Université Paris Saclay

École CentraleSupelec

UMR 8506 - Laboratoire des Signaux et Systémes L2S

M2 Internship Report:

A review of Python packages for Gaussian process regression

Discipline: Statistics

Author:

Subhasish Basak

Chennai Mathematical Institute, India

Supervised by:

Emmanuel Vazquez

CentraleSupelec, L2S

May-July, 2019

CentraleSupelec, L2S

Gif-sur-Yvette, France

Abstract

In recent years, Gaussian processes have gained popularity as a regression tool in the machine learning community, which has developed several Python packages, with different features and degrees of maturity. Some of the packages that implement Gaussian process regression are, for instance, Scikit-Learn, GPy, GPyTorch etc.
The question addressed in this report is a formal comparison of the existing toolboxes in terms of features, maturity, scalability, cpu charge, and extensibility. The aim is to publish the review and give recommendations for future developments.

Acknowledgements

I would like to thank ...

Contents

Section 1

Introduction

Theoretical and practical developments of over the last decade have made Gaussian processes a serious competitor for real supervised learning applications (Rasmussen & Williams, 2006)[7]. Gaussian process (GP) is commonly used for meta-modelling in simulation experiments where empirical mathematical models of sufficient accuracy are used to substitute complex functions (e.g., Santner et al.)[18]. Moreover due to the recent advances in computer modeling (e.g., in processor performance), the implementation of such computation rich methods have became very convenient using available open source software packages of **Python**, **R**, **Matlab** etc. Although in supervised learning, GP is broadly used as a non-parametric Bayesian approach in both regression and classification problems, in this report we restrict ourselves to the context of ***Gaussian Process Regression*** only.

1.1 Motivation

Many practitioners in the simulation and machine learning community use Gaussian processes for various purposes and are not always familiar with the functionalities, pros and cons of the available libraries, thus they often use the basic **kriging** model and do not try to experiment with the advanced parameter settings, different optimization algorithms and alternative correlation models that are available. In this report our main concern is how Gaussian Process regression is used by practitioners, so the main focus lies in the comparison of different software packages which implement GP regression. Although many available ***python libraries*** do this, but we have found that very different results can be obtained from different libraries even when using the same data and model.

1.2 Goals

Considering a similar study in the literature by (C.B. Erickson et al. 2017)[4] in which the authors describe the parameterization, features, and optimization used by eight different software packages that run on four different platforms

(viz. **R**, **Python**, **Matlab** and **JMP**) and they compare these eight packages using various data functions and data sets, revealing that there are stark differences between the packages. Also in addition to comparing the prediction accuracy, the predictive variance-which is important for evaluating precision of predictions and is often used in stopping criteria—is also evaluated. This report is intended to contribute to the literature of such similar comparative studies and promote more efficient usage of the software libraries and their developments.

1.2.1 A comparative study

Being a non-parametric probabilistic approach, Gaussian Process (GP) regression is unlike other regression techniques (for e.g. linear regression) where given the same input data, all the available software libraries yield exactly the same results. Thus for GP, even if two libraries using the same optimizing criterion and essentially the same equations, given the same input their output models and predictions are different. Therefore a formal comparison of the available python libraries in terms of the scalability, performance, maturity and extensibility is very important before using them to real world problems.
The reason behind choosing **Python** as the platform is that, python is one of the most important programming languages due to its rich ecosystem and it is the most used language in today's Machine learning community. Python libraries implementing GP regression have numerous application landscapes which can be benefited from this type comparative studies.

- ***Machine Learning*** : Regression is essential for any machine learning problem that involves continuous numbers, which includes a vast array of real-life applications in Finance, automobile, geology etc. For e.g. ***Scikit learn*** is one of the most popular libraries for ML practice.

- ***Computational experiments*** : These are experiments involving extensive computational resources to study the behavior of a complex system by computer simulation. libraries like ***Openturns*** are used for such purposes.

- ***Bayesian Optimization*** : GP regression is also a powerful method to perform Bayesian inference about functions. GP provides not just information about the likely value of the function, but importantly also about the uncertainty around that value.

1.3 Methodology

In this report we confine ourselves to a selected set of python libraries. This study consists of the following steps :

1.3.1 Building Wrapper

The primary task is to build a **Universal Wrapper** compatible for all the selected set of libraries. Different libraries have specific dependencies and prerequisites, they also differ on the methods of building the model, making predictions etc. To perform comparative experiments and similar operations on different libraries it is necessary to bring them all under the same roof, which is basically the purpose of building a universal library wrapper.

1.3.2 Identifying features

This study is primarily inspired by the different variety of features that different libraries possess. Features can be anything like the **availability of specific kernel and mean functions**, **available parameter estimation methods**, compatible **python version**, **prerequisite packages & dependencies**, **extensibility to multi-variable setup** etc. Knowledgeable practitioners may know how to improve the results by setting advanced options or tuning parameters. However, the goal is to find what works best for practitioners from specific domain and who may not have this in depth knowledge about the different aspects of the libraries. Thus identifying and summarizing the main features of the different libraries is a crucial step.

1.3.3 Performing tests

After identification of the major features of the libraries we move on to designing test to compare the libraries. We broadly classify these tests into 2 categories viz. ***Functionality Testing*** and ***Performance Testing***. The first one is concerned with the different features of the libraries and the user requirements which are (not) satisfied by the them. The later one is about comparing the different libraries in terms of their prediction accuracy, cpu usage, efficiency , running time etc.

1.3.4 Report outline

In this report , we study various Gaussian process fitting libraries, see Table 1 , and compare their performance, extensibility and responsiveness to different inputs. The report starts with the background material for GP regression consisting definitions and prerequisite concepts. Next we devote a whole section, see Section 3, for identifying and classifying the different features of the libraries and present tabular representations of comparative study of features as a part of ***Functionality Testing***. In section 4 we describe the building ad working methodologies of the universal wrapper. Section 5 onward we design experiments & ***Performance Testing***.

Section 2

Background Concepts

2.1 Definitions

Random variables and vectors.

Let E be a random experiment and $S = \{\Omega, \mathcal{A}, P\}$ be the corresponding probability space, with Ω a sample space, $\mathcal{A}$ a σ field of events and P a probability distribution. A real valued function X defined on Ω is said to be a *random variable* (r.v) if,

$$\{\omega : X(\omega) \leq x\} \in \mathcal{A}, \forall x \in \mathbb{R} \text{ and } \omega \in \Omega.$$

Similarly a d dimensional *random vector* X is a d-tuple vector of the form $X = (X_1, X_2, .., X_d)^t$, where each of $X_i^!s$ are *random variables*.

Probability distribution of a continuous r.v .

The **Cummulative Distribution Function (CDF)** of a real-valued *random variable* X is defined as,

$$F_X(x) = P[X \leqslant x],\ x \in \mathbb{R}$$

If the **CDF** F_X of a real valued random variable X is continuous, then X is called a continuous random variable; if furthermore F_X is absolutely continuous, then there exists a Lebesgue-integrable function $f_X(x)$ such that,

$$F_X(b) - F_X(a) = \mathrm{P}(a < X \leq b) = \int_a^b f_X(x)\,dx$$

for all real numbers a and b. The function $f_X(x)$ is equal to the derivative of $F_X(x)$ almost everywhere, and it is called the ***probability density function*** **(pdf)** of the distribution of X.

Mean, variance and covariance of a cont. r.v .

The **mean** or **expectation** of a random variable X with *pdf* $f(x)$ is defined as,

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x.f(x)\,dx \text{ provided } \int_{-\infty}^{\infty} |x|.f(x)\,dx < \infty.$$

The **variance** of a random variable X with *pdf* $f(x)$ is defined as,

$$\mathbb{V}[X] = \mathbb{E}[X^2] - \mathbb{E}^2[X] \text{ provided } \int_{-\infty}^{\infty} |x|^2.f(x)\,dx < \infty.$$

The **covariance** between 2 jointly-distributed real-valued continuous r.v s X and Y with finite second order moments is denoted by,

$$cov(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])].$$

Let $X_{d\times 1} = (X_1, X_2, .., X_d)^t$ be a random vector, the **mean vector** is defined as a d-dimensional vector, with ith entry defined as $\mathbb{E}[X_i]$, $\forall i = 1(1)d$.

The **covariance matrix** of $X_{d\times 1}$ is defined as the matrix of size $d \times d$ with $(i, j)th$ entry defined as $cov(X_i, Y_j)$ $\forall i, j = 1(1)d$. Clearly the ith diagonal element of the covariance matrix contains $\mathbb{V}[X_i]$.

Stochastic Process

Let $S = \{\Omega, \mathcal{A}, P\}$ be a probability space and $I \subset \mathbb{R}$ is of infinite cardinality. Suppose further that for each $\alpha \in I$, there is a random variable $X_\alpha : \Omega \to \mathbb{R}$ defined on $S = \{\Omega, \mathcal{A}, P\}$. The function $X : I \times \Omega \to \mathbb{R}$ defined by $X(\alpha, \omega) = X_\alpha(\omega)$ is called a ***Stochastic process*** with indexing set I, and is written as $X = \{X_\alpha, \alpha \in I\}$.

Gaussian Process

A continuous random variable X is called a **Gaussian random variable** if the pdf of X is of the following form,

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{2.1}$$

where, $-\infty < x < \infty$, $\mu \in \mathbb{R}$ is the mean and $\sigma^2 > 0$ is the variance of X.

A d-dimensional random vector $X_{d\times 1}$=$(X_1, X_2, .., X_d)^t$ is said to be **Gaussian random vector** if any linear combination of $X_1, X_2, .., X_d$'s is also a Gaussian r.v. A random vector X has a non-degenerate Gaussian Distribution if the covariance matrix is positive definite and the joint pdf of X is of the following form,

$$f_X(\tilde{x}) = \frac{1}{(2\pi)^{(d/2)}\sqrt{|\Sigma|}} e^{-(1/2)(\tilde{x}-\tilde{\mu})^t \Sigma^{-1} (\tilde{x}-\tilde{\mu})^2} \tag{2.2}$$

where $\tilde{\mu}$ and Σ are the corresponding mean vector and covariance matrix.
Now a Stochastic process $\{X_\alpha, \alpha \in I\}$ is called a ***Gaussian Process*** if for all integers $n < \infty$ and $\alpha_1, \alpha_2, \ldots, \alpha_n \in I$, the random vector $(X_{\alpha_1}, X_{\alpha_2}, .., X_{\alpha_n})^t$ has Gaussian distribution.
A Gaussian process is completely specified by its mean function and covariance function. Let the mean function be $m(x)$ and the covariance function be $k(x, x')$ of a Gaussian process $\xi(x)$ defined as,

$$m(x) = \mathbb{E}[\xi(x)] \text{ and } \ k(x, x') = \mathbb{E}[(\xi(x) - m(x))(\xi(x') - m(x'))]$$

Here the random variables represent the value of the function $\xi(x)$ at location x and here the index set I is the set of possible inputs of the function $\xi(x)$. Then we denote the ***Gaussian Process*** as

$$\xi(x) \sim \mathcal{GP}(m(x), k(x, x'))$$

2.2 Gaussian Process Regression

The Bayesian approach of making predictions using a Gaussian process is based on the fact that any finite collection of random variables from a Gaussian process not only has a multivariate normal distribution but also their conditional distributions are also Gaussian.

Lemma 1:
Let Y be a n-dimensional random vector (such that $n_1 + n_2 = n$) partitioned as $(Y_1^{n_1 \times 1}, Y_2^{n_2 \times 1}) \sim N(\tilde{\mu}, \Sigma)$, where $\tilde{\mu}$ and Σ are partitioned as $(\mu_1^{n_1 \times 1}, \mu_2^{n_2 \times 1})$ and

$$\begin{bmatrix} \Sigma_{11}^{n_1 \times n_1} & \Sigma_{12}^{n_1 \times n_2} \\ \Sigma_{21}^{n_2 \times n_1} & \Sigma_{22}^{n_2 \times n_2} \end{bmatrix}$$

Then the ***marginal distributions*** of Y_1 and Y_2 will be,

$$Y_1 \sim N(\mu_1, \Sigma_{11}) \tag{2.3}$$

$$Y_2 \sim N(\mu_2, \Sigma_{22}) \tag{2.4}$$

The ***conditional distribution*** of Y_1 given $Y_2 = y_2$ will be,

$$(Y_1 | Y_2 = y_2) \sim N(\mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(y_2 - \mu_2), \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}) \tag{2.5}$$

Now,given a set of n training data points $\{x_i, y_i\}_{i=1}^n$, to learn a latent function $\xi(x_i)$ transforming the input vector x_i into the target value y_i using GP, we just need to specify the **mean** and **covaiance** function.

2.2.1 Prediction with noise-free observations

For simplicity here we consider the ***Simple kriging*** case with zero mean function. For the Noise-free case our regression model is as follows:

$$y_i = \xi(x_i) \tag{2.6}$$

i.e. the functional values are themselves the target values. The joint distribution of the training outputs, ξ , and the test outputs ξ_* according to the prior is,

$$\begin{bmatrix} \xi \\ \xi_* \end{bmatrix} \sim N\left(0, \begin{bmatrix} K(X, X) & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix}\right) \tag{2.7}$$

If there are n^* test points along with n training points and then $K(X, X_*)$ denotes the $n \times n^*$ matrix of the covariances evaluated at all pairs of training and test points, and similarly for the other entries $K(X, X), K(X_*, X_*)$ and $K(X_*, X)$.
To get the posterior distribution over functions we need to restrict this joint prior distribution to contain only those functions which agree with the observed data points. Using ***lemma 1*** we get the joint posterior distribution for the test points as follows:

$$\xi_*|X_*, X, \xi \sim N(K(X_*, X)K(X, X)^{-1}\xi \\ , K(X_*, X_*) - K(X_*, X)K(X, X)^{-1}K(X, X_*)) \quad (2.8)$$

2.2.2 Prediction with noisy observations

A more realistic modelling situation is that we do not have access to functional values themselves, but with additional noise. Assuming additive independent identically distributed Gaussian noise ϵ, the prior covariance matrix on the noisy observations becomes, $K(X, X) + \sigma_n^2 I$ and our regression model is as follows:

$$y_i = \xi(x_i) + \epsilon_i \quad (2.9)$$

where ϵ_i is Gaussian noise with zero mean and variance σ_n^2. As a result, Introducing the noise term in eq. (2.7) we can write the joint distribution of the observed target values y and the function values ξ_* at the test locations under the prior as,

$$\begin{bmatrix} y \\ \xi_* \end{bmatrix} \sim N\left(0, \begin{bmatrix} K(X, X) + \sigma_n^2 \mathbf{I} & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix}\right) \quad (2.10)$$

Similarly we derive the conditional distribution as follows:

$$\xi_*|X_*, X, y \sim N(K(X_*, X)[K(X, X) + \sigma_n^2 \mathbf{I}]^{-1} y \\ , K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma_n^2 \mathbf{I}]^{-1} K(X, X_*)) \quad (2.11)$$

2.3 Mean function estimation

When the ***Mean function*** is zero i.e. in case of ***Simple Kriging*** we can get the predictions as shown in section 2.2. Even if the mean function is known (i.e. deterministic) we can simply apply the usual zero mean GP to the difference between the observations and the fixed mean function. With

$$\xi(x) \sim \mathcal{GP}(m(x), k(x, x')) \quad (2.12)$$

and the predictive mean becomes,

$$m(X_*) + K(X_*, X)[K(X, X) + \sigma_n^2 \mathbf{I}]^{-1}(y - m(X)) \quad (2.13)$$

and the predictive variance remains unchanged from eq. (2.11). But in practice most of the cases the mean function is unknown and it needs to be estimated. We first consider the following extension of our model in eq.(2.9),

$$y = g(x) + \epsilon, \text{ where } g(x) = \phi^\top(x)\beta + \xi(x) \tag{2.14}$$

where $\xi(x)$ is the same zero mean Gaussian process with covariance function K, $\phi(x)$ is a function which maps a D-dimensional input vector x into an N dimensional feature space, the vector of parameters β now has length N and $\Phi = \Phi(X)$ be the aggregation of columns $\phi(x)$ for all cases in the training set.

2.3.1 Kriging approach

If we assume the basis functions $\phi(x)$ are known and the parameter vector β is unknown, by substituting β by its generalized least square estimate (which is also its **Linear Unbiased Estimate**) given by,

$$\hat{\beta} = (\Phi^\top[K(X,X) + \sigma_n^2 I]^{-1}\Phi)^{-1}\Phi^\top[K(X,X) + \sigma_n^2 I]^{-1}y \tag{2.15}$$

, (assuming $K(X,X)$ is nonsingular and $\Phi(X)$ is of full rank) we get the ***Best Linear Unbiased Predictor*(BLUP)**,

$$\hat{y_*} = K(X_*,X)[K(X,X) + \sigma_n^2 I]^{-1}(y - \Phi\hat{\beta}) + \phi(X_*)^\top\hat{\beta} \tag{2.16}$$

with mse of prediction,

$$K(X_*,X_*) - K(X_*,X)[K(X,X) + \sigma_n^2\mathbf{I}]^{-1}K(X,X_*) + R^\top(\Phi^\top[K(X,X) + \sigma_n^2\mathbf{I}]^{-1}\Phi)^\top R \tag{2.17}$$

where $R = \Phi(X_*) - \Phi[K(X,X) + \sigma_n^2\mathbf{I}]K(X,X_*)$.
This prediction method is called ***kriging***.(see Stein M.L., 1999)[11]. If $\phi(x) = 1$, so that the mean of the process is assumed to be an unknown constant, then best linear unbiased prediction is called ***ordinary kriging***. Best linear unbiased prediction for more general $\phi(x)$ is known as ***universal kriging*** and best linear prediction with the mean assumed 0 is called simple kriging. The mean function for our GP regression setup can be estimated using the kriging approach. The Kriging method depends on the knowledge of how the mean varies functionally with position (i.e., $\phi_i(x)$'s need to be specified) and knowledge of the covariance function.

2.3.2 Bayesian approach

Alternatively, if we take the prior on β to be Gaussian, $N(b,B)$, we can also integrate out these parameters. Following O'Hagan, 1978 [8] we obtain another GP

$$g(X) \sim \mathcal{GP}(\phi(x)^\top b, k(x,x') + \phi(x)^\top B\phi(x')) \tag{2.18}$$

using eq. (2.11) we obtain the predictive distribution of this GP with the mean identical to the **BLUP** in eq. (2.16) and predictive variance given by,

$$K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma_n^2 \mathbf{I}]^{-1} K(X, X_*) + R^\top (B^{-1} + \Phi^\top [K(X, X) + \sigma_n^2 \mathbf{I}]^{-1} \Phi)^\top R \quad (2.19)$$

Taking the limit of the above expressions as the prior on the β parameter becomes vague, $B^{-1} \to O$ (zero matrix) and we get the predictive variance same as the mse of the **BLUP**.

2.3.3 Maximum likelihood approach

Considering our model in eq. (2.14) we can estimate the unknown parameter β while fitting the model,i.e. one could optimize over the parameters β jointly with the hyperparameters of the covariance function Rasmussen & Williams [7]. The method of maximum marginal likelihood in estimating the covariance parameters is described in details in section 2.5.1.

2.4 Covariance function

The kernel $K(.,.)$ plays a crucial role in the predictive mean and variance, they contain our presumptions about the function we wish to learn and define the closeness and similarity between data points. As a result, the choice of kernel has a profound impact on the performance of a GP regression model. Some commonly used kernels are listed as follows.

2.4.1 Some covariance funcions

We present below a selected set of covariance functions which are most commonly used in practice, along with their functional form and other properties.

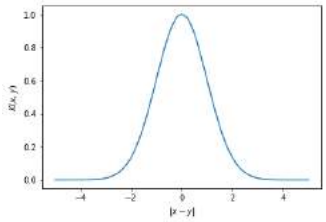
- **Squared exponential kernel**

  The **Squared exponential kernel**, also known as **Radial Basis Function (RBF)** kernel, **Exponentiated quadratic** or the **Gaussian kernel**. It is a stationary kernel of the following form,

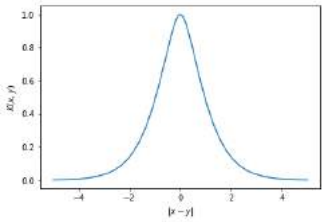
  $$K(x, x') = \sigma^2 \exp\left(-\frac{||x - x'||_2^2}{2l^2}\right) \quad (2.20)$$

  where σ^2 the kernel variance and can also be considered as an output-scale amplitude and the parameter l is the length-scale. We plot the Squared exponential kernel in Figure 2.1(a), with lengthscale parameter 1, with the scale parameter set to 1.

- **Matern kernel**



(a) RBF(l=1)



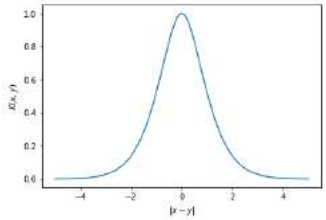
(b) Matern(l=1, ν=1.5)

Figure 2.1: Kernel plots showing $K(x, x')$ vs $|x - x'|$

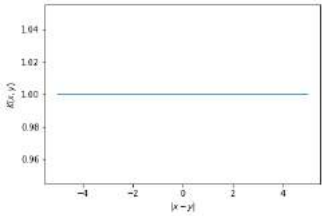
The **Matern kernel** is also a stationary covariance function of the following form,

$$K(x, x') = \sigma^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu} \frac{\|x - x'\|_2}{l} \right)^2 K_\nu \left(\sqrt{2\nu} \frac{\|x - x'\|_2}{l} \right) \tag{2.21}$$

where Γ is the gamma function, K_ν is the modified Bessel function of



(a) Matern(l=1, ν=2.5)



(b) Constant(constant=1)

Figure 2.2: Kernel plots showing $K(x, x')$ vs $|x - x'|$

the second kind, l is the length-scale and ν is the order parameter. When $\nu = p + 1/2$,$p \in \mathbb{N}^+$, the Matérn covariance can be written as a product of an exponential and a polynomial of order p (Abramowitz and Stegun, 1965, eq. 10.2.15)[1]. We elaborate two most commonly used parameter values ν= 3/2 and 5/2.

$$\text{for } \nu = 3/2 \text{ (p = 1)}; \; K_{3/2}(x, x') = \sigma^2 \left(1 + \frac{\sqrt{3}d}{l} \right) \exp\left(-\frac{\sqrt{3}d}{l} \right). \tag{2.22}$$

$$\text{for } \nu = 5/2 \text{ (p = 2)}; \; K_{5/2}(x, x') = \sigma^2 \left(1 + \frac{\sqrt{5}d}{l} + \frac{5d^2}{3l^2} \right) \exp\left(-\frac{\sqrt{5}d}{l} \right). \tag{2.23}$$

where d=$\|x - x'\|_2$. Figure 2.1(b) & 2.2(a) plots two Matern kernels with lengthscale parameter 1 and orders 1.5 and 2.5 respectively, with the scale parameter set to 1.

- **White noise kernel**

The main use of **White noise kernel** is as part of a sum-kernel

where it explains the noise-component of the signal. It tunes its parameter corresponds to estimating the noise-level.

$$K(x, x') = \text{noise level}, if x = x', else\ 0 \tag{2.24}$$

Figure 2.3(a) shows a plot of a White noise kernel with variance 1.

- **Constant kernel**

The **Constant kernel** can be used as part of a product-kernel where it scales the magnitude of the other factor (kernel) or as part of a sum-kernel, where it modifies the mean of the Gaussian process.

$$K(x, x') = \text{constant}, \forall x, x' \tag{2.25}$$

Figure 2.2(b) shows a plot of a Constant noise kernel with constant parameter 1.

(a) White(variance=1) (b) Rat.Qd.(lengthscale=1, power=2)

Figure 2.3: Kernel plots showing $K(x, x')$ vs $|x - x'|$

- **Rational Quadratic kernel**

The **Rational Quadratic kernel** can be seen as a scale mixture (an infinite sum) of RBF kernels with different characteristic length-scales, given by

$$K(x, x') = \sigma^2 \left(1 + \frac{||x - x'||_2^2}{2\alpha l^2}\right)^{-\alpha} \tag{2.26}$$

where σ^2 the kernel variance, α is the order parameter and l is the length-scale. Figure 2.3(b) shows a plot of a Rational Quadratic kernel with lengthscale 1 and power 2, with the scale parameter set to 1.

- **Polynomial kernel**

The **Polynomial kernel** computes the degree-d polynomial kernel between two vectors, it represents the similarity between two vectors.

$$K(x, x') = (x^\top y + c)^d \tag{2.27}$$

where c is the bias or offset parameter and d is the degree parameter.

2.4.2 Stationarity & Isotropy

Other aspects of covariance functions are **stationarity & isotropy**. A stationary kernel is one which is translation invariant:

$$K(x, x') = K(x - x') \tag{2.28}$$

that is, it depends only on the lag vector separating the two observations x and x'. Such a kernel is sometimes referred to as **anisotropic stationary kernel**, in order to emphasize the dependence on both the direction and the length of the lag vector. The assumption of stationarity has been extensively used in time series (Brockwell and Davis, 1991)[3] and spatial statistics (Cressie, 1993)[5] because it allows for inference on K based on all pairs of observations separated by the same lag vector.
If further the covariance function is a function only of $||x - x'||$ then it is called **isotropic (or homogeneous)**,and is thus only a function of distance:

$$K(x, x') = K(||x - x'||) \tag{2.29}$$

2.5 Estimation of hyperparameters

In GP regression models, the hyperparameters involved in the kernel need to be estimated from the training data. Although fully Bayesian approaches like ***Monte Carlo*** methods (Neal, 1997)[14] can perform GP regression without the need of estimating hyperparameters, the common and most classical methods of estimating the parameters is the ***Method of maximum marginal likelihood*** due to the high computational cost of Monte Carlo methods.

2.5.1 Maximum marginal likelihood estimate

considering our model in eq. (2.7), the optimized parameters are obtained by maximizing the marginal likelihood of the target values y given by the integral of the likelihood times the prior of the latent function ξ,

$$p(y|X) = \int p(y|\xi, X)p(\xi|X)d\xi \tag{2.30}$$

The term marginal likelihood refers to the marginalization over the function values ξ. Under the Gaussian process model the prior is Gaussian, $\xi|X \sim N(0, K)$,(assuming zero mean Gaussian Process) and the likelihood is a factorized Gaussian $y|\xi \sim N(\xi, \sigma_n^2 I)$ (for the case of noisy observations), so we can perform the integral yielding the log marginal likelihood and obtain the estimates as follows:

$$\hat{\theta}_{ML} = \arg\min_{\theta \in \Theta} \left(\frac{1}{2} y^\top \Sigma_\theta^{-1} y + \frac{1}{2} \log |\Sigma_\theta| + \frac{n}{2} \log 2\pi \right) \tag{2.31}$$

where, in case of noise-free observations $\Sigma_\theta = K_\theta(X, X)$ and in case of noisy observations $\Sigma_\theta = K_\theta(X, X) + \sigma_n^2 \mathbf{I}$ and y is the observed target values. This

numerical optimization problem has $O(n^3)$ computational cost, which is incurred due to the inversion of the covariance matrix.

2.5.2 Restricted Maximum Likelihod (ReML) estimate

Restricted ("residual") maximum likelihood estimation (REML) of variance and covariance parameters was introduced by Patterson and Thompson (1971)[12] as a method of determining less biased estimates of such parameters than maximum likelihood estimation. In simple problems where solutions to hyperparameters have closed-form, we can remove the bias by multiplying a correction factor. However, for complex problems where closed-form solutions do not exist, we need a more general method to obtain a bias-free estimation for covariance hyprparameters. Restricted maximum likelihood (ReML) is one such method, where the likelihood is so constructed that it becomes independent of some parameters and maximized over the the set of restricted parameters only. A number of simulation studies in the time series setting have demonstrated the general superiority of REML estimation to ML estimation (McGilchrist 1989; and Tunicliffe-Wilson 1989)[19].

2.5.3 Maximum a posteriori probability (MAP) estimate

Another method of estimating the hyperparameters is ***Maximum a posteriori probability (MAP) estimate***. It is a Bayesian approach of parameter estimation in which we assume a prior distribution over the hyperparameters and given the observations we compute its posterior distribution, which is then maximized to obtain the estimates as follows,

$$\hat{\theta}_{MAP} = \arg\max_{\theta} p(\theta|X,y) = \arg\max_{\theta\in\Theta} \frac{p(y|X,\theta)g(\theta)}{\int_{\theta\in\Theta} p(y|X,\theta)g(\theta)d\theta} \tag{2.32}$$

where y and X are the observed target values and inputs respectively and $g(\theta)$ is the prior on θ with domain Θ. Since we have a prior on θ we can compute the marginal likelihood of y given X as $p(y|X) = \int_{\theta\in\Theta} p(y|X,\theta)g(\theta)d\theta$, which will be independent of θ in this case. By Bayes rule we derive the posterior distribution of θ as $p(\theta|X,y)$. The posterior in eq. (2.32) combines the likelihood and the prior, and captures everything we know about the parameters. It can be shown that for Gaussian likelihood and a Gaussian prior on the hyperparameters, the posterior distribution $p(\theta|X,y)$ is also gaussian which has its mean equals to its mode, known as the **MAP** estimate.

2.5.4 Cross validation

Cross-validation is a popular method for choosing model parameters in parametric model settings. Important early references describing cross-validation are (Stone, 1977)[17]; (Hastie et al., 2001)[10] summarize recent applications.

An extreme case of k-fold cross-validation is ***Leave One Out Cross Validation (LOO-CV)***.
Here we present the approach mentioned in Rasmussen & Williams, 2006 [7]. The idea is to split the training data (of size n) into n disjoint,unit sized subsets; validation is done on a single subset and training is done using the union of the remaining n-1 subsets, the entire procedure being repeated n times, each time with a different subset for validation. For Gaussian processes regression, cross-validation is done using the negative log probability loss, given by

$$\log p(y_i|X, y_{-i}, \theta) = -\frac{1}{2}\log \sigma_i^2 - \frac{(y_i - \mu_i)^2}{2\sigma_i^2} - \frac{1}{2}\log 2\pi \tag{2.33}$$

for leaving out training case i, and μ_i and σ_i^2 are computed according to eq. (2.11), in which the training sets are taken to be (X_{-i}, y_{-i}). Accordingly, the objective loss function is,

$$L_{LOO}(X, y, \theta) = \sum_{i=1}^{n} \log p(y_i|X, y_{-i}, \theta) \tag{2.34}$$

which is minimized to get the estimates of the hyperparameters. The computational burden of the derivatives is greater for the LOO-CV method than for the method based on marginal likelihood.

Section 3

Identifying features

In this section we introduce different **features** of the libraries. Features can be categorized in two ways viz. ***Intrinsic Features*** and ***Performance features***. By Intrinsic features we mean the properties which are built-in and which describes a specific software library. On the other hand Performance features are those which shows the ability of a library to perform a particular task. Essentially these features make the libraries perform differently from one another and thus studying them is a necessary part of our report. This section presents a comparative study based on the following features,

Performance Features	**Intrinsic Features**
Available of Mean & Kernel modules	Library release version
Nugget parameter setting	Compatible Python version
Option for Scaling & Compounding kernel	Library Dependencies
Available parameter estimation options	Compatible input data type

Table 3.1: Features

We begin with the the description of the set of python libraries we are using. **Table 3.2** shows the python libraries with their versions, dependencies, compatible python version and input data types.

Library	Version	Python version	Dependencies	Data type
Scikit-learn	0.20.0	2.7,3.4 and higher	-	numpy array
Shogun	6.1.3	3 and higher	-	Shogun Labels
GPy	1.9.6	2.7, 3.4 and higher	-	n-d numpy array
GPflow	1.3.0	3.5 and higher	tensorflow	n-d numpy array
GPytorch	0.3.2	3.6 and higher	torch, Pytorch	tensors
Openturns	1.13	2.7,3.3 and higher	-	Openturns labels

Table 3.2: Python libraries

3.1 Data loading & Reshaping

In supervised learning we need ***Training data*** for feeding into the model and estimating the parameters. ***Testing data*** in on which we perform the prediction. Different python toolboxes uses different input data structures and formats, both for training and testing data. So the simulated/collected data needs to be transformed/reshaped according to the specific library configuration. Table 3.1 shows the different data configurations adopted by the different libraries.

3.2 Mean & Kernels modules

For our purpose we confined to a subset of ***kernels*** (covariance functions) and ***Mean functions***. The following table shows their possibility of implementation in different libraries, Apart from this selected set of kernels there are also

Library	RBF	Matern	White	Constant	Rat. Qd.	Poly.
Scikit-learn	✓	✓	✓	✓	✓	✓
Shogun	✓	-	-	✓	-	-
GPy	✓	✓	✓	-	✓	✓
GPflow	✓	✓	✓	✓	✓	✓
GPytorch	✓	✓	✓*	-	-	✓
Openturns	✓	✓	-	-	-	-

Table 3.3: available Kernel modules,(✓* : Not implemented)

many other kernel modules implemented in these libraries. Specially ***Scikit-learn***, ***GPflow***, ***GPy*** have a large collection of inbuilt kernel modules, viz. Cosine Kernel, Cylindrical Kernel, Periodic Kernel, Linear Kernel, Spectral Mixture Kernel, Exponential Sine Squared kernel etc.
All the libraries set a default **zero mean** and perform **simple kriging**. ***Scikit-learn*** has an option called "**normalize_y**" which is by default **Not active** and it assumes the target values have zero mean. If set to **active** it normalizes the target values by subtracting the mean and adds it again while making predictions.
For the libraries **Shogun**, **GPy**, **GPytorch**, **GPflow** and **Openturns** all of them have an option for a **Constant mean**. By default the constant value set for the constant mean is zero but the user can specify the constant. Only **Openturns** has an option to estimate the constant,i.e. it supports **ordinary kriging**.
Both ***GPflow*** & ***Openturns*** provide an option for **Linear Mean function** and for ***Openturns*** there is in-bulit quadratic basis functions aslo, with an extension to use polynomial basis functions.

Library	Linear mean	Zero mean	Constant mean
Scikit-learn	-	✓	-
Shogun	-	✓	✓
GPy	-	✓	✓
GPflow	✓	✓	✓
GPytorch	-	✓	✓
Openturns	✓	✓	-

Table 3.4: available Mean modules

3.3 Nugget effect

The nugget has the effect of smoothing the objective function and allowing for noise. Another reason for using a nugget is to provide computational stability. All the calculations shown in Section 2 require inverting the matrix K_θ , which can be near singular. Most of the libraries use a **Cholesky decomposition** to solve a system of linear equation to find the inverse. Adding a nugget will improve its stability in terms of numerical computations (Andrey Pepelyshev, 2010)[2]. Some libraries (see nugget type 2 in Table 3.5) support adding ***Heteroscedastic nuggets*** (i.e. the noise variance varies across the input domain), while some other libraries support only adding ***Homoscedastic nuggets***. Let K_θ be the covariance matrix which is near singular. If a nugget, δ , is included in the model, then the covariance matrix is increased along the diagonal by δ :

$$K_\theta^\delta = K_\theta + \delta \mathbf{I}$$

Table 3.5 shows the nugget settings for different libraries.

Library	Can set nugget	Nugget type	Default nugget	Can Optimize nugget
Scikit-learn	✓	2	1e-10	-
Shogun	-	-	1	✓
GPy	✓	2	1	✓
GPflow	✓	1	1	✓
GPytorch	✓	1	.693	✓
Openturns	✓	2	-	-

Table 3.5: Nugget types :: 1: Homoscedastic; 2: Heteroscedastic

3.4 Scaling & Compounding kernels

Depending on the domain of use, one may like to use a combination of more than one covariance functions, this is known as Compounding kernels. Another practice is to take a scaling factor to each of the kernels, called scaling of kernels also sometime known as the ***variance*** of the kernel. For e.g. very often practitioners take a combination of White noise kernel added

(this is equivalent to adding nuggets) to Matern/RBF kernel times a Constant kernel. Direct construction (mainly sum and product) of compound kernels of more than one kernels is supported by some libraries. For **Scikit-learn** one can not directly set the kernel scale parameters, but the purpose can be fulfilled by constructing a compound kernel by multiplying the main kernel by a constant kernel. **Table 3.6** gives description about compound & scale kernels for different libraries.

Library	Default Scale	Can set scale	Can optimize scale	Supports compounding
Scilit-learn	-	Const Kernel	✓	✓
Shogun	1	-	✓	-
GPy	1	✓	✓	✓
GPflow	1	✓	✓	✓
GPytorch	0.693	✓	✓	✓*
Openturns	1	✓	✓	-

Table 3.6: Scaling, Compounding (✓* : Not implemented)

Among the kernels we have chosen **RBF**, **Matern**, **Rational Quadratic** are all isotropic and stationary. **Polynomial** kernel is not isotropic and non-stationary.

3.4.1 Multidimensional input

It real life implementation the input variable is multidimensional in most of the cases. Although all the selected libraries supports multidimensional input, but their performances vary a lot. Except for ***Shogun***, in all the other libraries, the user can define different length-scale parameter to the kernel for different dimensions. In higher dimensions to deal with high computational complexity some of the libraries implement "**Approximation algorithms**" for interpolating the covariance matrix. ***GPytorch*** provides some options including **Grid GP** (for Grid Structured Training Data), **KISS-GP** which is used for kernel interpolation for scalable structured Gaussian processes. KISS-GP is more scalable than inducing point alternatives and can be used for fast and expressive kernel learning costing $O(n)$ time and storage for GP inference.(Wilson et al., 2015)[20] etc.

3.5 Parameter Estimation

For a particular Kernel function there is the **scale** parameter and the hyper-parameters of the kernel itself. For the Mean function it has its own parameters depending upon the type. Further if nugget effect is present, we need to estimate the optimal value of the nugget parameter also. Almost all the libraries have default values set to these parameters and the user has different choices

to optimize them. There are several parameter estimation methods which are used by the libraries, for e.g. ***Maximum likelihood*** or making predictions using ***Markov chain Monte Carlo***. For our selected set of libraries all of them supports ***Maximum likelihood*** method which involves the optimization of the likelihood function. Different libraries use different algorithms to optimize the likelihood. **Table 3.7** shows the parameter estimation settings of different libraries and the available algorithms they use for optimization.

Library	Parameter Estimation	Default Optimizer	Other Optimizers
Scikit-learn	MLE	L-BFGS-B	-
Shogun	MLE	Grad. Desc.	-
GPy	MLE,MCMC	L-BFGS-B	TNS,BFGS
GPflow	MLE,MCMC	L-BFGS-B	-
GPytorch	MLE	Adam optimizer	-
Openturns	MLE	Cobyla	TNC,SQP,NLopt

Table 3.7: Parameter Estimation

3.5.1 Likelihoods

Our main focus has been on regression with Gaussian noise, however, Gaussian processes can be used as priors with other likelihood functions. For example,(Diggle et al., 1998)[13] were concerned with modelling count data measured geographically using a Poisson likelihood with a spatially varying rate. They achieved this by placing a GP prior over the log Poisson rate. (Neal, 1997)[15] implemented GP regression by using a Student's t-distributed noise model rather than Gaussian noise.
Since the ***Gaussian likelihood*** is the most commonly used likelihood function all the libraries can implement that, but some of the libraries provide other choices too. ***Scikit-learn***, ***GPy***, ***GPflow*** and ***Openturns*** the default implementation is with Gaussian likelihood, but only ***GPy*** provides the option to choose other likelihoods. For ***Shogun*** and ***GPytorch*** no default likelihood is set, user can choose from available list of implemented likelihood functions. Some other popular implemented likelihood functions are **Poisson**, **Softmax**, **Bernoulli**, **Student's t** etc. The choice of the likelihood depends on the data also. In case of discrete count type or binary data, using Poisson or Bernoulli likelihood is preferable. However exact computation of posterior is no longer analytically tractable due to non-Gaussian likelihood, several approximation methods are used (Nickisch et al. 2008)[9].

3.5.2 Partial estimation

Sometimes the user may want to optimize some of the hyperparameters of the model instead of all. As we have seen that, although in all the libraries the user can set the kernel parameters, but not all of them supports specifying the

nugget or scale parameter. Moreover ***Scikit-learn*** and ***Openturns*** does not support optimization of the nugget parameter. For ***GPy*** there is an option to fix a particular parameter value and optimize the rest. No other library provide direct option for partial estimation of hyperparameters.

Section 4

Universal wrapper

In this section we describe the methodology of building the python wrapper. As mentioned in section 1.3.1 it is necessary to build a Universal wrapper using which we can implement GP regression using any of the selected libraries. Moreover the wrapper should be compatible with adjusting all the features as well as tuning the parameters. We briefly describe the structure of our wrapper incorporating all the features discussed in the previous section.

4.1 Structure

The wrapper is built in the form of a python package named ***pythongp***, which consists of several subpackages & modules. We present below (see fig. 4.1) a diagrammatic representation of the package ***pythongp***.

4.1.1 Subpackages & modules

The ***pythongp*** package has three subpackages, viz. ***core***, ***test_functions*** and ***tests***. Each of the subpackages have different functionalities and are built accordingly.

- **core**
  This subpackage is the main repository for storing the library specific wrappers and the parameter modules. textbf*core* has furthur 2 subpackages named ***params*** and ***wrappers***. ***params*** contains 2 modules ***kernel*** and ***mean***, whereas ***wrappers*** contains six modules corresponding to the 6 chosen libraries.

- **test_functions**
  The subpackage ***test_functions*** contains a library of test fuctions used for experiments.

- **tests**
  The subpackage ***tests*** contains a library of different performance tests.

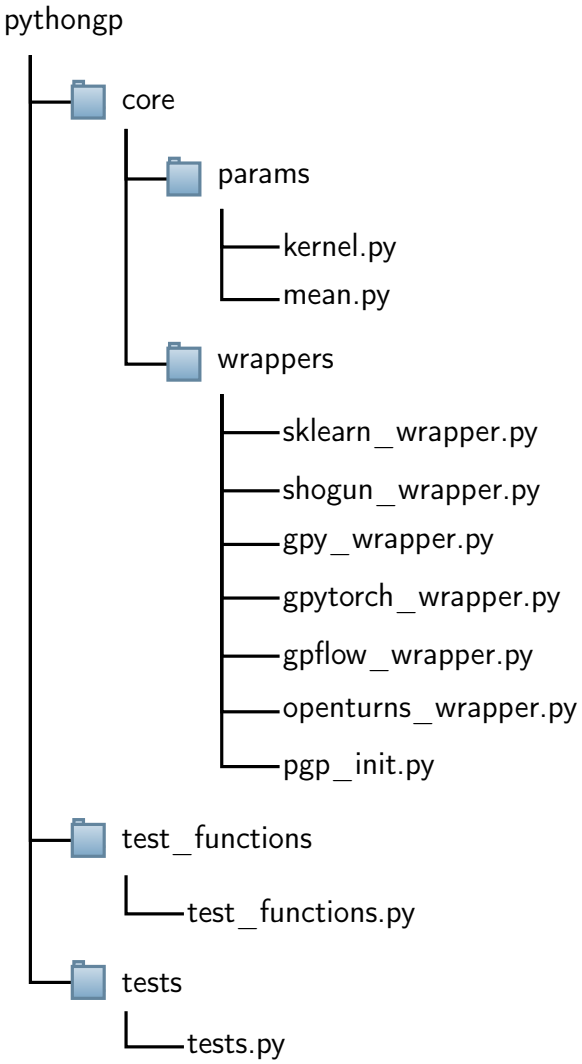


Figure 4.1: File structure of the wrapper

4.2 Functionalities

All the wrappers corresponding to the libraries has the same structure and they can implement the following methods. These methods are built considering all the necessary tasks that are needed to be performed in order to implement GP regression. To summarize we can list all the necessary tasks along with the methods that are implemented in the wrapper to perform them,

4.2.1 Initialization

To run a test we need to initialize the library we want to work with. There is a wrapper module named **pgp_init.py** which takes input the library name and returns the corresponding library wrapper. Now using this wrapper the user can construct tests. There is a library of implemented test functions (both single and multidimensional) in the module **test_functions.py**. The user can

import test functions along with their functional domains for using them in tests.

4.2.2 Data loading & reshaping

The primary step for implementing a regression task is to feed the training data to the system. In our case for the purpose of experiments, we have mostly worked with simulated data but in real life applications we have the data stored in various file formats which needs to be put in the code. Each of the wrapper in ***pythongp*** has 2 methods implemented, which loads the training as well as the test data and re-configures according to the library requirements.

- **load_data**: This method feeds the wrapper with train & test data.
- **load_mult_data**: This method feeds the wrapper with multivariate train & test data.
- **dftoxz**: This method reshapes the data according to library configuration.

4.2.3 Specifying the mean function

The next step is to specify the mean type for the regression. The mean function is treated as a **class** object named **Mean**, inside the module named **mean.py** located in the **params** subpackage. The user needs to create a class **Mean** by importing the module form **params** subpackage. This class supports the following methods,

- **construct**: This method constructs the **Mean** class by either taking user inputs about the mean type or it takes previously stored inputs.
- **get_mean_type**: This method upon calling returns the mean type specified.

Now once the **Mean** class is constructed the user can feed it in the wrapper using the method called **set_mean**.

- **set_mean**: This method takes the **Mean** class as input and calles the corresponding mean function implemented in the specific libraries.

4.2.4 Constructing the kernel

The covariance function is also treated as a class object ust like the mean function. The user can create a **Kernel** class which is defined inside **params** subpackage. This class supprots the following the following methods,

- **construct**: This method constructs the kernel by taking user inputs about the kernel type and its parameter values. Alternatively it can also take as input, a dictionary of specific format containing all the desired parameter values and the kernel type.

- **set_bounds**: This method is used to specify the lengthscale bounds of some of the kernels.
- **set_name**: This method upon calling returns the kernel name.
- **show**: This method upon calling returns the parameter values given as input.

Sometimes the user may want to create a compound kernel (a kernel constructed as a combination of more than one kernels). In that case all the elementary kernels are needed to be constructed like as described. Now for creating a compound kernel there is a latent class named **CompoundKernel** inside the **kernel** module.
The **CompoundKernel** class is an implementation of a **binary tree** structure which combines different kernels. A node (except the leaves) in the binary tree denotes either ***sum*** or ***product*** operator. The two children of each node again has the same **CompoundKernel** structure. The leaves stores the elementary kernels. The **CompoundKernel** class has ***add*** and ***mult*** methods which enables the user to combine elementary kernels using "+" or "*" symbols and store them automatically in the tree structure. Once the **Kernel** class is built the user can feed it into the wrapper using the following methods,

- **set_kernel**: This method calls the get_kernel function and sets the kernel_function in the wrapper class.
- **get_kernel_function**: This method returns the evaluated kernel function.
- **get_kernel**: This method takes input the Kernel_Tree and returns the evaluated the kernel function.

4.2.5 Building regression model

Once the mean and kernels are constructed the next step is to build the regression model using them. All the libraries have their own model classes which are called to build library specific GP models. The wrappers have the following methods fr this task,

- **init_model**: This method constructs the regression model by call. It takes three arguments as follows,
  - **param_opt**: The option for parameter optimizer setting. Currently 2 alternatives are available viz. **MLE**, for Maximum likelihood estimation and **Not_optimize** for kepping all the parameters fixed.
  - **itr**: This sets the number of iteration of the optimizer.
  - **noise**: This specifies the nugget parameter.
- **optimize**: This method optimizes the parameters of the model.

4.2.6 Making predictions & plots

The test data is feeded in the model and predictions are made using the following methosd.

- **predict**: Makes prediction for the test data.
- **predict_mult**: Makes prediction in the multivariate case.

For plotting the predictions in One dimension there is a plot function outside the wrapper in **tests.py** module, so that the tests can directly call that function to generate plots. For multivariate plotting there is function named **contour** which generates contour plots. For predicting the accuracy of the model a function named **accuracy** is implemented in the **tests.py** module which returns several accuracy measures.
All the methods used in the wrapper are generalized to all libraries and by calling them sequentially the user can perform GP regression using any of the library.

Section 5

Performance Tests

In this section, we discuss a class of **Empirical testsusing** which we will compare different libraries. When constructing a metamodel, two properties of GP modeling are of primary importance:

- Accuracy of actual prediction
- Accuracy of the estimate of prediction error

The first one is important for obvious reasons. The second is important to allow the practitioner to assess whether the metamodel is fit for use or whether additional data is needed to improve its fit.

5.1 Defining Measures

In this section we define some statistical measures to compare the aforementioned properties of a GP modelling. To measure the accuracy of the model's prediction, we use the square root of the mean of the squared errors at the prediction points.

$$\text{EMRMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\xi(\mathbf{x}_i^*) - \xi(\hat{\mathbf{x}}_i^*))^2} \tag{5.1}$$

(Santner et al., 2003, p. 108)[18] named this measure as ***Empirical Model Root Mean Square Error***. Where x_i^*'s are test points and $\xi(x_i^*)$, $\xi(\hat{x}_i^*)$ are observed and predicted values at those points respectively. This measures gives a single number which can be roughly thought as the standard deviation of the error, that summarizes the quality of the fit of the model. The less the EMRMSE better the fit of the model.
Next to measure the accuracy of the estimate of prediction error we estimate the model's predicted posterior variance $\hat{\sigma}^2$ obtained by substituting the fitted model's parameter estimates for the unknown parameters in Eq. (2.7) . The square root of the average of the estimated mean squared errors over all prediction points is used as the summary measure for the predicted model RMSE

and called the “PMRMSE.”

$$\text{PRMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\hat{\sigma}_i^2} \tag{5.2}$$

(C.B. Erickson et al. 2018) [4] suggested "Since EMRMSE and PMRMSE both measure the model's RMSE, we expect them to be approximately equal". Thus if we observe ERMSE $\approx$ PMRMSE it confirms the accuracy of the model's prediction error. If EMRMSE is much larger than PMRMSE, then the model is overconfident in its fit, since its estimated prediction errors will be smaller than the empirical errors. Conversely, if EMRMSE is much less than PMRMSE, then the model's estimated prediction errors are conservative.
Another approach to measure the accuracy of the fitted model is using ***Leave One Out Cross Validation*** (**LOOCV**). LOOCV uses a single observation from the original sample as the validation data, and the remaining observations as the training data. This is repeated such that each observation in the sample is used once as the validation data. Leave-one-out cross-validation is usually very expensive from a computational point of view because of the large number of times the training process is repeated.

5.2 Some Empirical tests:

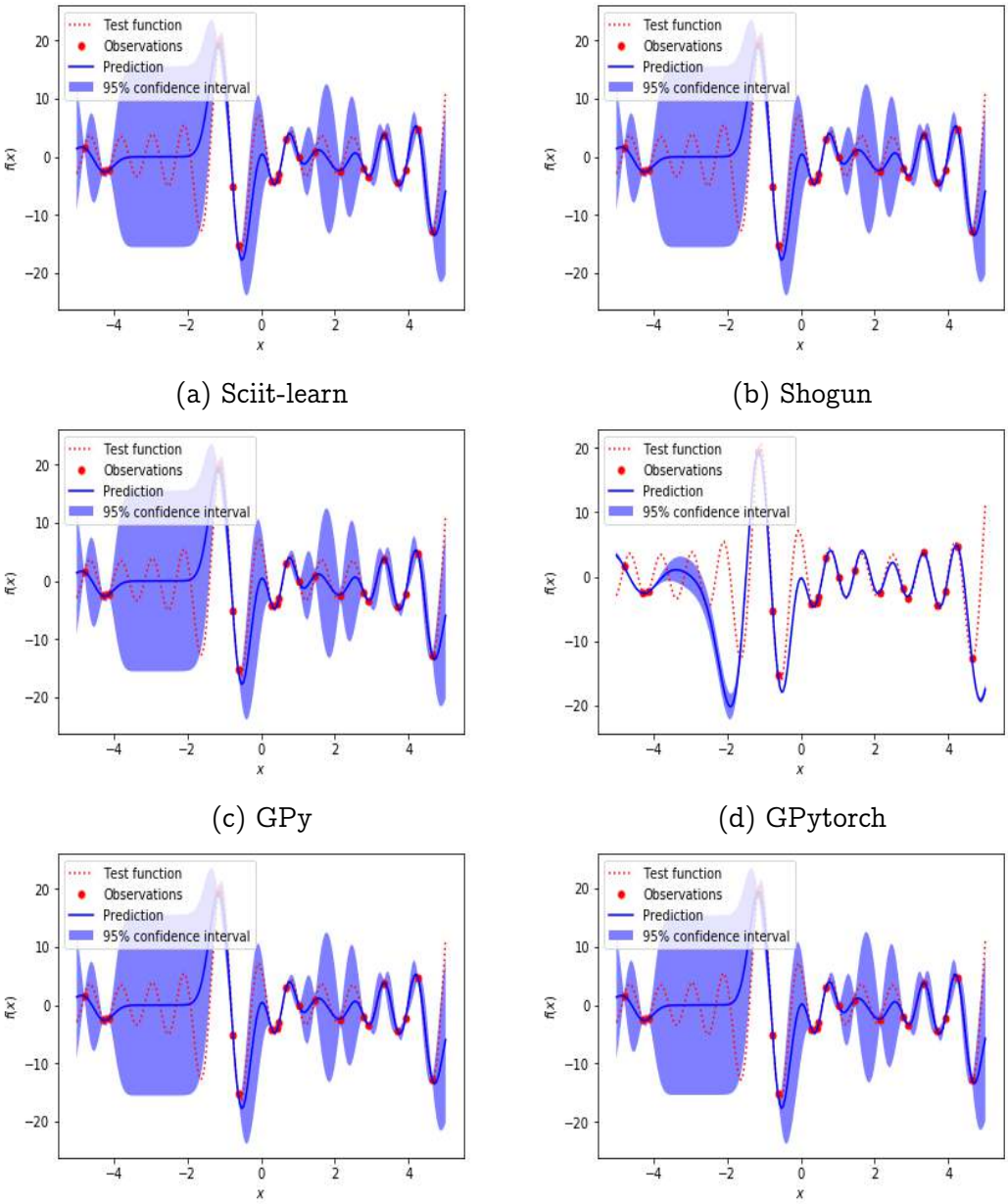
In this section we compare the selected set of python libraries on some test functions. For the tests we have generated the **Training sample** of size n_t form the corresponding input domains, as a **Uniformly distributed** random sample and of much larger size, n_s points for **Test sample**. For each of the libraries we record the **Running time**, **EMRMSE**, coefficient of determination and the estimated parameters.

5.2.1 Simple one dimensional test function

We start with a simple one dimensional test function defined as follows,

$$f(x) = -\sum_{k=1}^{6} k\cos[(k+1)x + k] \tag{5.3}$$

We take the bound of the function domain as $[-5, 5]$, with $n_t = 20$ and $n_s = 200$. Both the training and test points were generated from a Uniform distribution defined on the domain. The covariance function setting for each of the library was kept the same with the **RBF** kernel with lengthscale parameter 1 and kernel scale parameter 1. The nugget for the prediction was kept at low level of 0.0001. The mean function was set to the default **zero mean** of that of simple kriging. We present below the plots of the prediction vs the actual test function observations in fig 5.1. Table 5.1 shows the summary statistics of the prediction and the estimated parameter values for the different libraries. As we can see



(a) Sciit-learn

(b) Shogun

(c) GPy

(d) GPytorch

(e) GPflow

(f) Openturns

Figure 5.1: Plots showing simple kriging predictions for different libraries

from the plots almost all the libraries yield similar predictions for this simple test function. However **GPytorch** performed a bit differently. To observe the differences in prediction among the libraries it is important to note the estimated parameter values, as shown below (see Table 5.1). The table clearly indicates that there are major differences in the estimated parameter values. Considering the running time of the code **Scikit-learn** is the fastest library with running time less than a second, whereas in the other hand **Shogun** takes more than 9 seconds to implement the same. From this particular experiment we found that although we are using the same kernel, mean and exactly the

Library	Run time	EMRMSE	Coeff. of determination	Corr. coeff.	lengthscale & scale	nugget
Sklearn	0.8763	4.4171	0.4561	0.7247	0.257, 63.04	Unch
Shogun	9.9768	4.4177	0.4559	0.7246	0.132,7.93	0.0002
GPy	2.4887	4.4176	0.4559	0.7246	0.257,63.04	8.61e-09
GPflow	3.7210	4.4176	0.4559	0.7246	0.257,63.03	1.02e-06
GPytorch	1.1903	6.2650	-0.0941	0.6409	0.538,1.60	9.99e-05
Openturns	1.6085	4.3751	0.4663	0.7286	0.089,7.85	Unch

Table 5.1: Test results for One D test function

same input data, predictions can be different based on the specific library.

5.2.2 The Branin finction

Next we move on to higher dimensions with the ***Branin finction*** which is a **2-D** function with input domain $[-5, 10] \times [0, 15]$. It has a global minimum with functional value of 0.398 at the points $(-3.142, 12.275)$, $(3.142, 2.275)$, $(9.425, 2.425)$.

$$f(x) = \left(x_1 - \frac{5.1x_0^2}{4\pi^2} + \frac{5x_0}{\pi} - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)cos(x_0) + 10 \tag{5.4}$$

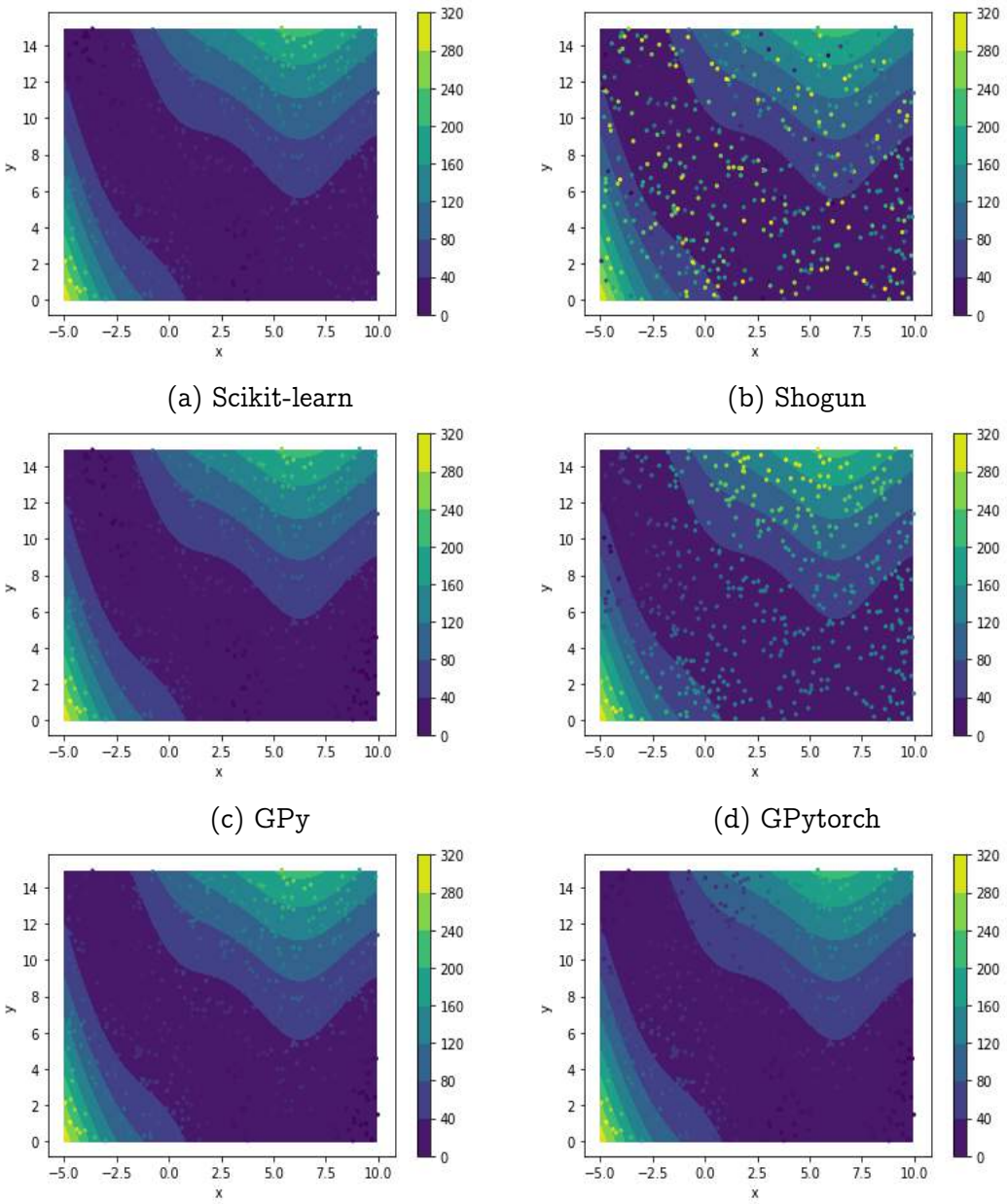
In this test also we have used the **RBF** kernel with initial lengthscale parameter set to 1 for both the dimensions, kernel scale parameters set to 1, nuggets set to 0.0001 and with **zero mean**. THe training and test points are generated by the same strategy as in section 5.2.1. Here we have taken $n_t = 50$ and $n_s = 500$.

Library	Run time	EMRMSE	PMRMSE	Coeff. of determination	Corr. coeff.
Sklearn	0.2735	54.7016	0.8750	-0.1638	0.5924
Shogun	0.9548	71.2286	1.3986	-0.9734	0.0206
GPy	0.3462	54.7017	0.8750	-0.1638	0.5924
GPflow	2.2883	54.7016	0.8750	-0.1638	0.5924
GPytorch	0.2509	62.3077	nan	-0.5100	0.7223
Openturns	0.2190	54.7016	0.8750	-0.1638	0.5924

Table 5.2: Test results without optimization

As mentioned in section 3.4.1, for multivariate setup the implementation in **GPytorch** is done using **KISS-GP** (Wilson et al., 2015)) [20], which is an approximation method for kernel interpolation with low time complexity. Thus even using the same settings, the results obtained with not optimized parameters for **GPytorch** are different from the other libraries. **Shogun** also yield different results inspite of having fixed parameters and identical settings. One plausible reason may be that there is not option to prefix the nugget parameter and specifying different lengthscale parameters for different dimensions in

Shogun. Table 5.2 and 5.3 shows the results with ***Branin function*** with non-optimized and optimized parameters respectively.



(a) Scikit-learn

(b) Shogun

(c) GPy

(d) GPytorch

(e) GPflow

(f) Openturns

Figure 5.2: Contour plots of Simple kriging predictions in 2D

We clearly notice that when the parameters are not optimized almost all the libraries are performing somehow in the same manner, but when we estimate the parameters their performances differ a lot. In case of predictions with optimized parameters, ***GPflow*** is most time consuming and ***Openturns*** is the fastest. However performance-wise ***GPy*** has the lowest EMRMSE, Lowest PM-RMSE (and both the measures are not much different, which indicates a good estimate of the prediction accuracy). It also has the highest Correlation Coeffi-

cient value among the predicted and the observed test points. The **KISS-GP** approximation algorithm of ***GPytorch*** does not yield good prediction compared to other libraries and **Shogun** performs the worst. One plausible We

Library	Run time	EMRMSE	PMRMSE	Coeff. of determination	Corr. coeff.
Sklearn	1.1240	7.0212	13.5093	0.9808	0.9910
Shogun	0.4206	384.5779	289.7910	-56.5284	-0.0003
GPy	3.1960	1.1645	2.5192	0.9994	0.9997
GPflow	28.5465	2.6066	6.6640	0.9973	0.9987
GPytorch	0.3802	46.9196	nan	0.1437	0.7695
Openturns	0.2231	19.7488	36.0832	0.8482	0.9364

Table 5.3: Test results with optimized parameters

use the same training & test points with the same kernel structure and record the results and the the estimated hyperparameters (see Table 5.4). Followings are the contour plots of the same with fixed parameters.

Library	Kernel lengthscale	Kernel scale	Nugget
Sklearn	4.19, 18.6	99856	Unch
Shogun	66.61	489.99	33.22
GPy	4.94, 47.63	1821324.71	5.5626e-309
GPflow	4.43, 25.83	246987.56	1e-06
GPytorch	0.98, 0.99	1.69	9.99e-05
Openturns	3.18,6.84	86.28	Unch

Table 5.4: Optimized parameters

5.3 Simulation test for prediction accuracy

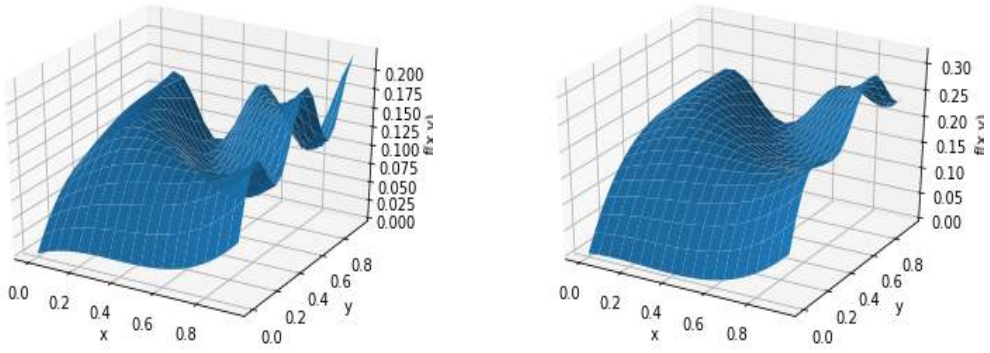
In this subsection we compare the libraries on the basis of their prediction accuracy. The measure used for this is the **EMRMSE** as defined in eq. (5.1). Here we use simulated test beds for generating surfaces on which predictions are made. The test methodology followed here is drawn from the textbook by Santner et al.,2003 [18].

5.3.1 Test bed 1

The first test bed of functions we have considered is "near-cubic" surfaces given by,

$$y(x_1, x_2) = \frac{x_1^3}{3} - (R_1 + S_1)\frac{x_1^2}{2} + (R_1 S_1)x_1 + \frac{x_2^3}{3} - (R_2 + S_2)\frac{x_2^2}{2} + (R_2 S_2)x_2 + A \sin \frac{2\pi x_1 x_2}{Z} \quad (5.5)$$

the six model coefficients $(R_1, S_1), (R_2, S_2)$, and (A, Z) were mutually independent random variables, with the following distributions,
R_1 and $S_1 \sim U(0, 0.5)$
R_2 and $S_2 \sim U(0.5, 1.0)$
$A \sim U(0, 0.05)$ and $Z \sim U(0.25, 1.0)$. The additive sin(.) term has a scale factor Z that provides between one and four oscillations in the product $x_1 x_2$ with an amplitude that ranges up to 0.05. The small amplitude coefficient of the sin(.) term, A, assured that there were only minor deviations from the cubic model. Two $y(x1, x2)$ functions drawn using this stochastic mechanism are displayed in Fig. 5.3. These surfaces are smooth but can contain a significant interaction.



(a) (b)

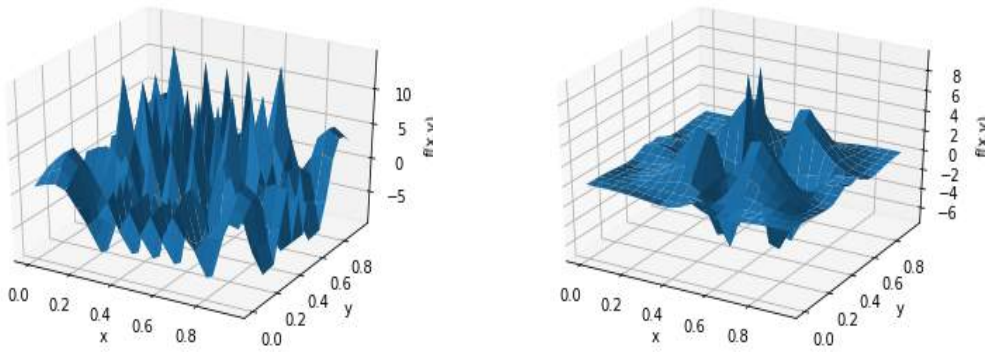
Figure 5.3: Two of the 100 near cubic simulated surfaces

5.3.2 Test bed 2

The second group of test bed functions is a scaled and centered version of the functions considered by Ba and Joseph, 2012 [16] and earlier by Xiong et al, 2007 [21], also referred to as XB functions. In brief, XB functions are smooth but have different behavior in the middle of the input range than near its edges. Hence they represent a significant challenge for stationary interpolation models. The members of this test bed have the form,

$$y(x) = C \prod_{i=1}^{d} \left\{ \sin\left(A_i (z_i - B_i)^4\right) \times \cos(2z_i - B_i) + ((z_i - B_i)/2) \right\} \qquad (5.6)$$

where $z_i = |x_i - 0.5|$ and $x_i \in [0, 1]^d$. The model coefficients are drawn independently. $\{A_i\}_{i=1}^d$ is drawn as i.i.d $U(20, 35)$ independently of $\{B_i\}_{i=1}^d$ drawn as i.i.d $U(0.5, 0.9)$. Here C was set equal to 10 for two dimensional input. Figure 5.4 shows some of the simulated surfaces of this test bed. From the plots ts visible that the surfaces behave differently at the centre than its edges, thus it is harder to predict such surfaces.

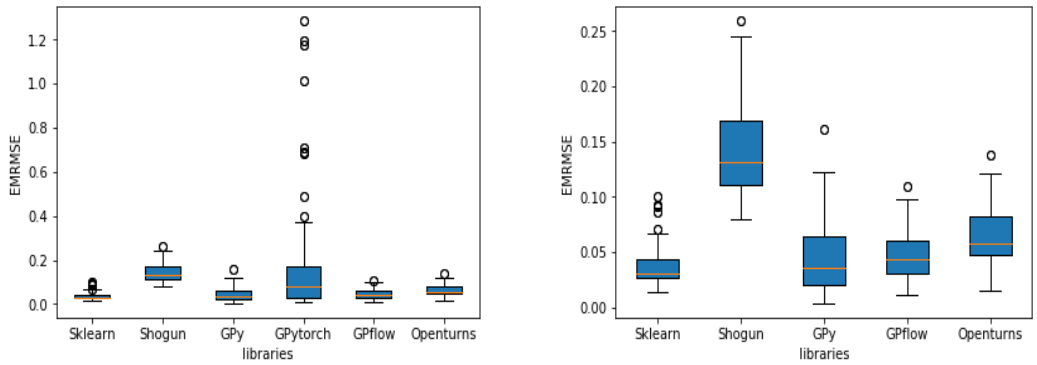


(a) (b)

Figure 5.4: Two of the 100 simulated XB surfaces

5.3.3 Test methodology :

Using the test beds given by eq.(5.5) and eq.(5.6) we generate 100 random near cubic surfaces and for each of the library we fit GP regression on the surfaces and observe the EMRMSE, PMRMSE and running time. The kernel used in this setup is the same **RBF** kernel with lengthscale and and kernel scale parameter both initialized to 1. The nugget parameter is set to 0.0001. Moreover in this case we draw the training and test sample using similar methodology in section 5.2.2 with $n_t = 100$ and $n_s = 1000$. Next we plot the EMRMSE, PMRMSE and running time for each library using boxplots as shown in figure 5.5 onwards.

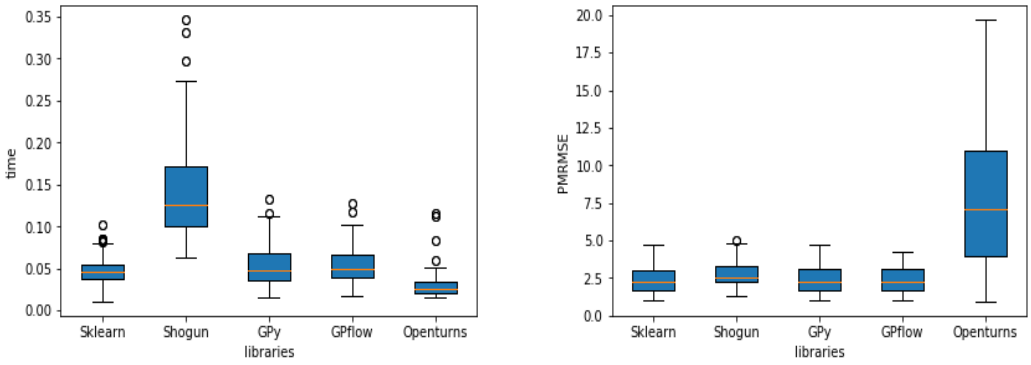


(a) all libraries (b) except GPytorch

Figure 5.5: Boxplots of EMRMSE for Test bed 1

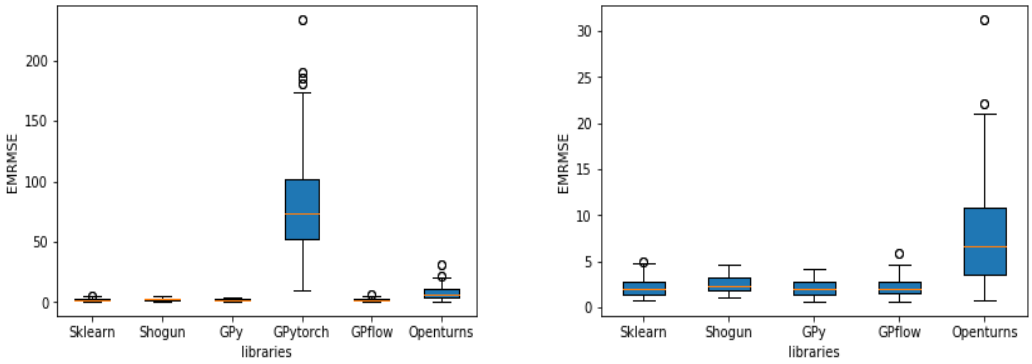
5.3.4 Observations

We observe interesting features of the different libraries from the box plots. Looking at figure 5.5 and 5.6 suggests that the performance of ***GPy*** is much better than other close competitors like ***Scikit-learn*** and ***GPflow***.
As far as the running time is concerned ***GPflow*** is taking much longer time



(a) Test bed 1 (b) Test bed 2

Figure 5.6: Boxplots of PMRMSE for Test bed 1 & 2



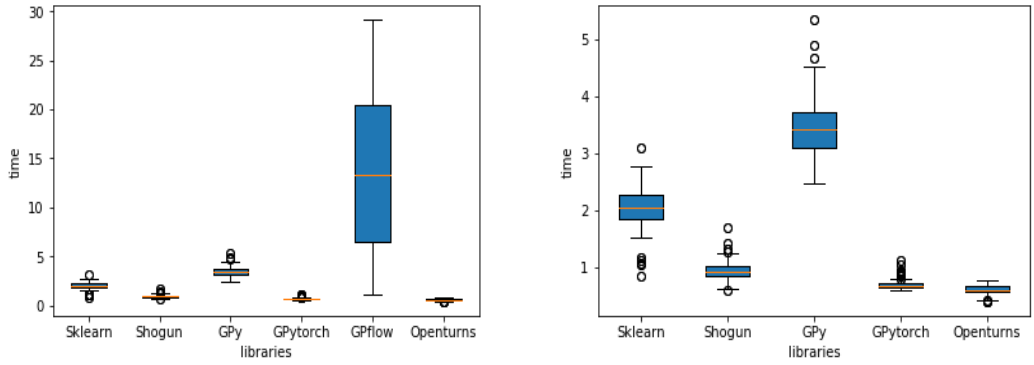
(a) all libraries (b) except GPytorch

Figure 5.7: Boxplots of EMRMSE for Test bed 2

than others in executing a single prediction fro a particular surface, followed by ***GPytorch*** which is a bit faster than it but the median time of execution is still more thn 3 seconds. Using a machine with a GPU, is thus recommended incase of ***GPflow***. On the other hand ***Openturns*** is the fastest followed by ***GPy***.

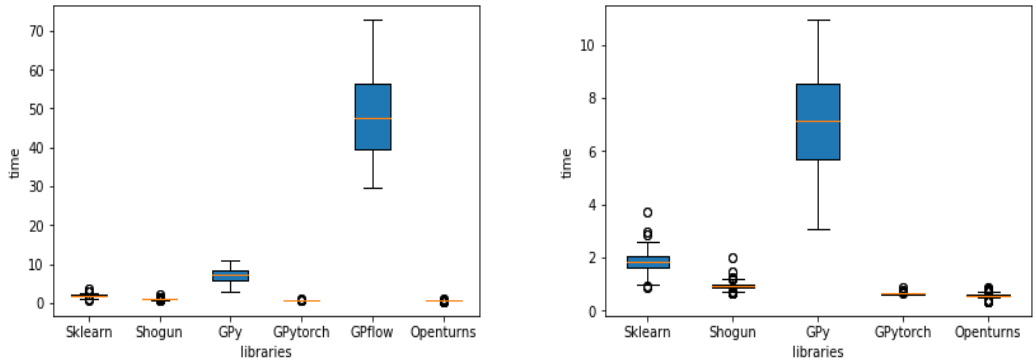
5.4 Tests on numerical stability

Numerical stability is one of the crucial factors for the libraries implementing Gaussian process regression. As mentioned in section 2.5.1 one of the practical limitations is expensive computation, typically on the order of $O(n^3)$ where n is the number of data points, in performing the necessary matrix inversions. For large datasets, storage and processing also lead to computational bottlenecks, and numerical stability of the estimates and predicted values degrades with increasing n. The algorithm (see Rasmussen & Williams, 2006.)[7] uses Cholesky decomposition, instead of directly inverting the covariance matrix, since it is faster and numerically more stable. But for typical data sets the Cholesky decomposition fails and leads to negative posterior variance in the output. The



(a) all libraries (b) except GPflow

Figure 5.8: Boxplots showing running time for Test bed 1



(a) all libraries (b) except GPflow

Figure 5.9: Boxplots showing running time for Test bed 2

following test is used to test such issues using the universal wrapper.

5.4.1 Negative variance test

This test compares the issue of negative posterior variances (which occurs due to numerical computational constraints) for different toolboxes. It also compares the prediction error of Gaussian Process regression with that of One Nearest Neighbour estimate and Two Nearest Neighbour estimate.

- **Description:**
  The test uses the following test function:

$$f(x) = \sin(2\pi x) \tag{5.7}$$

  The training points are generated using a sequence $x_n = \frac{(-1)^n}{n}$ converging to zero. The test iteratively increases the number of training points starting from n_{start} to n_{stop} (both the parameters n_{start} and n_{stop} are user inputs in the test function).
  In each iteration the test makes predictions on n_{test} test data points and computes the following:

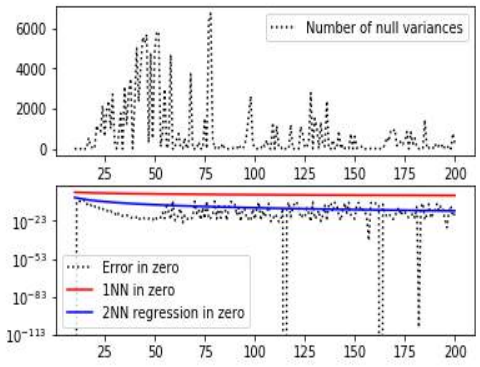
- ***Unusual variances***
  Given a user input ***eps*** the test computes the number of predicted posterior variances less than ***eps***.
- ***kriging Error***
  Since the sequence is converging to zero here we take the posterior predicted mean value itself as the kriging error. Square is taken to nullify the effect of positive and negative error.
- ***ONN Error***
  For this particular test function the One Nearest Neighbour estimate is nothing but the functional value of the previous training data point. Square is taken to nullify the effect of positive and negative error.
- ***2NN Error***
  The 2 nearest neighbours are the last 2 training data points and the estimate is their average. This estimate itself gives the estimate of the error. Square is taken to nullify the effect of positive and negative error.

- **Tests specifications** The test is performed for different python libraries with the floowing model specifications
  - **Covariance function** : Matern($\nu = 2.5$, lengthscale $= 1$)
  - **Parameter estimator** : Maximum likelihood
  - **Mean function** : Zero mean
  - **Test parameters** :
    - Noise level : 0
    - n_{start} : 10
    - n_{stop} : 200
    - n_{test} : 10001
    - $n_{restart}$: 10
    - **eps** : 0

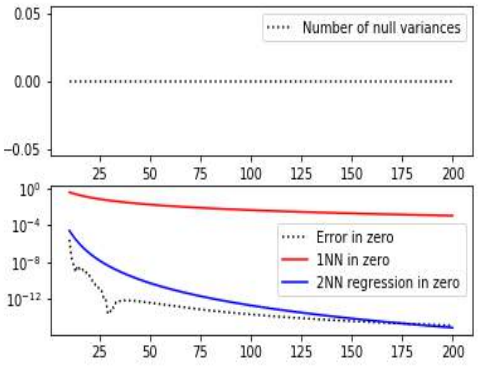
- **Results**

  Performing the test on various libraries we summarize the following results as shown in fig. 5.10.

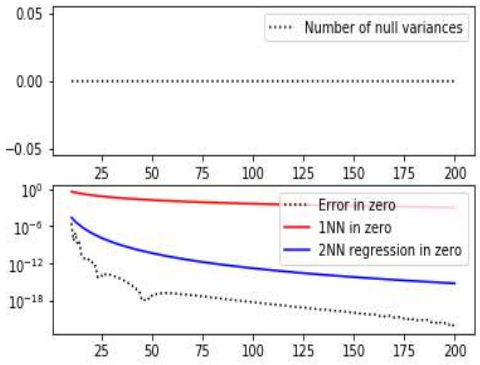
- **Conclusions:** The graphs clearly show that if the user defined nuggets is **zero**, for only ***Scikit-learn*** and ***Openturns*** the model is predicting negative variances. All the other libraries showed perfectly 0 predicted negative variances.
  For the comparison with ***One Nearest neighbour*** & ***Two Nearest neighbour*** estimate, ***GPy*** performs the best followed by ***Openturns***. ***GPflow*** aslo performs better than **1NN** and **2NN** at the beginning but as training points exceed more than near 160, GP regression's performance become poorer than **2NN**. For ***Scikit-learn*** GP fails to perform better than **2NN** and for ***GPytorch*** **2NN** performs much better than GP regression. In case of all the libraries **1NN** gives the worst prediction.



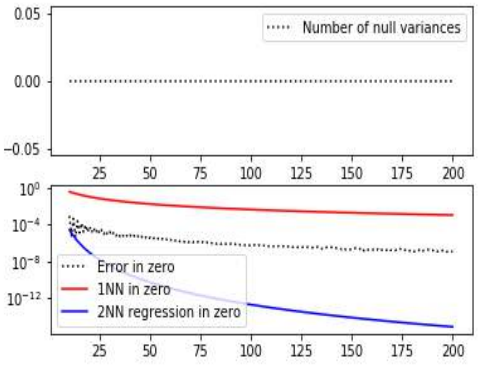
(a) Scikit-learn



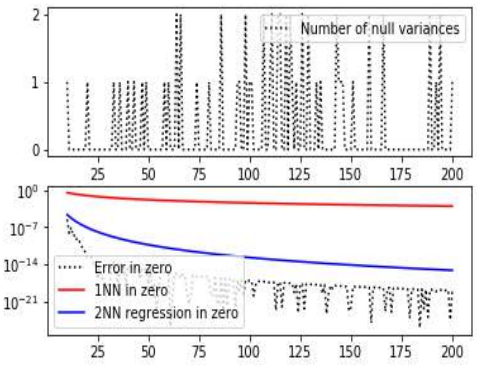
(b) GPflow



(c) GPy



(d) GPytorch



(e) Openturns

Figure 5.10: Results of negative variance test

Bibliography

[1] Abramowitz M. and Stegun I. A. *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical tables*. United States Department of Commerce, National Institute of Standards and Technology (NBS), (1985).

[2] Andrey Pepelyshev, H Dette. *Generalized Latin hypercube design for computer experiments.*Taylor Francis, (2010).

[3] Brockwell, Peter J., Davis, Richard A. *Time Series: Theory and Methods*, (1991).

[4] Collin B. Erickson, Bruce E. Ankenman, Susan M. Sanchez. *Comparison of Gaussian process modeling software*. European Journal of Operational Research, (2017).

[5] Cressie, N.A. *Statistics for spatial data*. John Wiley Sons, New York, (1993).

[6] C.A. McGilchrist. *Bias of ml and reml estimators in regression models with arma errors*, Journal of Statistical Computation and Simulation, Volume 32, 1989 - Issue 3 (1989).

[7] C. E. Rasmussen & C. K. I. Williams. *Gaussian Processes for Machine Learning, the MIT Press, 2006*, ISBN 026218253X. c 2006 Massachusetts Institute of Technology.[www.GaussianProcess.org/gpml].

[8] Hagan A.O. *Curve Fitting and Optimal Design for Prediction*. (1978)

[9] Hannes Nickisch, Carl Edward Rasmussen. *Approximations for Binary Gaussian Process Classification*. (2008).

[10] Hastie, T. J. et al. *The elements of statistical learning: data mining, inference, and prediction*. Springer, (2001).

[11] Michael L. Stein. *Interpolation of Spatial Data, Some Theory for Kriging*. Springer Series in Statistics, (1999).

[12] Patterson, H. D. and Thompson, R. *Recovery of interblock information when block sizes are unequal*. Biometrika58, 545-554, (1971).

[13] P. J. Diggle, J. A. Tawn and R. A. Moyeed. *Model-based geostatistics*. (1998).

[14] R. M. Neal. *Monte Carlo Implementation of Gaussian Process Models for Bayesian Regression and Classification.*Technical Report No. 9702, Dept of Statistics, University of Toronto, (1997).

[15] R. M. Neal. *Markov chain Monte Carlo method based on 'Slicing' the density function.* Technical Report No. 9722, Dept of Statistics, University of Toronto, (1997).

[16] Shan Ba and V. Roshan Joseph. *Composite Gaussian process models for emulating expensive functions*, (2012).

[17] Stone, M. *An asymptotic equivalence of choice of model cross-validation and Akaike's criterion.* J. R. Stat. Soc. B 36, 44–47 (1977)

[18] Thomas J. Santner, Brian J. Williams and William I. Notz *The Design and Analysis of Computer Experiments*, Springer Series in Statistics, (2003).

[19] Tunnicliffe-Wilson, G. *On the use of marginal likelihood in time series model estimation.* J. Roy. Statist. Soc. Ser. B 51, 15–27, (1988).

[20] Wilson A.G. , Nickisch H. *Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP)*

[21] Xiong Y, Chen W, Apley D, Ding X *A non-stationary covariance-based kriging method for metamodelling in engineering design.* Int J Numer Methods Eng 71(6):733–756, (2007).