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 and simulation community. Some popular **python toolboxes implementing GP regression :**

- Scikit-Learn, GPy, GPyTorch, GPflow etc.

Need for a comparative study:

- Difference in implementation among the libraries
- Unawareness of available parameter settings & functionalities

Similar studies in the literarure:

- C.B. Erickson et al. (2017)

Outline

A quick recall of GP regression

A **Gaussian Process** is a Stochastic process, completely specified by its mean function $m(x)$ and covariance function $K(x, x')$ and for which any finite collection of r.v's have a Gaussian distribution.

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

We now consider the regression model, with a **zero mean** Gaussian Process $\xi(x)$ and Gaussian noise ϵ with variance σ_n^2,

$$y_i = \xi(x_i) + \epsilon_i \tag{2}$$

Now,given training data points $\{X, y\}$, using Bayes Rule the **predictive distribution** for test points X_* becomes,

$$\begin{aligned}\xi_*|X_*, X, y \sim N(&K(X_*, X)[K(X, X) + \sigma_n^2\mathbf{I}]^{-1}y \\ &, K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma_n^2\mathbf{I}]^{-1}K(X, X_*))\end{aligned} \tag{3}$$

Outline

Mean function

The mean function can be any of the following.

- **zero/known mean :** Simple Kriging (eq 3)
- **unknown mean :** $m(x) = \phi^{\top}(x)\beta$
  with known basis function $\phi(x)$ & unknown β

β can be estimated using the following methods,

- **Kriging approach** : Using 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** : Estimating β jointly with the hyperparameters of the covariance function.
- **Bayesian approach** : Assuming 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 define the closeness and similarity between data points.

- **Some commonly used kernels :** Squared Exponential kernel(RBF), Matern kernel, polynomial kernel etc.

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

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

Covariance functions

For our purpose we confined to a subset of **kernels** (covariance 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

Other kernel modules : Cosine Kernel, Cylindrical Kernel, Periodic Kernel, Linear Kernel, Spectral Mixture Kernel, Exponential Sine Squared kernel etc.

Mean modules

- All the libraries set a default **zero mean** (simple kriging)
- Except **Scikit-learn** all the other libraries have an option for a **Constant mean** with user defined constant
- **Openturns** & **GPflow** supports universal kriging

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
- **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

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. The following table 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.

Partial estimation :

- **Scikit-learn** and **Openturns** does not support optimization of the nugget parameter
- **GPy** has an option to fix nugget parameter value and optimize the rest

No other library provide direct option for partial estimation of hyperparameters.

Outline

Subpackages & modules

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

- **core**
  Contains furthur 2 subpackages viz.
  - **params :**
    contains 2 modules **kernel** and **mean**
  - **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.

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

Initialization & Data loading, reshaping

To initialize the library there is a wrapper module named **pgp_init.py**.

example code for library initialization

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

For data loading each wrapper has 2 methods implemented, which loads the training & test data and re-configures it according to the library requirements.

- **load_data**: Feeds the wrapper with train & test data.
- **load_mult_data**: Feeds the wrapper with multivariate train & test data.
- **dftoxz**: 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 & making predictons

Building regression model

- **init_model**: This method constructs the regression model by call. It takes the following argument,
  - **noise**: specifies the nugget parameter.
- **optimize**: This method optimizes the parameters of the model. The arguments are,
  - **param_opt**: sets whether to optimize the parameters or not.
  - **itr**: sets the number of iteration.

Making predictions

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

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.

$$\text{EMRMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\xi(\mathbf{x}_i^*) - \xi(\hat{\mathbf{x}}_i^*))^2} \quad \text{PMRMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\hat{\sigma}_i^2} \tag{4}$$

Where x_i^*'s are test points and $\xi(x_i^*)$, $\xi(\hat{x}_i^*)$ are observed and predicted values at those points respectively. σ_i^2's are the models predicted posterior variance for the test points.

- 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

Results with optimized parameters

- **Test function :** Branin function

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

The results with optimized parameters are as follows,

Library	Run time	EMRMSE	PMRMSE	Coeff. of determination	Corr. coeff.
Sklearn	0.7138	0.3349	0.1438	0.9999	0.9999
Shogun	1.0735	51.4738	48.5896	-0.0305	0.0035
GPy	3.6295	0.0221	nan	0.9999	0.9999
GPflow	1.2403	0.0236	0.0241	0.9999	0.9999
GPytorch	0.6011	12.7833	nan	0.9364	0.9681
Openturns	0.2297	2.0887	1.4570	0.9983	0.9992

Table: Test results with optimized parameters

Outline

Test beds

Test bed 1

$$y(x_1, x_2) = \frac{x_1^3}{3} - (R_1 + S_1)\frac{x_1^2}{2} + (R_1 S_1)x_1 + \frac{x_2^3}{3} - (R_2 + S_2)\frac{x_2^2}{2} + (R_2 S_2)x_2 + A\sin\frac{2\pi x_1 x_2}{Z} \quad (6)$$

Test bed 2

$$y(x) = C\prod_{i=1}^{d}\left\{\sin\left(A_i(z_i - B_i)^4\right) \times \cos(2z_i - B_i) + ((z_i - B_i)/2)\right\} \quad (7)$$

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

mean : Zero mean, **nugget :** 0.0001

$\mathbf{(}n_{train}\mathbf{,}n_{test}\mathbf{)}$ **:** 100,1000

Random simulated surfaces

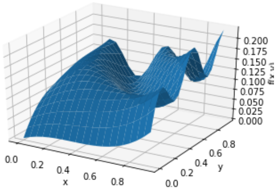

Figure: Test bed 1

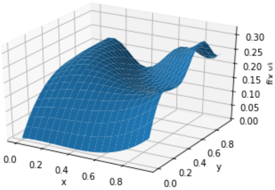

Figure: Test bed 1

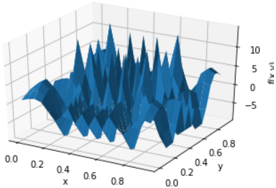

Figure: Test bed 2

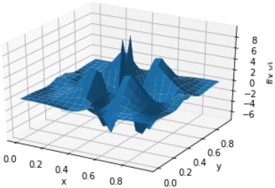

Figure: Test bed 2

Boxplots of accuracy measures for Test bed 1

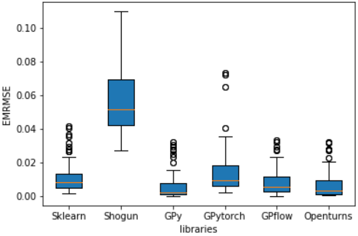

Figure: EMRMSE

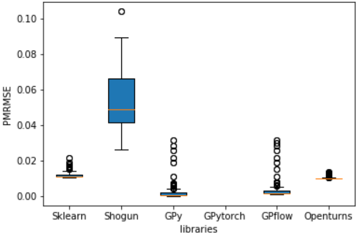

Figure: PMRMSE

Boxplots of accuracy measures for test bed 2

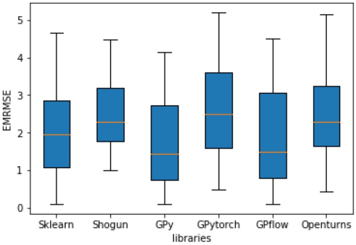

Figure: EMRMSE

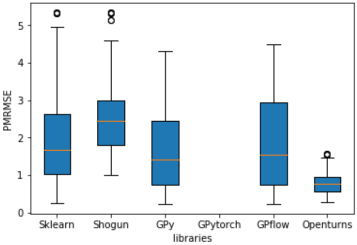

Figure: PMRMSE

Boxplots of running time

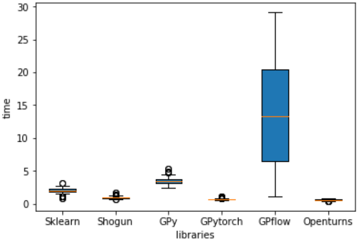

Figure: all libraries

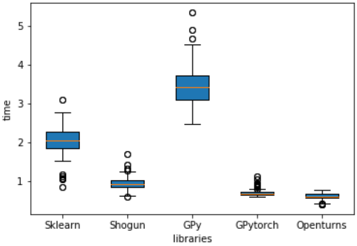

Figure: without GPflow

Figure: Test bed 1

Boxplots of running time

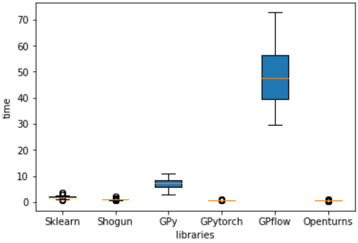

Figure: all libraries

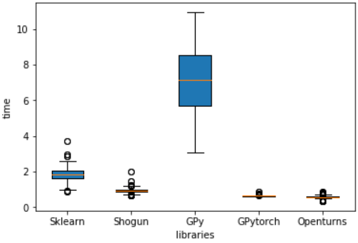

Figure: without GPflow

Figure: Test bed 2

Observations

Looking at the boxplots suggests the following,

- **GPy** is much better than its other close competitors like **Scikit-learn** and **GPflow**.
- **GPflow** is taking much longer time than others in executing a single prediction on a particular surface, followed by **GPytorch** which is a bit faster than it but the median time of execution is still more than 3 seconds.
- On the other hand **Openturns** is the fastest followed by **GPy**.

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