Gaussian Process Regression From Scratch

Manthan Bagade, Anay

July 2025*

1 Introduction

Gaussian Process Regression (GPR) is a powerful **non-parametric Bayesian** approach that is particularly useful when dealing with problems involving continuous data, where the relationship between input variables and output is not explicitly known or can be complex. It can model certainty in predictions, making it a valuable tool for various applications, including optimization, time series forecasting, and more.
We have implemented GPR from scratch using Python, which allows us to understand the underlying mechanics of the algorithm and how it can be applied to real-world problems. To keep it simple and intuitive, we have implemented only one kernel (refer section 3), the Radial Basis Function (RBF) kernel, for the time being. The code, along with some experiments and results is available on GitHub at `https://github.com/yourusername/GPR`.

2 Why GPR?

Gaussian Process Regression (GPR) is a non-parametric, Bayesian approach to regression. It provides not just predictions, but also confidence intervals (uncertainty estimates) for those predictions. This is particularly useful in scenarios where understanding the uncertainty of predictions is crucial, such as in medical diagnosis, financial forecasting, and environmental modeling. It does not assume a functional form for the relationship between input and output, making it flexible for various types of data. GPR can also handle noisy observations effectively, providing a robust framework for regression tasks. In summary, GPR has several advantages over traditional regression methods such as :

- **Non-parametric Approach:** GPR does not assume a fixed form for the underlying function, allowing it to fit to data with complex relations.
- **Uncertainty Quantification:** It provides not only predictions but also the confidence level of these predictions at each point.
- **Flexibility:** GPR can use multiple kernels, or a combination of them to model different types of the data based on the user's requirements.

*Thanks to Coding Club, IITG for this assignment.

- **Robustness to Noise:** It can effectively handle noisy observations and control the noise level, modeling real world problems.
- **Regularization:** GPR inherently includes a regularization mechanism through the kernel function, which helps prevent overfitting.

3 Kernel

The **Kernel** function is the most important component of GPR. It defines the similarity between points in the input space and is used to compute the covariance matrix of the Gaussian Process. The choice of kernel function can significantly affect the performance of GPR, determines the shape of the functions model is trying to learn, and the confidence it has in its predictions. Different kernels can capture different types of relationships in the data, such as periodicity, smoothness, or linearity. There are several types of kernel functions available such as **Linear, Polynomial, Exponential, Matern, and Radial Basis Function (RBF)**. Each kernel has its own characteristics and is suitable for different types of data. Furthermore, You can combine different types of kernels to create a custom kernel that fits your specific problem by addition or multiplication of this standard kernels.
In this implementation, we use the **Radial Basis Function (RBF)** kernel, also known as the Gaussian kernel. It is particularly popular due to its ability to model smooth functions and its infinite differentiability, making it a good choice for many regression tasks. **RBF** is defined as:

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

where σ^2 is the variance, l is the length scale, and $||x - x'||^2$ is the squared Euclidean distance between the input points x and x'. The parameters σ^2 and l control the amplitude and smoothness of the function, respectively.

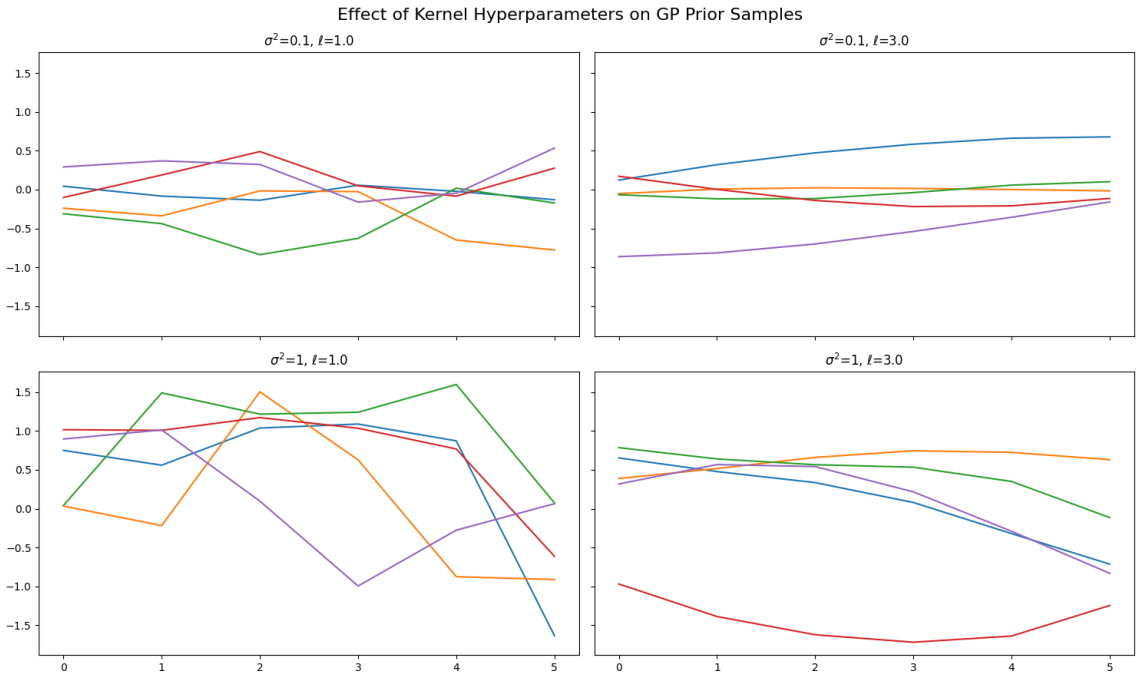

Figure 1: RBF Kernel Visualization

4 The Algorithm

The Gaussian Process Regression algorithm can be summarized in the following steps:

1. **Define the Kernel:** The user will choose a kernel function that defines the covariance between points in the input space. In our case, we use the Radial Basis Function (RBF) kernel.

2. **Compute the Covariance Matrix:** For a given set of training data points, compute the covariance matrix K_{11} using the kernel function. This matrix captures the relationships between all pairs of training points.

3. **Add Noise:** If the observations are noisy, add a noise term to the diagonal of the covariance matrix to account for observation noise. This results in a modified covariance matrix $K_{11} + \sigma_n^2 I$, where σ_n^2 is the noise variance and I is the identity matrix.

4. **Compute the Inverse:** Compute the cholesky decomposition of the covariance matrix. Using this decomposition, you can efficiently compute the inverse of the covariance matrix and using cholesky solve function you can quickly compute $K_{11}^{-1} y$.

5. **Make Predictions:** For a new input points $X_t est$, compute the covariance matrix K_{21} between the training points and the test point, and the covariance matrix K_{22} for the test

point itself. The predictive mean and variance can then be computed using these quantities using the formulas. The formula for the predictive mean μ and variance σ^2 is given by:

$$\mu = K_{21}K_{11}^{-1}y \tag{2}$$

$$\Sigma = K_{22} - K_{21}K_{11}^{-1}K_{12} \tag{3}$$

To know more about the formulas, refer to [2].

6. **Loss Function:** The loss function is typically the negative log marginal likelihood, which can be computed using the covariance matrices and the observed outputs. This loss function can be minimized to optimize the hyperparameters of the kernel.

$$\text{Loss} = \frac{1}{2}y^T K_{11}^{-1}y + \log|K_{11}| + \frac{n}{2}\log(2\pi) \tag{4}$$

where n is the number of training points. The first term represents the fit of the model to the data, the second term penalizes large covariance matrices, and the third term is a constant that does not depend on the parameters.

5 Implementation

The implementation of GPR is done in Python, utilizing libraries such as NumPy for numerical computations and Matplotlib for visualization. The code is structured to allow easy modification of the kernel function and hyperparameters. The implementation is done in two phases:

5.1 Kernels

In this phase, we define thekernel functions and their hyperparameters. The RBF kernel is already implemented, and you can easily add more kernels by defining their respective functions.

```
class RBFKernel:
  def __init__(self, variance = 1.0, length_scale = 1.0):
    self.variance = variance
    self.length_scale = length_scale

  def __call__(self, X1, X2):
    K = (self.variance)*(np.exp(-cdist(X1, X2,
        'sqeuclidean')/(2*self.length_scale**2)))
    return K
```

5.2 GPR Class

In this phase, we implement the GPR class that uses the kernel to perform regression. The class includes methods for fitting the model to training data, making predictions, and computing the loss function.

```
class GaussianProcessRegressor:
  def __init__(self, kernel, noise_variance = 1e-8) -> None:
    self.kernel = kernel
```

```
    self.noise_variance = noise_variance
    self.X_train = None
    self.y_train = None

  def fit(self, X, y) -> None:
    self.X_train = X
    self.y_train = y
    self.K11 = self.kernel(X,X) + self.noise_variance*np.eye(X.shape[0])
    self.L = cholesky(self.K11, lower=True)
    self.alpha = cho_solve((self.L, True), y)

  def predict(self, X) :
    K21 = self.kernel(X, self.X_train)
    mean = K21 @ self.alpha
    K22 = self.kernel(X, X)
    var = K22 - (K21 @ cho_solve((self.L, True), K21.T))
    diag_indices = np.diag_indices_from(var)
    var[diag_indices] = np.clip(var[diag_indices], 1e-8,np.inf)
    return mean, var
  def loss(self):
    negative_log_marginal_likelihood = self.y_train.T @ self.alpha
    penalty_term = 2*np.sum(np.log(np.diag(self.L)))
    constant_term = self.X_train.shape[0]*np.log(2*np.pi)
    return
        0.5*(negative_log_marginal_likelihood+penalty_term+constant_term)
```

6 Results

We have tested the GPR implementation on a synthetic dataset - sine function with some added noise, and compared our model's performace with that of sklearn. Our goal is not to model the data perfectly, but to create alike sklearn's GPR model and test it. The results show that our implementation can effectively capture the underlying function and provide uncertainty estimates for the predictions.

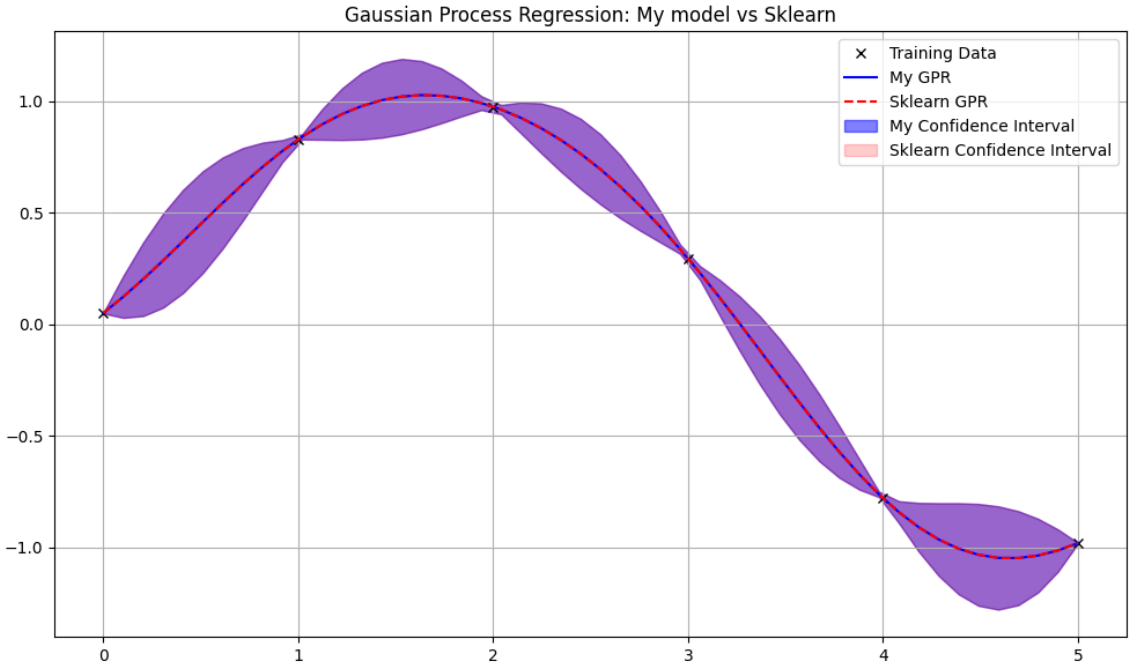

Figure 2: GPR Results on Synthetic Data

References

[1] *Visual Exploration of Gaussian Processes*, by Distill, 2019.

[2] *Gaussian Process - From Scratch*, by Peter Roelants.

[3] *Gaussian Processes*, by Scikit-learn.

[4] *How do I interpret the length scale parameter of the RBF kernel?*, on Data Science Stack Exchange.

[5] *Gaussian Process Regression (GPR)*, by GeeksforGeeks, July 24 2025.