Model Selection for Gaussian Process Regression

📅Date: 02-05-2023
🕐Time: 22:25

Bipin Koirala

Table of Contents

Preliminary

In a regression setting, we are interested in finding an optimal map $f(x)$ between data points and labels/ function values.

$$f : X \to Y$$

#Gaussian-Process can be used to represent a prior distribution over a space of functions. Gaussian Process can be written as,

$$y = f(x) \sim GP\Big(m(x),\ k(x, x')\Big)$$

Note

This Note contains the following:

(1) Compute the marginal likelihood for Gaussian Process Model

(2) Compute the gradients of (1) w.r.t the hyper-parameters of kernel

(3) Check (2) using Standard Finite Difference

(4) Evaluate (1) and (2) to scipy's 'SLSQP' to maximize log marginal likelihood

---

Model Selection for GP Regression

Although Gaussian Process is a non-parametric model, the so called 'hyper-parameters' in the kernel heavily influence the inference process. It is therefore essential to select the best possible parameters for the kernel. For example in the R.B.F-kernel; the *length* (l) and *scale* (σ) are the hyper-parameters.

$$k(x, x') = \sigma^2 \exp\left(\frac{|x - x'|^2}{2l^2}\right)$$

where, l controls the 'reach of influence on neighbors' and σ dictates the average amplitude from the mean of the function.

Marginal Likelihood

Bayes' Rule

$$\text{Posterior} = \frac{\text{Likelihood} \times \text{ Prior}}{\text{Marginal Likelihood}}$$

$$\mathbb{P}(\theta|y, X) = \frac{\mathbb{P}(y|X, \theta) \times \ \mathbb{P}(\theta)}{\mathbb{P}(y|X)}$$

A #Marginal-Likelihood is a likelihood function that has been integrated over the parameter space. It represents the probability of generating the observed sample from a #prior and is often referred to as the #model-evidence or #evidence.

Let θ represent the parameters of the model. We now formulate a #prior over the output of the function as a #Gaussian-Process.

$$\mathbb{P}(f|X, \theta) = \mathcal{N}\Big(0, k(x, x')\Big) \tag{1}$$

We can always transform the data to have zero mean and (1) can be viewed as a general case. Assume that the #likelihood takes the following form

$$\mathbb{P}(Y|f) \sim \mathcal{N}(f, \sigma_n^2 I) \tag{2}$$

(2) tells that the observations y are subject to additive Gaussian noise. Now, the joint distribution is given by;

$$\mathbb{P}(Y, f|X, \theta) = \mathbb{P}(Y|f)\, \mathbb{P}(f|X, \theta) \tag{3}$$

It is worth noting that we would eventually like to optimize the hyper-parameters θ for the kernel function. However, the #prior here is over the mapping f and not any parameters directly. In the ***evidence-based*** framework, which approximates Bayesian averaging by optimizing the #Marginal-Likelihood we can make use of the denominator part in the ***Bayes' Rule*** as an objective function for optimization. For this we take the joint distribution (3) and marginalize over f since we are not directly interested in optimizing it. This can be done in the following way:

$$\begin{aligned} \mathbb{P}(Y|X, \theta) &= \int \mathbb{P}(Y, f|X, \theta)\, df \\ &= \int \mathbb{P}(Y|f)\, \mathbb{P}(f|X, \theta)\, df \end{aligned} \tag{4}$$

(4) is an integration performed all possible spaces of f and it aims to remove f
from the distribution of Y. After marginalization Y is no longer dependent on f but it depends on the hyper-parameters θ.

As per Rasmussen & Williams, the log marginal likelihood is given by;

$$\log \mathbb{P}(y|X, \theta) = -\frac{1}{2} y^T K_y^{-1} y - \frac{1}{2} \log |K_y| - \frac{n}{2} \log (2\pi) \tag{5}$$

where $K_y = K_f + \sigma_n^2 I$ is the covariance matrix for the noisy targets y and K_f is the covariance matrix for the noise-free latent f. The first term penalizes wrong prediction, second penalizes model complexity and the third monomial is normalization term.

Gradients of Marginal Likelihood

Recall

$\frac{\partial}{\partial \theta} K^{-1} = -K^{-1} \frac{\partial K}{\partial \theta} K^{-1}$

$\frac{\partial}{\partial \theta} \log |K| = \text{trace} \left(K^{-1} \frac{\partial K}{\partial \theta} \right)$

Now, the partial derivatives w.r.t. the hyper-parameters is given by;

$$\begin{aligned} \frac{\partial}{\partial \theta_j} \log \mathbb{P}(y|X, \theta) &= \frac{1}{2} y^T K^{-1} \frac{\partial K}{\partial \theta_j} K^{-1} y - \frac{1}{2} \text{trace} \left(K^{-1} \frac{\partial K}{\partial \theta_j} \right) \\ &= \frac{1}{2} \text{trace} \left((\alpha \alpha^T - K^{-1}) \frac{\partial K}{\partial \theta_j} \right) \end{aligned} \tag{6}$$

where, $\alpha = K^{-1} y$

Warning

In general, computing the inverse of a matrix directly (e.g: np.linalg.inv()) is not stable and there is a loss of precision. In the case when the matrix is positive definite, Cholesky decompostion can be used to compute inverse.

Example:

Let K be a symmetric positive definite matrix. Now, if we want to calculate $\alpha = K^{-1} y$, we can do the following:

$$K = \text{Cholesky } \rightarrow LL^T$$

$$K^{-1} = (L^T)^{-1} L^{-1}$$

$$\alpha = \text{np.linalg.solve(L.T, np.linalg.solve(L, y))}$$

Implementation

```
def kernel(x, xp, σ, l):
        '''k(x,x') = sigma^2 exp(-0.5*length^2*|x-x'|^2)'''
        length = l
        sq_norm = (scipy.spatial.distance.cdist(x, xp))**2
        return σ**2 * np.exp(-0.5*sq_norm/(length**2))
```

```python
def dKdL(x1, x2, σ, l):
        '''
        computes partial derivative of K w.r.t length (l)
        arg: x1 = (N1, D), x2 = (N2, D)
        return: (N1, N2)
        '''
        sq_norm = (scipy.spatial.distance.cdist(x1, x2))**2
        return (σ**2) * np.exp(-sq_norm/(2*l**2)) * (sq_norm) / (l**3)

def dKdσ(x1, x2, σ, l):
        '''
        computes partal derivatice of K w.r.t sigma (std not variance)
        arg: x1 = (N1, D), x2 = (N2, D)
        return: (N1, N2)
        '''
        sq_norm = (scipy.spatial.distance.cdist(x1, x2))**2
        return 2*σ*np.exp(-sq_norm/(2*l**2))


def dLdt(a, iKxx, dKdt):
        '''
        computes gradient of log marginal likelihood w.r.t. a hyper-parameter
        i.e. either sigma or length
        '''
        return 0.5**np.trace(np.dot(a @ a.T - iKxx), dKdt)


def f_opt(kernel, X, y, σ, l):
        '''
        Evalaute Negative-Log Marginal Likelihood
        '''
        σ_n = 0.1 # std of noise hard-coded for now
        K = kernel(X,X, σ = σ, l = l) + (σ_n**2)*np.eye(X.shape[0])
        L = np.linalg.cholesky(K) + 1e-12 # Cholesky decomposition
        a = np.linalg.solve(L.T, np.linalg.solve(L, y)) # compute alpha

        #log_likelihood = -0.5 * y.T @ a - 0.5 * np.trace(np.log(L)) - 0.5 * X.shape[0] * np.log(2*np.pi)
        log_likelihood = -0.5 * y.T @ a - 0.5 * np.log(np.linalg.det(K)) - 0.5 * X.shape[0] * np.log(2*np.pi)

        return -log_likelihood


def grad_f(kernel, X, y, l, σ):
        '''
        Compute gradient of objective function w.r.t. two parameters
        '''
        l, σ = params
        σ_n = 0.1 # std of noise hard-coded for now
        K = kernel(X,X, σ = σ, l = l) + (σ_n**2)*np.eye(X.shape[0])
        L = np.linalg.cholesky(K) # Cholesky decomposition
        a = np.linalg.solve(L.T, np.linalg.solve(L, y)) # compute alpha

        inv_k = np.linalg.inv(K)
        grad = np.empty([2,])
        grad[0] = dLdt(a = a, iKxx = inv_k, dKdt = dKdσ(X, X, σ, l)) # gradient w.r.t sigma
        grad[1] = dLdt(a = a, iKxx = inv_k, dKdt = dKdL(X, X, σ, l)) # gradient w.r.t length

        return grad



def marginal(params, X, y):
        '''
        Evalaute Negative-Log Marginal Likelihood -- for scipy optimization
        '''
        #print (params)
        l, σ = params
        σ_n = 0.1 # std of noise hard-coded for now
        K = kernel(X, X, σ = σ, l = l) + (σ_n**2)*np.eye(X.shape[0])
        L = np.linalg.cholesky(K) + 1e-12 # Cholesky decomposition
        a = np.linalg.solve(L.T, np.linalg.solve(L, y)) # compute alpha

        #log_likelihood = -0.5 * y.T @ a - 0.5 * np.trace(np.log(L)) - 0.5 * X.shape[0] * np.log(2*np.pi)

        log_likelihood = -0.5 * y.T @ a - 0.5 * np.log(np.linalg.det(K)) - 0.5 * X.shape[0] * np.log(2*np.pi)

        return -log_likelihood
```

```
'''
True function f(x) = sin(x) & X ~ Unif(-4,4) with 10 samples
'''
f_sin = lambda x: (np.sin(x)).flatten()
X = np.random.uniform(-4, 4, size = (10, 1))
y = f_sin(X)

# Scipy-optimization via SLSQP

lim = [10**-3, 10**3]
bound = [lim, lim]
start = [0.3, 0.1] # initial hyper-parameters
result = scipy.optimize.minimize(fun = marginal, x0 = start, args = (X, y), method = 'SLSQP', options =
{'disp':True}, bounds = bound, tol = 0.0001)
```

```
'''
Contour Plot
'''

L = np.linspace(10**-3, 10**2, 1000)
S = np.linspace(10**-3, 10**3, 1000)
σ, l = np.meshgrid(L, S)

func_val = np.zeros_like(σ)

for i in range(σ.shape[0]):
        for j in range(l.shape[0]):
                func_val[i, j] = f_opt(kernel = kernel, X = X.reshape(-1,1), y = y.reshape(-1,1), σ = S[i], l =
L[j])

plt.contourf(σ, l, func_val, cmap = 'plasma')
plt.xscale('log')
plt.yscale('log')
plt.scatter(result.x[0], result.x[1], color = 'black', marker = 'x')
plt.colorbar()
plt.show()
```