Optimal Gaussian Process Kernel Choice

Arjun Dhamrait

November 2024

Abstract

The objective of this project was to use Gradient Descent to choose the optimal kernel function for a Gaussian Process. When approximating a sinusoidal function, The optimized kernel function weights obtained from maximizing the log-likelihood of the Gaussian Process correctly weighed the periodic kernel function considerably more than the other kernel functions. This choice of kernel function additionally allowed the model to predict unseen data very accurately.

1 Introduction

1.1 Background

Gaussian Process Regression attempts to predict function values of a function that isn't known. It does this by using kernel functions that describe how correlated two different input training values are, then using those correlations it correlates test inputs to those same values using the same kernels and outputs the expected value of those test inputs. If the underlying function is linear, a linear kernel function would be best, r, if the underlying function were sinusoidal, a periodic kernel function would be best, etc. however what kernel function should be used when we don't actually know the underlying shape of our data? This project, I will choose the appropriate kernel function(s) that maximizes the log likelihood of the Gaussian process to output the training data, the test it on some test data and see how close it got!

1.2 A Simple Gaussian Process

To start off, assume I have been given n training points $\boldsymbol{x}$, their training values $\boldsymbol{y}$, and some test points $\boldsymbol{x^*}$:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) + \epsilon, \boldsymbol{x^*} = \begin{bmatrix} x_1^* \\ x_2^* \\ \vdots \\ x_m^* \end{bmatrix}$$

where ϵ is some random noise added to the function values and $\boldsymbol{f}$ is the function we want to learn using the gaussian process. For a simple Gaussian Process, one can compute a kernel matrix K where each of its values are computed from the kernel function:

$$K_{ij} = k(x_i, x_j)$$

A simple kernel function, the linear kernel function, looks like the following:

$$k(x_i, x_j) = \boldsymbol{\alpha}_1 x_i x_j + \boldsymbol{\alpha}_2$$

where $\boldsymbol{\alpha}_1$ and $\boldsymbol{\alpha}_2$ are the kernel function's hyperparameters. More kernel functions can be found here [1]. Importantly, kernel functions have the following properties:

1. $k(x_i, x_j) \geq 0$
2. $k(x_i, x_j) = k(x_j, x_i)$

From these properties, it's clear that $K \in S_+^n$.

1.3 Predicting using a Gaussian Process

A Gaussian Process assumes the training outputs $\boldsymbol{y}$ and test outputs $\boldsymbol{y}^*$ have a joint distribution whose prior is

$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{y^*} \end{bmatrix} \sim \mathcal{N}(\mathbf{0}, \begin{bmatrix} K + \sigma_n^2 I & K^* \\ K^{*T} & K^{**} \end{bmatrix})$$

where $K^*_{i,j} = k(\boldsymbol{x}_i^*, x_j)$ is the covariance matrix of the test points and the training points, and $K^{**}_{i,j} = k(\boldsymbol{x}_i^*, \boldsymbol{x}_i^*)$ is the covariance matrix of the test points, and σ_n is the the signal noise. To find the predicted output of the Gaussian Process, we can find the mean function of the $\boldsymbol{y}^*$, which is the following:

$$\boldsymbol{y^*} = K^*(K + \sigma_n^2 I)^{-1}\boldsymbol{y}$$

σ_n for our purposes will be hand picked.

1.4 Choosing a kernel function

Choosing the kernel function depends on the actual underlying shape of the function learned. If the function is periodic, for example, a periodic kernel function will provide better results, and if a function is linear a linear kernel function will give us better results. I want to find the kernel function that is most likely to give us the most accurate approximation. To set this up, assume I am given p different kernel functions k_i. Define the composite kernel function k as a linear combination of those kernel functions:

$$k(x_i, x_j) = \theta_1 k_1(x_i, x_j) + \cdots + \theta_p k_p(x_i, x_j)$$

where $\theta_i > 0$. The composite kernel matrix K will be equal to the weighted sums of each individual kernel matrix K_i:

$$K = \theta_1 K_1 + \cdots + \theta_p K_p$$

Because K is a linear combination of the individual kernel matrices $K_i \in S_+^n$ and the weights are all positive, $K \in S_+^n$ (I think this was a homework problem at the start of the quarter...).

In vector form, our kernel matrix looks like this:

$$K = \boldsymbol{\theta}^T \boldsymbol{K}$$

where $\boldsymbol{K} = [K_1, \ldots, K_p]$ a vector of the kernel matrices used and $\boldsymbol{\theta} = [\theta_i, \ldots, \theta_p]$ is the the weight of that matrix. Our conditions on $\boldsymbol{\theta}$ are

$$\boldsymbol{\theta}_i > 0 \; \forall \; i \in 1, \ldots, p$$

1.5 Likelihood

The goal is to maximize the likelihood that the training values y are observed from the training values $\boldsymbol{x}$ using the Gaussian Process. it is known that $\boldsymbol{y} \sim \mathcal{N}(0, K + \sigma_n^2 I)$ [2], and therefore the marginal likelihood is:

$$p(\boldsymbol{y}|\boldsymbol{x}) = \frac{1}{(2\pi)^{n/2}} |K + \sigma_n^2 I|^{\frac{1}{2}} \exp(-1/2 \boldsymbol{y}^T (K + \sigma_,^2 I)^{-1} \boldsymbol{y})$$

and the log likelihood is

$$\log p(\boldsymbol{y}|\boldsymbol{x}) = -\frac{1}{2}(n \log(2\pi) + \log |K + \sigma_n^2 I| + \boldsymbol{y}^T (K + \sigma_n^2 I)^{-1} \boldsymbol{y})$$

replacing K with $\boldsymbol{\theta}^T \boldsymbol{K}$. Convincing CVXPY that this function is indeed concave may be a bit tricky, but this seems like a good time to use Gradient Descent!

2 Methods

2.1 Problem setup

Turning this into a minimization problem and removing constants, the convex optimization problem is as follows:

$$\min_{\theta} \; \log |\boldsymbol{\theta}^T \boldsymbol{K} + \sigma_n^2 I| + \boldsymbol{y}^T (\boldsymbol{\theta}^T \boldsymbol{K} + \sigma_n^2 I)^{-1} \boldsymbol{y}$$
$$s.t. \; \boldsymbol{\theta}_i >= 0 \; \forall \; i \in 1 \ldots p$$

where p is the number of kernel functions, $\boldsymbol{K} = [K_1, \ldots, K_p]$ is the vector of the kernels matrices $K_i \in S_{++}^n$, $\boldsymbol{\theta} = [\theta_i, \ldots, \theta_p]$ is the the weights of those matrices. σ_m is the standard deviation of signal noise, chosen to be a value of 0.1. $\boldsymbol{y}$ is our noisy input. It is clear that the objective function is the log likelihood function from above with the constant removed and the values scaled so it is a minimization problem instead of a maximization problem.

2.2 Approach

To find the optimal value of $\boldsymbol{\theta}$, Grwadient Descent will be used. A learning rate of $1e-2$ and a cutoff of $1e-2$, seemed to finish gradient descent in the range of 1000 iterations in around 30 seconds, which was appropriate for my free use of Google Colab.

To generate input data $\boldsymbol{y}$, random gaussian noise with a standard deviation of 0.1 was added to the simple function of $x \in [0, 1]$:

$$\boldsymbol{y}_i = f(\boldsymbol{x}_i) + \epsilon = \sin(20\pi \boldsymbol{x}_i) + \epsilon$$

The underlying function was sinusoidal with a period of 0.1.

Choosing kernel functions is a bit more nuanced. The most common kernel functions were as follows:

$$k_1 = k_{rq}(x_1, x_2) = (1 + \frac{(x_1 - x_2)^2}{2.2 * 0.01})^{-1.1}$$
$$k_2 = k_l(x_1, x_2) = 1/2 + (x_1 - 1)(x_2 - 1)$$
$$k_3 = k_p(x_1, x_2) = \exp(\frac{-2\sin(\pi \frac{|x_1 - x_2|}{0.2})}{0.02})$$
$$k_4 = k_{se}(x_1, x_2) = \exp(\frac{-2(x_1 - x_2)^2}{0.02})$$

which were taken from here [1]. as some of the most popular kernel functions. The hyperparameters of these kernel may have also been able to be optimized using Gradient Descent, however that is out of the scope of this final report so I just chose values that made sense to me. An important thing to note is that the periodic kernel's period hyperparameter is equal to our input function's. From these kernel functions, the corresponding kernel matrices K_i were assembled.

3 Results

The results were great! The ultimate theta chosen for the input function heavily favors the periodic function, as expected:

$$\boldsymbol{\theta}_{opt} = [1.74974111,\ 4.76193398,\ 16.53755544,\ 1.83998171]$$

From graphing the objective function versus the iteration number, we can definitely see it converging on an optimal solution 1. Looking at the Gaussian Process interpolating the function, it looks very close to the truth2 and the error is fairly low 3. Additionally, even once extrapolated to values it never trained near it approximates pretty well! 4

4 Discussion

The results obtained were very good. The optimized theta clearly favored the periodic kernel function almost 16x as strongly as it favored the the other kernel

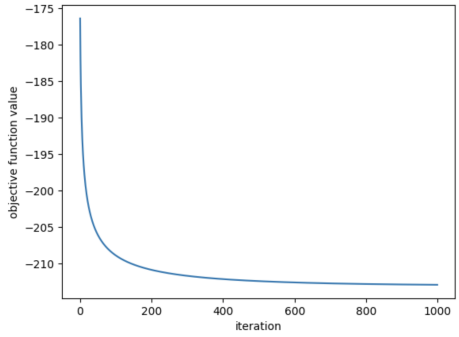

Figure 1: The objective function rapidly decreases, and flattens out as it get closer to an optimal theta

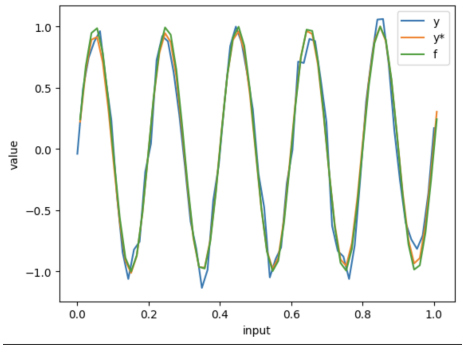

Figure 2: The predicted values of the function $\boldsymbol{y}^*$ compared to the input $\boldsymbol{y}$ and the actual function $\boldsymbol{f}$

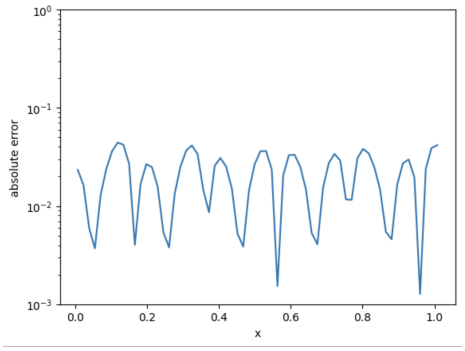

Figure 3: The absolute error of the approximation

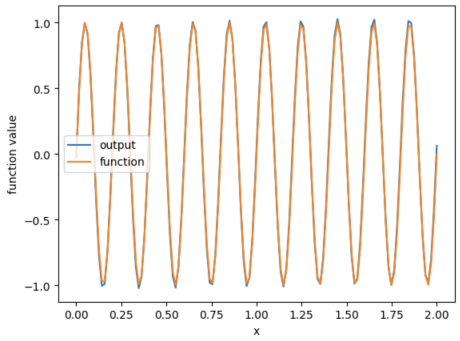

Figure 4: Extrapolation of function

function, except for the linear kernel function. If I were to guess, this is probably because the function sin(x) almost looks like a line if you look at it from afar, so a line with the function $y = 0$ would be a rough approximation. Looking at the errors, the error seems to spike near where the function changes the most. This error is most likely coming from the small squared-exponential and rational quadratic function dependence, which tends to approximate smooth functions the best.

Further studies of model choice can go down many routes. In this project, I manually chose hyperparameter values of the kernel functions, however those can also be optimized often with convex optimization methods like gradient descent. Additionally, one could use gradient descent to find the noise value , instead of manual input like I used. Of course, one could also use the exact same methods I used to approximate different functions.

5 Supporting Information

This code was run on a colab here.

```
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt
from ipywidgets import IntProgress
from IPython.display import display

n = 64
sigma = 0.1
f = lambda x: np.sin(x*10*np.pi)
# f = lambda x: x - 1/2

X = np.linspace(0, 1, n)
Y = np.vectorize(lambda x: f(x) + np.random.normal(0, sigma))(X)
l = 0.1
kernels = [
  lambda x1, x2: (1 + (x1 - x2)**2/(2*1.1)*(l**2))**(-1.1), # Rational Quadratic kernel, l=c
  lambda x1, x2: 1/2+(x1-1)*(x2-1), # Linear kernel, y-intercept = 1/2, x-intercept = 1
  lambda x1, x2: np.exp(-2*np.sin(np.pi*abs(x1-x2)/0.2)**2*l**2), # Pereodic kernel, l=dx1,
  lambda x1, x2: np.exp(-(x1-x2)**2 * l**2), # Squared exponential kernel, l=dx
]
noise = np.eye(n) * sigma**2
K = np.array([np.array([[kernel(x1, x2) for x1 in X] for x2 in X]) for kernel in kernels])
K_tot = lambda theta: np.sum([theta[i] * K[i] for i in range(len(theta))], axis=0) + noise


objective_function = lambda theta: np.log(np.linalg.det(K_tot(theta))) + Y.T @ np.linalg.inv
gradient_function = lambda theta: np.array([np.trace(np.linalg.inv(K_tot(theta)) @ K_t) + Y.

# Gradient descent
iterations = 1000
close_enough = 1e-6
learning_rate = 1e-2

prog = IntProgress(min=0, max=iterations)
display(prog)

obj = []
theta = np.ones(len(K))
for _ in range(iterations):
  prog.value += 1
  theta_old = theta
  theta = theta + learning_rate * gradient_function(theta)[:len(theta)]
```

```
  # Projection
  theta[theta < 0] = 0
  obj.append(objective_function(theta))
  # print(theta_old, theta, objective_function(theta), gradient_function(theta))
  if abs(objective_function(theta) - objective_function(theta_old)) < close_enough:
    break

print("done :)")
print(theta)
```

References

[1] David Kristjanson Duvenaud. *Automatic Model Construction with Gaussian Processes*. PhD thesis, University of Cambridge, 2014.

[2] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.