose the student-t distribution and devised a "student-t process" model), we may have different uncertainty characteristics. Yes, you are right. The uncertainty is subjective to the distribution we choose. And since we don't observe the uncertainty directly, it is always going to be a subjective choice, balancing between model expressiveness and ease of computation.

We've been through quite some material, the only thing left is that we need to find good values for the three parameters that we introduced during modeling, the lengthscale l, the signal variance σ^2 and the noise variance η^2. The process of finding good values for them is called *parameter learning*. But before we start describing parameter learning, I would like to summarise the Gaussian Process model for you.

Gaussian Process model summary and model parameters

Gaussian Process model

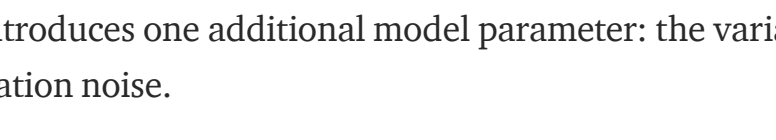
We call the GP prior together with the likelihood the Gaussian Process model. In fact, all Bayesian models consist of these two parts, the prior and the likelihood.

The prior is a joint Gaussian distribution between two random variable vectors *f(X)* and *f(X_*)*. They respectively represent the possible function values at training locations X and testing locations *X_**. The prior does not mention observations *Y*. Its formula is:

where m is the usually a zero function and k is the kernel function:

The prior introduced two model parameters: the lengthscale l and the signal variance σ^2.

The likelihood is a multivariate Gaussian distribution connecting our observation random variable $y(X)$ to the latent random variable $f(X)$:



The likelihood introduces one additional model parameter: the variance η^2 of the Gaussian observation noise.

That's it, a simple model. This model introduces three random variables (each is a random variable vector):

`f(X)`: latent random variables that represent possible values at training locations `X`.

`f(X_*)`: latent random variables that represent possible values at testing locations `X_*`.

`y(X)`: random variables that represent possible values for observations at training location `X`. We model our actual observations `Y` as a sample from `y(X)`.

Note: the above three are random variables, so when you see the expression `f(X)`, `f(X_*)`, and `y(X)`, don't interpret them as function applications. They are random variable names.

Note: the expression `m(X)` is a function application. It represents the mean function we use in the prior. It is a function application, and not the name of a random variable.

function; we need to decide on the squared exponential kernel, and we need to decide that our observation random variable is a linear transformation from our latent random variable. Once we've decided on these parts, computing the posterior and making predictions are all mechanical, following the rules from probability theory and Gaussian distribution.

Now I want to talk more about those three model parameters.

Model parameters

In total, the Gaussian Process model introduces three model parameters. Let's discuss their intuitions.

Lengthscale l

As we've already seen from the *Sampling from the GP prior* section, lengthscale dramatically changes the look of the functions our GP prior can model. Let's see how the lengthscale does this by looking at its formula:

We can see that the inverse of the lengthscale controls how far away, in the sense of $(x - x')^2$, two random variables need to be so they become uncorrelated. It is the inverse of the lengthscale because l appears in the denominator inside the exponential function. The following chart shows for fixed signal variance $\sigma^2=1$, how two kernel functions look like. One has lengthscale $l=0.01\pi$ in red, and the other lengthscale $l=0.5\pi$ in blue.

Kernel function values with different lengthscales, l=0.5π in blue, and l=0.01π in red.

In this figure, the x-axis is the distance between x and x'. Let's look at the distance $(x-x')^2=0.5\pi\approx1.57$. That is the blue dot and the red dot in the figure. When l is 0.5π, two random variables at 0.5π distance apart are still quite correlated, with covariance about 0.6. On the contrary, when l is 0.01π, two random variables at 0.5π distance apart are almost uncorrelated, with a covariance value of about 0.

When trying to understand the effect of changing parameters to a function, I find it useful to analyze the extreme cases. In our kernel example:

1. When lengthscale l approaches 0, the kernel value approaches 0; our model believes that any two different random variables are not correlated, no matter how close they are.
2. When lengthscale l approaches ∞, the kernel value approaches 1; our model believes that any two random variables are correlated 100%, no matter how far away they are.

How does this change in random variable covariance affect the look and feel of the functions that our GP prior can model? Let's see two samples, one from each prior. The red sample is from the prior with the lengthscale 0.01π, the blue sample with lengthscale 0.5π:

Samples from the GP prior with different lengthscales

In this figure, the data locations are deliberately chosen to be evenly 0.04π apart so the distance between two adjacent data points is larger than lengthscale 0.01π but smaller than lengthscale 0.5π.

Let's first look at the blue curve with $l=0.5\pi$. $l=0.5\pi$ means that random variables (i.e., the blue dots) within a 0.5π window should be quite correlated. I highlighted such a window using the green rectangle. So a smooth curve, like the blue, is *consistent* with these covariance characteristics.

What does "consistent" mean? Remember that all the random variables:

1. have the same mean 0 because we use a 0 mean function. Having the same mean suggests that on average, samples from those random variables should appear at the same y-axis level.
2. have the same variance, because their variances are defined in the main diagonal of the covariance matrix. They all have variance σ^2. Random variables having the same variance suggests that samples from them should vary in the same horizontal range.
3. the random variables are positively correlated. This suggests that if a sample from a random variable $f(x_1)$ is larger than the mean 0, samples of another random variable within the $l=0.5\pi$ lengthscale distance $f(x_2)$ are more likely to also be larger than 0. This is the behavior we expect to see from positively correlated random variables — they vary in the same direction.

The blue dots, which are samples of the random variables (one sample per random variable) from the prior with lengthscale $l=0.5\pi$, in the highlighted green window, match these characteristics.

the y-axis in different ranges. In fact, for the red dots, their lengthscale is $l=0.01\pi$, which is smaller than the distance 0.04π between each two data points. So any two red data points are coming from random variables that are effectively uncorrelated. That's why the red curve connecting the red points jumps up and down.

Signal variance σ^2

The model parameter is σ, but usually, we write it as σ^2 to emphasize that it is a variance. As we already saw in the *Sampling from the GP prior* section, different σ^2 values scale the functions that our GP prior can model in the y-axis dimension. This is the intuition for this model parameter — it reflects the range of the function that you want your GP prior to be able to handle.

Variance η^2 of the observation noise

Same as signal variance, we write η^2 to emphasize that σ defines a variance. We have η^2 because we assume that our observations Y are samples from the random variable $y(X)$ with added Gaussian noise. That Gaussian noise has 0 mean, and as η^2 its variance.

We don't know the values for these three model parameters. We use parameter learning to find good values for them.

Parameter learning

Good values for the model parameters help our model explain the training data better. Parameter learning is a process to tweak those model parameter values until they can explain the training data good enough.

In Gaussian Process, this tweaking of model parameter values is done via optimization. We first come up with an objective function that quantifies how well our model explains the training data. This objective function takes and only takes *all* the model parameters as arguments and returns a real number. We interpret a larger number from the objective function as the model explains the training data better. Parameter learning tries to find concrete model parameter values to maximize the objective function.

Choosing an objective function

training data X and Y.

In our Gaussian Process model, we have two formulas that mention the full training data X and Y. The likelihood $p(y(X)|f(X))$ and the marginal likelihood $p(y(X))$.

Likelihood

When we plugged in Y into the above formula we get a function with two arguments, the model parameter set and the latent random variable $f(X)$, shown below. This function has the model parameters as an argument because it mentions the observation noise variance η^2 and it mentions the lengthscale l and the signal variance σ^2 through K. It has argument $f(X)$ because $f(X)$ is mentioned in the exponential function. Everything else is constant.

Different model parameter values will result in different probabilities of observing our observation data Y. We can interpret a high probability as "explaining the training data better". So the likelihood function may be a candidate for the objective function.

The problem with the likelihood function is that besides mentioning all the model parameters, it also mentions the random variable $f(X)$. It violates our requirement that an objective function should be a function that mentions only the model parameters and no other unknowns. We have this requirement because the optimization algorithm — gradient descent — only works on functions with scalars unknowns, not on functions with random variables as unknowns.

Marginal likelihood

So let's look at the marginal likelihood $p(y(X))$:

This is a function with a single argument, the model parameter set. This is because (1) even though $y(X)$ is a random variable, we can plug in Y to remove it. (2) $m(X)$ is not a random variable, it is a quantity that we can compute given X. And we usually set $m(X)=0$. (3) We know X. (4) all the model parameters are mentioned. Everything else is constant.

We can still interpret a high marginal likelihood value as our model being able to explain the training data better. After all, the meaning of marginal likelihood is the expected likelihood $p(y(X)|f(X))$ with respect to the random variable $f(X)$ coming from the prior: $f(X) \sim \mathcal{N}(0, k(X, X))$:

Note that even though we used the Gaussian linear transformation rule to derive the formula for the marginal likelihood, instead of the above integration, the transformation rule is just a shortcut to the above integration.

We can understand that the marginal likelihood is a measure of how well our model explains the training data in the following sense:

1. How well our model explains the observation data Y is measured by the likelihood value when we plug Y in — $p(y(X)=Y|f(X))$. This quantity depends on $f(X)$ because its formula, which is a multivariate Gaussian probability density function, mentions $f(X)$.

2. $f(X)$ is a random variable from the prior $p(f(X))$. Different samples of $f(X)$ result in different likelihoods of our training data $p(y(X)=Y|f(X))$. To quantify the overall likelihood of Y using a single number, we look at the average of all these likelihoods, which is the marginal likelihood.

The good news is that the marginal likelihood formula satisfies our requirement for an objective function. So we choose it as our objective function.

observation data *Y* can be generated by our model with a maximal likelihood.

Since the marginal likelihood is an exponential function, we can take the log of it to cancel the exponential. The *log* function is strictly increasing, so maximizing *log p(y(X))* results in the same optimal model parameter values as maximizing *p(y(X))*.

Understanding the objective function

The objective function embeds our definition of model quality into a single number. It's important to understand how this function behaves. Let's write down its formula:

There are three terms in the objective function. I highlighted them in red boxes— the model complexity term, the data fit term, and the constant term. We can ignore the constant term because it will not change the result of the optimization. We will carefully study the data fit term and the model complexity term.

Please note that the data fit term and the model complexity term exists in the objective function because we decided to use the log marginal likelihood formula as our objective function. These terms are the components in the log marginal likelihood formula. We did not design the components in these two terms and we did not decide to add them together.

Please also note the data fit term and the model complexity term includes the minus sign "-" at the front. And we want to maximize the objective function. Forgetting these two minus signs will make you think all the analysis below should be in the opposite direction.

Space-efficiency revisited

million data points, $k(X, X)$ is of size million by million and takes a lot of memory to hold. Compare this to batch learning in linear regression, where you only need to hold a small fraction — a mini-batch — of training data in memory, the Gaussian Process way is not space-efficient.

Worse, we need to perform matrix inversion on this matrix in the data fit term. Matrix inversion is a slow operation and it can have numerical stability issues. So it is not space-efficient nor time-efficient for Gaussian Process to do parameter learning.

If you look at my code that implements Gaussian Process, I used the inverse operation from NumPy to invert $k(X, X)$. This only works when you have a small training data set. In practice, you will use smarter methods, such as Cholesky decomposition, to invert $k(X, X)$, taking advantage that it is a definite positive symmetric matrix. More on this in future articles.

You may wonder, what about the space and time complexity of the determinant operator in the model complexity term? Compared to inverting $k(X, X)$, the determinant takes much less time. The inversion of $k(X, X)$ is the dominant factor in the space and time analysis for Gaussian Process parameter learning.

The data fit term

Let's think why $-\frac{1}{2}(y(X)-m(X))^T(k(X, X)+\eta^2 I_n)^{-1}(y(X)-m(X))$ is called the data fit term. I think "data term" is a better name than the data fit term. I want to call it the data term because, in the objective function, this term is the only term that mentions the observation Y through $y(X)$. I dislike the name "data fit term" because when we hear the phrase "data fit", we tend to think about a term that measures the distance between the model-predicted value at training locations X and the actual observation Y at those locations. This is an important point, let me clarify.

For example, in linear regression, the data fit term is the Euclidean distance between the model prediction $aX+b$ and the observation Y: $(aX+b-Y)(aX+b-Y)^T$. But in the Gaussian Process objective function, the so-called "data fit term" $-\frac{1}{2}(y(X)-m(X))^T(k(X, X)+\eta^2 I_n)^{-1}(y(X)-m(X))$ does not represent such a distance between model prediction and observation.

For one thing, in Gaussian Process, the prediction is in the form of the posterior distribution with mean and covariance. It is not clear how to calculate Euclidean

For another thing, even if we ignore the covariance and only look at the posterior mean (when we set $m(X)=0$) at location X, which is $k(X, X)(k(X, X) + \eta^2 I_n)^{-1}Y$. The Euclidean distance between it and the observation Y is:

We can see minimizing this quantity is different from minimizing the data fit term without the -½ at the front: $Y^T (k(X, X)+\eta^2 I_n)^{-1}Y$.

In Gaussian Process, the data fit term is just one term that appears in the log marginal likelihood. That's it.

But you may wonder, there must be some mechanism in Gaussian Process to make sure the model predicted values at location X are close to the observations Y, right? The answer is the Bayes rule, which gives us the posterior. The Bayes rule uses the observation data Y to update the prior distribution into the posterior distribution, such that the posterior can generate Y with a probability higher than the prior does.

You may ask, if the Bayes rule already gives us the posterior distribution, which by definition, should explain the observation data Y better than the prior does, why do we still need to maximize our objective function, the log marginal likelihood? This is because:

First, the probability density function for the prior $p(f(X))$ and the posterior $p(f(X)|y(X))$ are functions of the model parameters. They are symbolic expressions. Only when we decide on some concrete values for those model parameters, the posterior probability density function become evaluate-able. So we need a way to find concrete values for model parameters.

Second, not all choices of concrete model parameter values are equally good. Even though the structure of the prior, the likelihood and the posterior stays the same, plugging in different model parameter values will give you different prior, likelihood and posterior probability densities. Different posteriors means that they explain the same observation data Y with different probabilities. As you will see in the *overfitting* and *underfitting* sections, bad choices of those model parameter values result in models that won't explain our observation data Y well. Obviously, we want the posterior that

How does the data fit term change

How does the data fit term change when we change the lengthscale l? The trend is easily demonstrated if we set the observation noise $\eta^2 I_n$ to 0, and set the mean function $m(X)$ to 0, and set the signal variance σ^2 to 1. We further assume there are only two training points (X_1, Y_1) and (X_2, Y_2). In this setting, the data fit term becomes:

Now, let's vary the lengthscale l and see how the data fit term changes. When l approaches 0, the data fit term evaluates to the following limit:

When l approaches ∞, the data fit term evaluates to the following limit:

Now we see that when we increase the lengthscale l from 0 to ∞, the data fit term decreases from $-½(Y_1+Y_2)^2$ to -∞.

This trend of decreasing data fit term is still true when we have different values for the signal variance and the noise variance. But the decrease is not monotonic.

The above derivation assumes $Y_1 \neq Y_2$, meaning not all the training Y are equal. I will analyze the case when $Y_1 = Y_2$ at the end of this section.

The model complexity term

Let's think why $-½log(det(k(X, X)+\eta^2 I_n))$ is called the model complexity term. $det(k(X, X)+\eta^2 I_n)$ means the determinant of the matrix $k(X, X)+\eta^2 I_n$. Similar to the data fit term, let's focus on the lengthscale l.

GP prior to model more wiggly functions. Here are two sampled functions from two GP priors, the blue from a prior with larger lengthscale $l=0.5\pi$, and the red from a prior with smaller lengthscale $l=0.1\pi$.

Samples from the GP prior with different lengthscales

The blue curve is smoother and the red more wiggly. A smoother curve reminds us of the linear regression setting of using a lower degree of a polynomial, in other words, a simpler model. A more wiggly curve reminds us in the linear regression setting with a higher degree of a polynomial, in other words, a more complex model.

In case you don't have a mental picture about low and high degree polynomial linear regression, I used this website to try out linear regression with different degrees of a polynomial on our training data. Here is the fit for degree 3:

Polynomial linear regression with degree 3

And here is the fit for degree 49:

Polynomial linear regression with degree 49

We can see that in the linear regression setting, a lower degree fit, or alternatively, a simpler model, gives a smoother fit curve. A higher degree fit, or alternatively, a more complex model, gives a more wiggly fit curve.

In Gaussian Process, we adopt the same notion of model complexity. We call a GP prior that permits smoother functions simpler, and a model that permits more wiggly functions more complex.

on the observation Y. The term, $-\frac{1}{2}\log(\det(k(X, X) + \eta^2 I_n))$, by not mentioning Y, seems to be a good candidate to measure the complexity level. We now study how the value of this term changes when the lengthscale l changes from 0 to ∞.

Similar to the study of the data fit term, let's assume $\eta^2 I_n$ is 0, so the model complexity term is $-\frac{1}{2}\log|K|$, and we continue to use only two training locations X_1 and X_2.

When the lengthscale l approaches 0, the model complexity term becomes:

At line (3), the variance terms $k(X_1, X_1)$ and $k(X_2, X_2)$ evaluates to 1. And the covariance terms $k(X_1, X_2)$ and $k(X_2, X_1)$ evaluates to 0 when l approaches 0. So the $k(X, X)$ becomes an identity matrix.

At line (4) the determinant of an identity matrix evaluates to 1.

At line (5) log(1) evaluates to 0.

So when the lengthscale l approaches 0, the model complexity term approaches 0.

When the lengthscale l approaches ∞, the model complexity term becomes:

At line (3), the covariance terms $k(X_1, X_2)$ and $k(X_2, X_1)$ evaluates to 1 when l approaches ∞. So the $k(X, X)$ becomes a square matrix of all ones.

At line (4), the determinant of a matrix of all ones evaluates to 0.

At line (5), log (0) evaluates to $-\infty$.

So the model complexity term evaluates to ∞ when the lengthscale l approaches ∞.

The intuition of this change is clear from the meaning of the determinant operator. The determinant of a matrix measures the volume of the space enclosed by the column vectors of that matrix. For example, if a matrix has three columns $[a\ b\ c]$, then its determinant is the volume enclosed by a, b and c, shown below (image adapted from here):

1. When the lengthscale is 0, the matrix $k(X, X)$ is an identity matrix. The column vectors in an identity matrix are perpendicular to each other. So the space volume that they enclose is maximum, 1.

2. When the lengthscale approaches ∞, $k(X, X)$ becomes more and more a matrix with all ones, so all the columns are 1 columns. These columns point to the same direction. So the space volume that they enclose is minimum, 0.

So now we can see that the model complexity term does give us a good measure of the model complexity. When a model is complex, the model complexity term evaluates to a small value 0. When a model is simple, the term evaluates to a large value ∞. Since we want to maximize the objective function, we would like the model complexity term to be larger. In other words, this term conveys the idea that we prefer a simpler model.

Overfitting

Now we have our measure for model complexity. And we hinted that an unnecessarily complex model is bad. But why? Because although a complex model can fit the training data very well in the data fit term sense, it has no ability in making good predictions for new testing locations. This is called overfitting. Here is how to see the effect of overfitting:

1. As we've already seen from the analysis of the data fit term, a complex model with a very small lengthscale will give you the best data fit in the sense that the data fit term evaluates to its maximum value.

2. However, since the lengthscale is very small, the kernel function evaluates to 0 for any two locations that are not equal. Assuming the test locations X_* is different from the training locations X, then the random variables in $f(X\_*)$ will have zero correlation with the random variables in $f(X)$.

3. As we already mentioned at the beginning of this article, this means that knowing the value Y at location X gives us no information about how the function values at test location X_* should be. In other words, our extremely complex model has no ability to make predictions at test locations because the random variables at training location $f(X)$ are independent of the random variables at testing locations $f(X\_*)$. The complex model will fail to generalize to testing data. This is overfitting.

and a single test point.

When the lengthscale approaches 0, the posterior mean is:

At line (2), since the lengthscale is very close to 0, $k(x, x')$ evaluates to 0 when $x \neq x'$. So we have a few 0 entries.

And finally, the posterior mean evaluates to 0 at line (4). The interpretation is: since the random variables in $f(X\_*)$ are independent of those in $f(X)$, the model does not know any information about $f(X\_*)$ beside the assumption that its mean is 0, defined in the GP prior. That's why the model gives 0 as the posterior mean.

Now, let's look at the posterior covariance:

Obviously, we don't want overfitting. But shall we go for the simplest model, with lengthscale approaching ∞? The answer is no because then we will have another problem — underfitting.

Underfitting

To reveal the problem of underfitting, let's use a large lengthscale say $l=100$ to form a simple model. In this setting, $k(x, x')$ when $x \neq x'$ evaluates to a value that is very close to 1, say 0.999, no matter where x and x' are because the denominator inside the kernel's exponential is dominating. Let's again look at the ability of this model to make predictions by studying the posterior mean and covariance at a single test location X_*.

The posterior mean is:

The above formula means that no matter where X_* is, the posterior mean evaluates to equal-weighted sum of the training observation Y, which is a constant. This constant prediction is clearly a bad one.

To finish up, we look at the posterior covariance when the lengthscale is large:

We can see that when underfitting, the model is very confident about its predictions, unlike the case of overfitting, when the model is very uncertain.

Why in the case of underfitting, the model is very certain? This is because the lengthscale is very large, so the covariance between any two random variables is almost 1. This means according to our model, $f(X)$ will contain full information about $f(X\_*)$ — $f(X\_*)$ covaries with $f(X)$ one hundred percent. Once we know the value of $f(X)$, which is our observation Y (since we assume there is no observation noise), there is no room for the values of $f(X\_*)$ to vary. Hence the 0 covariance, and the absolute certainty.

As you've seen in both the overfitting and underfitting cases, bad choices of model parameter values result in models that won't explain the observation data Y well. The good news is by maximizing $log(p(y))$ with respect to the model parameters, we can find sensible concrete values for the model parameters that avoid both underfit and overfit. Here is why.

The battle: the data fit term vs. the model complexity term

Let's summarize the behavior of the data fit term and the model complexity term in the following table.

We can see that as the lengthscale l increases from 0 to ∞, the data fit term and the model complexity term changes in opposite directions — the data fit term decreases towards negative infinity while the model complexity term increases towards positive infinity.

The two terms balance each other to give us a model that fits the training data well and at the same time not too complex. This property of auto-balancing is desirable as we

Can we trivially solve parameter learning?

The answer is no. But let's first try to understand the question.

Looking at the above table, an interesting question may arise: since we want to maximize the objective function, and the only term that has the potential to go very very large is the model complexity term $-\frac{1}{2}\log(\det(k(X, X)+\eta^2 I_n))$ when the lengthscale approaches infinity. Why don't we trivially just set the lengthscale to a very large number? Of course, if we do that, the data fit term will decrease and drive the overall value of the objective function down. Which term will win?

To answer this question, we need to study which term approaches infinity faster: is the data fit term approaching $-\infty$ faster or the model complexity term approach ∞ faster. We use limits to study their speeds. We continue to use the setting where there are two training data points (X_1, Y_1) and (X_2, Y_2), the signal variance σ^2 is 1 and there is no observation noise, so $\eta^2 I_n = 0$.

To keep the formula a one-liner, I will introduce the symbol r (short for relevance) to represent the kernel value $k(X_1, X_2)$ for location X_1 and X_2. Since we only have two training locations, we have only one r.

So when the lengthscale l approaches infinity, which is the situation we are studying, r approaches 1, as the argument of the exponential function approaches 0.

With the new symbol r in place, the model complexity term becomes:

The data fit term becomes:

Now we study what is the limit of the objective function, with the constant term ignored, when the lengthscale l approaches ∞, or alternatively, when r approaches 1:

As the explanation at line(4) indicates, the above derivation is for the case when $Y_1 \neq Y_2$. In this case, as the length scale approaches ∞, or as the relevance r approaches 1, the whole objective function approaches $-\infty$. In other words, even though the model complexity term increase to ∞, the data fit term decreases to $-\infty$ faster. So the data fit term wins. So we cannot trivially set the lengthscale to a very large value and declare parameter learning is done.

Now we study the case when $Y_1 = Y_2$:

This is simpler. When all the observations in Y are equal, as the length scale approaches ∞, or the relevance r approaches 1, the object function approaches ∞. So in this case, the model complexity term wins. So in the unusual case when all observations are the same, we can trivially solve parameter learning by choosing a very large lengthscale.

Winning and losing aside, it is important to get the intuition why in the case when all observations Y are equal, a large lengthscale is better.

The intuition comes from the fact that the posterior mean at a test location is a weighted sum of all training observations Y. In current case, since the lengthscale is large, all observations Y have close to 1 correlation with the function value at the test location, so the weights associated

Since the lengthscale is large,

Since all the observations Y are the same, and the lengthscale is large,

it makes sense to predict values at test locations to be the same as the observed value. Remember that the posterior mean at a test location is a weighted sum of training observations Y. And there is no reason to give any particular observation more weight than others as they are the same. So the model gives them equal weights. Equal weights translate to a large lengthscale so $k(x, x')$ evaluates to 1 no matter where x and x' are.

Since in most cases, we cannot trivially solve parameter learning, let's solve it non-trivially.

Finding optimal model parameter values

To find the lengthscale value that maximizes the objective function, I line searched over its values from 0 to 3 at a step of 0.01. I used this code to perform parameter learning via line search.

As the lengthscale increases, the data fit term decreases, meaning we are getting a worse data fit; and the model complexity term increases, meaning we are getting a simpler model. This is consistent with our mathematical analysis from above.

When lengthscale $l=2.1$, the objective function has its maximum value. I highlighted the maximum objective function value with a big red dot.

Data fit term, model complexity term and objective function curves

Here is what the posterior mean and variance look like with the optimal lengthscale $l=2.1$:

Posterior mean and variance with optimal lengthscale l=2.1. Blue crosses are training points, red dots are testing points.

Again, the blue crosses "x" are training data points, and the red dots are testing data points. I have to say, this is very sine wave-like. And the 95% confidence interval is very small, we don't see it anymore.

Gradient descend to optimize the objective

The above grid search is just for illustration. In practice, we will use the gradient descent algorithm to find the optimal values for model parameters. The requirement of gradient descent is that the objective function is differentiable with respect to the model parameters. This is true for our objective function. Another thing I want to point out is that our objective function is not convex with respect to the model parameters. Gradient descent will find a local maxima.

With this, we finished parameter learning. Congratulations!

The incapability of the squared exponential kernel

Let's look back at our original task: we want to find a function f: $\mathbb{R} \longmapsto \mathbb{R}$ that explains our training data well. The posterior mean and variance of the Gaussian Process model does give us this function.

But, you may wonder, the function f has domain $\mathbb{R}$, but we've only looked at the domain from 0 to 2π. "Show me predictions beyond 2π", you demand! Here it is with domain from 0 to 6π:

Posterior mean and variance beyond 2π. Blue crosses are training points, red dots are testing points.

Wow, we can see that beyond 2π, the posterior mean starts to deviate from the "correct" values, and it gradually falls back to 0 starting from the location around 4.5π. At the same time, the posterior variance starts to increase until it finally maxes out at around the location 4.5π.

This gradual deterioration of prediction quality happens because when we reach the location around 4.5π, the correlation between the random variable at training locations and a test location is essentially 0. So the model can only report the posterior mean to be the mean 0. Also, the posterior variance, representing the uncertainty level, maxes out.

You may complain, after so much work, we have not found the correct regression function for the sine wave. This is because sine is a periodic function. How much two points in the sine function correlates to each other not only depends on how far they are apart but also depends on whether they appear in the same location of a cycle. The kernel function that we chose, which is the squared exponential kernel, is not capable of modeling this periodic effect.

adding or multiplying two kernels together. I will provide another article on kernels. For the moment, it is sufficient to understand that there are different kernels already defined for us. Different kernels model different kinds of functions. And choosing kernel is the most important task in using Gaussian Process.

Conclusion

This article introduced how to use Gaussian Process to perform a regression task. It covered the Gaussian Process regression model, how to make predictions using posterior mean and variance, and how to do parameter learning.

This article only covers the most basics that you need to master before you can march into more advanced topics. As I said at the beginning of this article, to use Gaussian Process in a real application setting, you may need two other techniques:

- Variational Gaussian Process — What To Do When Things Are Not Gaussian introduces variational inference to allow us to use a non-Gaussian likelihood in a Gaussian Process model.
- Sparse and Variational Gaussian Process — What To Do When Data is Large introduces inducing variables to allow us to scale a Gaussian Process model to large datasets.

Of course, the Gaussian Process field includes much much more. In the future, I will provide other articles for those advanced topics.

Support me please

If you like my story, I will be grateful if you consider supporting me by becoming a Medium member via this link: https://jasonweiyi.medium.com/membership.

I will keep writing these stories.

Acknowledgments

I would like to thank many of my Prowler.io colleagues for their help in writing this article. Especially, Nicolas Durrande, Stefanos Eleftheriadis, Vincent Adam, Vincent Dutordoir, ST John for their ideas and inspiration, and their kindness of sitting down and go through many of the formula derivations with me. I would also like to thank Sherif Akoush and Andrew Liubinas for their feedback on this article.