A review of Python packages for Gaussian process regression

Subhasish Basak

supervisor

Emmanuel Vazquez

Laboratoire des Signaux et Systémes (L2S)
Université Paris Saclay

May-July 2019
Gif-sur-Yvette, France

Outline

Outline

Motivation & Goal

In recent years, Gaussian processes have gained popularity as a regression tool in the machine learning community, which has developed several Python packages (e.g. Scikit-Learn, GPy, GPyTorch etc), with different features and degrees of maturity. Being a non parametric Bayesian approach with a $O(n^3)$ computational complexity, the implementation of GP regression differs a lot among the libraries and hence their performance and results. Often practitioners using Gaussian processes are not familiar with the functionalities, disadvantages and benefits of the different libraries and they do not try experimenting with the advanced parameter settings, different optimization algorithms and hyperparameter tuning.
Considering similar studies (**C.B. Erickson et al. 2017**) in the literature, our report aims to present a comparative study of python libraries and their performances.

A comparative study

In this report we confine ourselves to a selected set of python libraries and our study consists of the following steps,

- **Identifying features** The primary step is to discriminate between the libraries on the basis of their features. Features can be anything like the availability of specific kernel and mean functions, available parameter estimation methods, compatible python version, prerequisite packages etc.
- **Building a Universal wrapper** 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.
- **Performing tests** Next we follow the scientific method of comparing software packages through simulated test functions. These tests are mainly concerned with the performance of the different libraries when they face extreme situations.

Outline

Gaussian Process

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

$$\xi(x) \sim \mathcal{GP}(m(x), K(x, x')) \tag{1}$$

GP Regression

Although Gaussian Process is used in both regression and classification problems, in our report we are only concerned with **Gaussian Process Regression**. This Bayesian approach is based on the following property of a Gaussian Process:

key property

Any finite collection of random variables from a Gaussian process not only has a multivariate normal distribution but also their conditional distributions are Gaussian.

GP Regression

The key property of a Gaussian process leads to the following lemma which constructs the base of the posterior prediction.

Lemma

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

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

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

$$(Y_1 | Y_2 = y_2) \sim N(\mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(y_2 - \mu_2), \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}) \qquad (2)$$

Regression Model

We consider a realistic modelling situation where we do not have access to functional values y_i's themselves, but with additional noise. With the Gaussian process $\xi(x)$ as defined in equation (1) (for sake of simplicity we have considered a **zero mean**, i.e. $m(x) = 0$) ,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:

$$y_i = \xi(x_i) + \epsilon_i \tag{3}$$

where ϵ_i is Gaussian noise with zero mean and variance σ_n^2. Now,given a set of n training data points $\{x_i, y_i\}_{i=1}^{n}$, we need to learn the latent function $\xi(x_i)$ and make predictions on the test points x_*.

Making predictions

The joint distribution of the target values, y , and the test outputs ξ_* according to the Gaussian process prior is,

$$\begin{bmatrix} y \\ \xi_* \end{bmatrix} \sim N\left(0, \begin{bmatrix} K(X,X) + \sigma_n^2 \mathbf{I} & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right) \tag{4}$$

and using the leamma (eq 2) we derive the conditional distribution of $\xi_*|X_*, X, y$ which is also Gaussian with the mean and covariance matrix as follows,

$$\begin{aligned} E[\xi_*|X_*, X, y] &= K(X_*, X)[K(X,X) + \sigma_n^2 \mathbf{I}]^{-1} y \\ Var[\xi_*|X_*, X, y] &= K(X_*, X_*) - K(X_*, X)[K(X,X) + \sigma_n^2 \mathbf{I}]^{-1} K(X, X_*)) \end{aligned} \tag{5}$$

which is the predicted posterior distribution obtained by Gaussian Process regression.

Outline

Mean and Covariance function

As mentioned earlier we need to specify the mean and covariance functions of a Gaussian process to perform the regression. Like other machine learning algorithms Gaussian process regression also learns its hyperparameters from the training observations using optimizing algorithms.
Some of the hyperparameter optimizing techniques are presented below, but most of the python libraries do not have all of them implemented.

Mean function

When the mean function is assumed to be zero and predictions can be made using eq 5. It is known as **Simple Kriging**. For other cases we take the mean function as a linear combination of known basis functions $\phi(x)$ as $\phi^\top(x)\beta$ with unknown coefficients β, which can be estimated using the following methods,

- **Kriging approach** : Kriging uses the generalized least square estimate of β yielding the GP predictor as the *Best Linear Unbiased Predictor* **(BLUP)**. For $\phi(x) = 1$ its called *Ordinary kriging* and for more generalized basis functions its called *Universal kriging*.
- **Maximum likelihood** : The parameters β can be optimized jointly with the hyperparameters of the covariance function.
- **Bayesian approach** : One can also assume priors on β and the obatain the posterior using eq 5.

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. Some commonly used kernels are Squared Exponential kernel(RBF), Matern kernel, polynomial kernel etc.
Although fully Bayesian approaches like Monte Carlo methods can be used to fit GP without estimating the kernel parameters but due to their high computational cost some popular hyperparameter estimation techniques are,

- **Maximum marginal likelihood**
- **Maximum a posteriori probability (MAP) estimate**
- **Restricted Maximum Likelihod (ReML) estimate**
- **Cross validation**

Outline

Features of python libraries

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, for e.g. Library release version, Compatible Python version, Library Dependencies etc. 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, for e.g. Available of Mean & Kernel modules, Nugget parameter setting, Available parameter estimation options etc.

Python libraries

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

Library	Version	Python version	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	n-d numpy array
GPytorch	0.3.2	3.6 and higher	tensors
Openturns	1.13	2.7,3.3 and higher	Openturns labels

Table: Python libraries

Among the libraries above **GPflow** has backend dependency on tensorflow and **GPytorch** has dependencies on torch, Pytorch.

Outline

Mean & kernel 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,

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

Table: available Kernel modules

Apart from this selected set of kernels there are also many other kernel modules implemented in these libraries viz. Cosine Kernel, Cylindrical Kernel, Periodic Kernel, Linear Kernel, Spectral Mixture Kernel, Exponential Sine Squared kernel etc.

Mean & kernel modules

All the libraries set a default **zero mean** and perform **simple kriging**. Except **Scikit-learn** all the other libraries have an option for a **Constant mean** with user defined value for the constant. Only **Openturns** supports **ordinary kriging** and it along with **GPflow** provide an option for **Linear Mean function** and for **Openturns** there is in-built quadratic basis functions also, with an extension to use polynomial basis functions.

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

Table: available Mean modules

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. The following 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: Nugget types :: 1: Homoscedastic; 2: Heteroscedastic

Scaling & Compounding kernels

Scaling refers to multiplying a scale factor to the covariance function and **Compounding** refers to combining 2 or more kernels into one by addition or multiplication. The following table describes about compound & scale kernels settings 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: Scaling, Compounding

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) etc.

Parameter Estimation

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: Parameter Estimation

Likelihoods & partial estimation

- **Likelihood :** Our main focus has been on regression with Gaussian noise, however, Gaussian processes can be used as priors with other likelihood functions.
  - Default Gaussian likelihood :
    Scikit-learn, **GPy**, **GPflow** and **Openturns**
  - No default likelihood :
    Shogun and **GPytorch**

  GPy, **Shogun** and **GPytorch** provide the option to choose other likelihoods viz. **Poisson**, **Softmax**, **Bernoulli**, **Student's t** etc. The choice of the likelihood depends on the data.
- **Partial estimation :**
  Sometimes the user may want to optimize some of the hyperparameters of the model instead of all. **Scikit-learn** and **Openturns** does not support optimization of the nugget parameter. For **GPy** there is an option to fix nugget parameter value and optimize the rest. No other library provide direct option for partial estimation of hyperparameters.

Outline

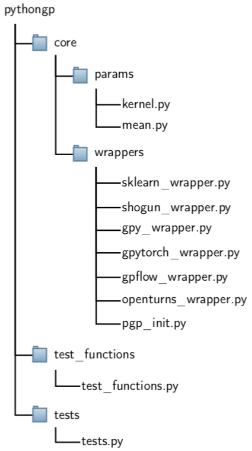
Need of a universal wrapper

In this section we describe the methodology of building the python wrapper which is compatible for all the selected set of libraries. Different libraries have specific dependencies and pre-requisites, 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. 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

Outline

File structure

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



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. **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.

Outline

Tasks

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. he followings are the tasks along with the methods that are implemented in the wrapper to execute them.

- Initialization
- Data loading & reshaping
- Specifying the mean function
- Constructing the kernel
- Building regression model
- Making predictions & plots

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** inside the **wrapper** subpackage. This module initiates the library wrapper to construct tests. There is a module of implemented tests in the module **tests.py** inside the subpackage **tests** . The user can import test functions along with their functional domains for using them in tests.

example code for initialization

```python
from pythongp.core.wrappers import pgp_init
from pythongp.tests.tests import test01
pgp = pgp_init.set_library("GPy")
test01(pgp)
```

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 inside each wrapper reshapes the data according to library configuration.

Specifying the mean function

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** which supports the following methods,

- **construct**: Constructs the **Mean** class by either taking user inputs.
- **get_mean_type**: 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**.

example code for mean function

```python
from pythongp.core.params import mean
m = mean.Mean()
m.construct('Zero')
pgp.set_mean (m)
```

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**: Constructs the kernel by taking user inputs about the kernel type and its parameter values.
- **set_bounds**: Specifies the lengthscale bounds of some of the kernels.
- **set_name** : Returns the kernel name.
- **show** : Returns the parameter values given as input.

For compounding there is a class named **CompoundKernel** which is an implementation of a **binary tree** structure which combines different kernels. The **add** and **mult** methods enables the user to combine elementary kernels using "**+**" or "*****".

Constructing mean and covariance function

example code for covariance function

```
from pythongp.core.params import kernel
kernel_dict_input_1 = {}
kernel_dict_input_1['Matern'] = {'lengthscale': 1, 'order':
    1.5, 'lengthscale_bounds': '(1e-5, 1e5)'}

kernel_dict_input_2 = {}
kernel_dict_input_2['RBF'] = {'lengthscale': 1, '
    lengthscale_bounds': '(1e-5, 1e5)'}

k_1 = kernel.Kernel()
k_1.construct(kernel_dict_input_1)

k_2 = kernel.Kernel()
k_2.construct(kernel_dict_input_2)

pgp.set_kernel (k_1+k_2)
```

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.

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.

Outline

Defining measures

Before beginning the tests we briefly introduce the measure which we have used. The material is drawn from Santner et al., 2003, p. 108.

- $$\text{EMRMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\xi(\mathbf{x}_i^*) - \xi(\hat{\mathbf{x}}_i^*))^2} \tag{6}$$

  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

- $$\text{PMRMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\hat{\sigma}_i^2} \tag{7}$$

  where σ_i^2's are the models predicted posterior variance for the test points. PMRMSE gives the predicted model's RMSE.

Defining measures

C.B. Erickson et al. 2018 suggested "Since EMRMSE and PMRMSE both measure the models RMSE, we expect them to be approximately equal".

- EMRMSE $\approx$ PMRMSE : accurate prediction error.
- EMRMSE $>$ PMRMSE : model is overconfident in its fit, since its estimated prediction errors will be smaller than the empirical errors.
- EMRMSE $<$ PMRMSE : models estimated prediction errors are conservative.

Outline

Test specification

The first test we perform is with a simple one dimensonal test function.

- **Test function**

$$f(x) = -\sum_{k=1}^{6} k \cos[(k+1)x + k] \tag{8}$$

- **Function domain :** $[-5, 5]$
- **Training points(n_t) :** 20
- **Testing points(n_s) :** 200
- **Mean function :** Zero mean
- **Kernel :** RBF(lengthscale = 1, scale = 1)
- **nugget :** 0.0001

Results

We present below the performance of the different libraries along with their estimated parameter values.

Library	Run time	EMRMSE	PMRMSE	lengthscale & scale	nugget
Sklearn	0.8763	4.4171	4.6444	0.257, 63.04	Unch
Shogun	9.9768	4.4177	4.6441	0.132,7.93	0.0002
GPy	2.4887	4.4176	4.6442	0.257,63.04	8.61e-09
GPflow	3.7210	4.4176	4.6440	0.257,63.03	1.02e-06
GPytorch	1.1903	6.2650	0.5091	0.538,1.60	9.99e-05
Openturns	1.6085	4.3751	4.6408	0.089,7.85	Unch

Table: Test results for One D test function

Observations

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 same input data, predictions can be different based on the specific library. Plots of prediction are as follows,

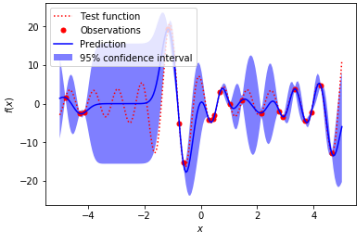


Figure: Scikit-learn

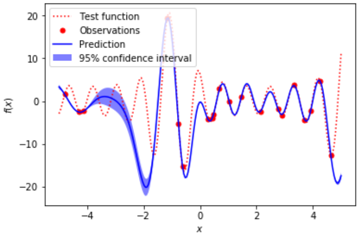


Figure: GPytorch

Outline

Test specification

Next we move on to higher dimensions with the Branin function.

- **Test function :** Branin function

$$f(x) = \left(x_1 - \frac{5.1x_0^2}{4\pi^2} + \frac{5x_0}{\pi} - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)cos(x_0) + 10 \quad (9)$$

- **Function domain :** $[-5, 10] \times [0, 15]$
- **Training points(n_t) :** 50
- **Testing points(n_s) :** 500
- **Mean function :** Zero mean
- **Kernel :** RBF(lengthscale = [1,1], scale = 1)
- **nugget :** 0.0001

Results with fixed parameters

We present below the results of the test when all the parameters are kept fixed.

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: Test results without optimization

Justification

All except **GPytorch** and **Shogun** performed differently. We summarize the following plausible causes for their behaviour.

- **GPytorch**
  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 are different from the other libraries.
- **Shogun**
  There is not option to prefix the nugget parameter and it is by default set to 1. Moreover we can not specify different lengthscale parameters for different dimensions in **Shogun**.

Results with optimized parameters

The results with optimized parameters are as follows,

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: Test results with optimized parameters

Observations

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. We summarize the following,

- **lowest running time :** Openturns
- **higest running time :** GPflow
- **lowest EMRMSE :** GPy
- **lowest PMRMSE :** GPy
- **lowest ratio(PMRMSE/EMRMSE) :** GPy

The **KISS-GP** approximation algorithm of **GPytorch** does not yield good prediction compared to other libraries and **Shogun** performs the worst.

Some contour plots of the Branin function fit

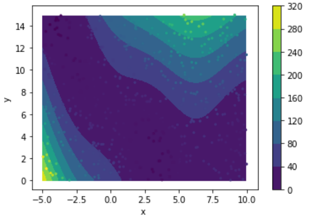


Figure: Scikit-learn

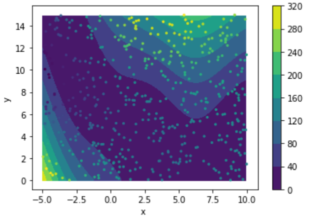


Figure: GPytorch

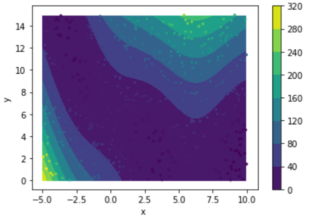


Figure: GPy

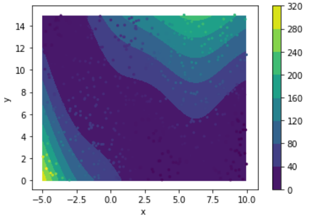


Figure: Openturns

Outline

Test methodology

In this section we use different test beds to generate 100 random surfaces and for each of the library we fit GP regression on the surfaces and observe the EMRMSE, PMRMSE and running time. The test specifications are as same as follows,

Training points(n_t) : 100

Testing points(n_s) : 1000

Mean function : Zero mean

Kernel : RBF(lengthscale = [1,1], scale = 1)

nugget : 0.0001

Test bed 1

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

$$y(x_1, x_2) = \frac{x_1^3}{3} - (R_1 + S_1)\frac{x_1^2}{2} + (R_1 S_1)x_1 + \frac{x_2^3}{3} - (R_2 + S_2)\frac{x_2^2}{2} + (R_2 S_2)x_2 + A \sin \frac{2\pi x_1 x_2}{Z} \quad (10)$$

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
$\sim U(0.25, 1.0)$.

Random simulated surfaces

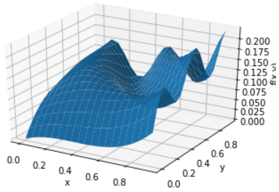


Figure: Test bed 1

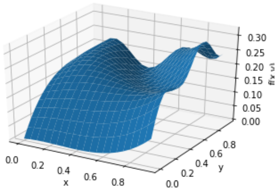


Figure: Test bed 1

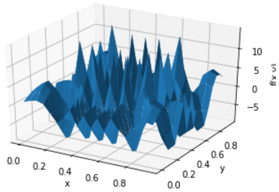


Figure: Test bed 2

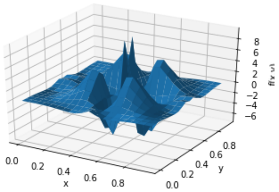


Figure: Test bed 2

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,

$$y(x) = C \prod_{i=1}^{d} \left\{ \sin\left(A_i(z_i - B_i)^4\right) \times \cos(2z_i - B_i) + ((z_i - B_i)/2) \right\} \quad (11)$$

Boxplots of accuracy measures

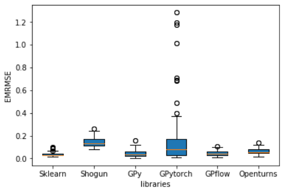


Figure: EMRMSE : Test bed 1

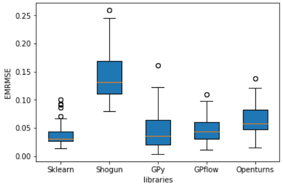


Figure: EMRMSE : Test bed 1 (without GPytorch)

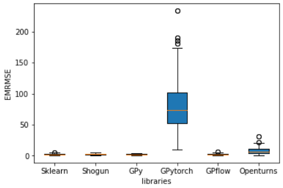


Figure: EMRMSE : Test bed 2

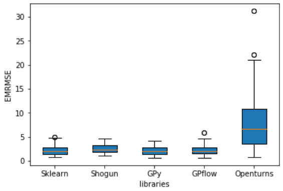


Figure: EMRMSE : Test bed 2 (without GPytorch)

Boxplots of accuracy measures

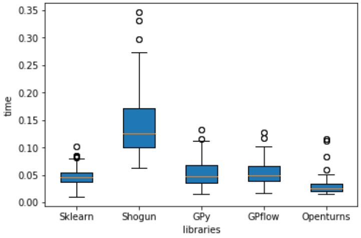


Figure: PMRMSE : Test bed 1

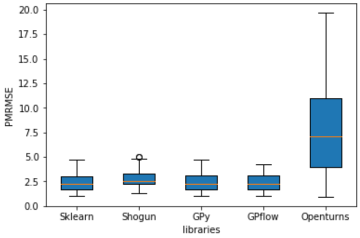


Figure: PMRMSE : Test bed 2

Boxplots of running time

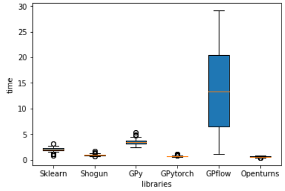


Figure: Test bed 1

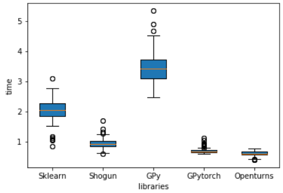


Figure: Test bed 1 (without GPflow)

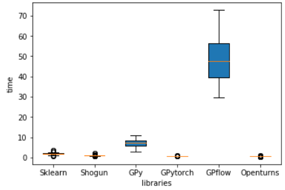


Figure: Test bed 2

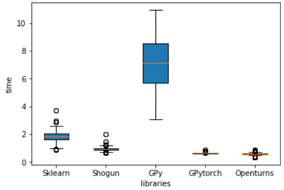


Figure: Test bed 2 (without GPflow)

Observations

We observe interesting features of the different libraries from the box plots. Looking at the boxplots 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 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**.

Outline

Numerical stability of python libraries

Numerical stability is one of the crucial factors for the libraries implementing Gaussian process regression. One of the practical limitations of GP regression 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 following test is used to test such issues using the universal wrapper.

Negative variance test

This test compares the issue of negative posterior variances for different toolboxes. It also compares the prediction error of GP regression with that of One Nearest Neighbour estimate and Two Nearest Neighbour estimate.

- **Test function :**

$$f(x) = \sin(2\pi x) \tag{12}$$

  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}

- **Test 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

Measures

The test records 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.

Result plots

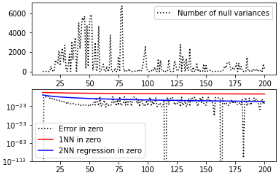


Figure: Scikit-learn

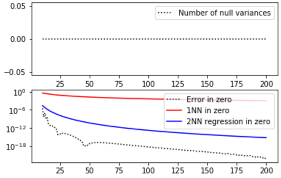


Figure: GPy

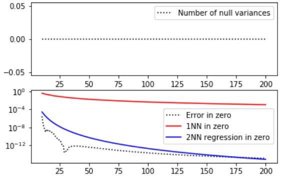


Figure: GPflow

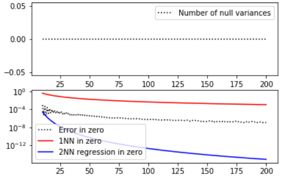


Figure: GPytorch

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.

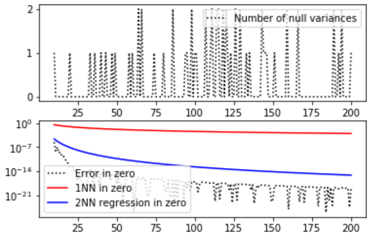


Figure: Openturns

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.

References

- Collin B. Erickson, Bruce E. Ankenman, Susan M. Sanchez. *Comparison of Gaussian process modeling software*. European Journal of Operational Research, (2017).
- 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].
- Michael L. Stein. *Interpolation of Spatial Data, Some Theory for Kriging*. Springer Series in Statistics, (1999).
- Thomas J. Santner, Brian J. Williams and William I. Notz *The Design and Analysis of Computer Experiments*, Springer Series in Statistics, (2003).

merci de votre attention