GPdoemd: Gaussian processes for design of experiments for model discrimination

Simon Olofsson
Department of Computing
Imperial College London
SW7 2AZ, United Kingdom
simon.olofsson15@imperial.ac.uk

Updated: 23 August 2018

Abstract

This document describes how to install and use the GPdoemd python package.

1 Introduction

The GPdoemd package contains functionality for performing design of experiments for model discrimination. The idea is to hybridise analytical and data-driven approaches to design of experiments to exploit useful features of both approaches, mainly computational efficiency and the ability to accommodate black-box models [Olofsson et al., 2018]. The aim has been to develop a plug-and-play python package. Our hope is that the package will be easy to extend and build upon. The GPdoemd uses functionality from the GPy [since 2012] package.

GPdoemd contains functionality for testing and comparing analytic, numerical and GP surrogate methods for design of experiments for model discrimination.

GPdoemd contains a set of analytic and non-analytical case studies. Most case studies are taken from literature, and a few have been constructed in the process of developing this package.

2 License

The GPdoemd package is released under the MIT License. Details can be found in the file LICENSE included in the package, as well as online.

3 Installation

The GPdoemd package has been tested and validated on OSX and Ubuntu. No guarantees are provided that GPdoemd works on Windows-based systems.

3.1 Prerequisites

Python 3.4+.

Required python packages are:

- numpy >= 1.7
- scipy >= 0.17
- GPy

GPy (at the time of writing this) requires the packages six and paramz >= 0.9.0.

3.2 Creating a virtual environment

We recommend installing GPdoemd in a virtual environment. To set up a new virtual environment called myenv (example name), run the command

```
python3 -m venv myenv
```

in the folder where you want to store the virtual environment. After the virtual environment has been created, activate it as follows

```
source myenv/bin/activate
```

It is recommended that you upgrade the pip installation before proceeding

```
pip install --upgrade pip
```

3.3 Installing GPdoemd

To install GPdoemd, first install all required packages. They are listed in the file requirements.txt.

```
pip install numpy scipy
pip install GPy
```

Then run the following to install GPdoemd

```
pip install git+https://github.com/cog-imperial/GPdoemd
```

It is also possible to clone into the GPdoemd git repository and install it using setup.py, but this is not recommended for most users.

3.4 Uninstalling GPdoemd

The GPdoemd package can be uninstalled by running

```
pip uninstall GPdoemd
```

Alternatively, the folder containing the virtual environment can be deleted. This will remove the entire virtual environment in one go. There is no need to uninstall all packages in the virtual environment prior to deleting it.

4 Using `GPdoemd`

4.1 Initialising a surrogate model

A surrogate model is initialised by calling surrogate model class with an input dictionary `d`. For example, to initialise a regular GP surrogate model, we run the following

```
from GPdoemd.models import GPmodel
M = GPModel(d)
```

The different model classes differ in the way they compute/approximate the gradients of the model function with respect to the model parameters. The available model classes are

Model class	Gradients
`Analytic`	$\partial f/\partial \boldsymbol{\theta}$ provided by user through model function
`Numerical`	Finite difference approximation of $\partial f/\partial \boldsymbol{\theta}$
`GPModel`	$\partial \mu/\partial \boldsymbol{\theta}$, $\partial^2 \mu/\partial \boldsymbol{\theta}^2$, $\partial \Sigma/\partial \boldsymbol{\theta}$, $\partial^2 \Sigma/\partial \boldsymbol{\theta}^2$
`SparseGPModel`	Same as `GPModel`

The model classes `GPModel` and `SparseGPModel` are subclasses of `SurrogateModel`.

The dictionary `d` has the following mandatory and optional fields:

Field	Type	Default	Description
`name`	`str`	'model'	Model name
`call`	callable	-	Function handle $f(\boldsymbol{x}, \boldsymbol{\theta})$
`dim_x`	`int`	-	Number of design variables
`dim_p`	`int`	-	Number of model parameters
`num_outputs`	`int`	1	Number of target dimensions
`p_bounds`	`list`, `numpy.ndarray`	[]	Number of target dimensions
`meas_noise_var`	`float`, `numpy.ndarray`	1	Measurement noise (co)variance
`binary_variables`	`list`	[]	Binary design variable dimensions

See the file `models/model.py` for more information. A field missing a default value is mandatory. It is recommended to set the non-mandatory fields as well, in order to avoid any later problems. In addition, the model classes `GPModel` and `SparseGPModel` have the following optional field:

Field	Type	Default	Description
`gp_noise_var`	`float`	10^{-6}	Fixed GP noise variance hyperparameter value

It is recommended to leave this value with its default value, unless you run into numerical problems.

4.2 Performing parameter estimation

Given experimental data `Xdata` and `Ydata`, we can tune the model parameters $\boldsymbol{\theta}$ to make the model predictions $f(\boldsymbol{x}, \boldsymbol{\theta})$ fit the data as well as possible. Best-fit model parameter values are required for each model in order to design the next experiment. The best-fit model parameter values, in the form of a 1D numpy array `pstar` of length `dim_p` can be given directly to the surrogate model `M` as

```
M.pmean = pstar
```

Alternatively, one of the two built-in parameter estimation methods (differential evolution `diff_evol` or least squares `least_squares`) can be used in one of the following ways

```
from GPdoemd.param_estim import diff_evol, least_squares
M.pmean = diff_evol(M, Xdata, Ydata, M.p_bounds)          # Alternative 1
M.pmean = least_squares(M, Xdata, Ydata, M.p_bounds)      # Alternative 2
M.param_estim(Xdata, Ydata, diff_evol, M.p_bounds)        # Alternative 3
M.param_estim(Xdata, Ydata, least_squares, M.p_bounds)    # Alternative 4
```

The parameter estimation methods are wrappers for the differential evolution and least squares methods found in `scipy`. We would recommend using `least_squares` for models with more expensive function evaluations, such as solving systems of ODEs, and `diff_evol` for models with cheaper function evaluations.

4.3 Setting GP surrogate model training data

We assume a GP surrogate model `M` has been initialised, and that we are given training data `Y` and `Z`

```
Y = np.array([ M.call(x,p) for x,p in zip(X,P) ])
Z = np.c_[ X, P ]
```

for locations `X` (`n` $\times$ `dim_x` numpy array) and model parameter values `P` (`n` $\times$ `dim_p` numpy array). The training data is then fed to the GP surrogate model using **one** of the following commands

```
M.set_training_data(Z, Y)       # Alternative 1
M.set_training_data(X, P, Y)    # Alternative 2
M.Z, M.Y = Z, Y                 # Alternative 3
```

4.4 Setting GP surrogate model kernel functions

The kernel function of the GP prior is assumed to be multiplicative in the form $k(\{\boldsymbol{x}, \boldsymbol{\theta}\}, \{\boldsymbol{x}', \boldsymbol{\theta}'\}) = k_x(\boldsymbol{x}, \boldsymbol{x}')k_\theta(\boldsymbol{\theta}, \boldsymbol{\theta}')$. This assumption enables us to easily calculate $\partial k(\cdot, \cdot)/\partial \boldsymbol{\theta}$. The `GPdoemd` package has extended the kernel functions of the `GPy` package by including second derivatives with respect to the input of some standard stationary kernel functions. All kernel functions assume automatic relevance detection (ARD), such that the distance measure $r(\boldsymbol{z}, \boldsymbol{z}') = \sqrt{(\boldsymbol{z} - \boldsymbol{z}')^\top \boldsymbol{\Lambda}^{-1} (\boldsymbol{z} - \boldsymbol{z}')}$, where $\boldsymbol{\Lambda}$ is a diagonal matrix of squared length scales.

To set the kernel functions used in the surrogate model, e.g. the RBF kernel, the following lines are used

```
from GPdoemd.kernels import RBF
M.kern_x = RBF
M.kern_p = RBF
```

Other kernels available are `Exponential`, `Matern32`, `Matern52`, `Cosine` and `RatQuad`.

4.5 Constructing GP surrogates

When the training data `Z` and `Y`, and kernels `kern_x` and `kern_p`, have been given, the GP surrogate model must be constructed. A separate GP model is constructed for (i) each target dimension and (ii) binary variable combination. If there are 5 target dimensions, and 3 binary variables, a total of 5×2^3 GP models are constructed.

Constructing the GP models for the surrogate model `M` is done as follows

```
M.gp_surrogate()
```

Alternatively, setting the training data and kernels can be done at the same time as the GP surrogate is constructed

```
M.gp_surrogate(Z=Z, Y=Y, kern_x=RBF, kern_p=RBF)
```

If any of `Z`, `Y`, `kern_x` or `kern_p` have already been set, they do not need to be included in the `gp_surrogate` call.

4.6 Learning GP surrogate model hyperparameters

In order to train the hyperparameters for the GP models of surrogate model `M`, we make one of the following calls

```
M.gp_optimise()     # British spelling
M.gp_optimize()     # American spelling
```

The function `gp_optimize` calls `gp_optimise`. Optional function call inputs are:

- `index`, an `int` or list of `int`s with index/indices for the target dimension(s) for which to (re)train the GP model hyperparameters. Default is to train all GP models.
- `max_lengthscale`, the maximum lengthscale allowed for any design or model parameter dimension, given that all training data have been normalised to the space [0,1]. Default value is 10.

4.7 Approximating the model parameter distribution

The model parameter distribution $p(\boldsymbol{\theta} \mid \boldsymbol{X}_{\text{data}}, \boldsymbol{Y}_{\text{data}}) \approx \mathcal{N}(\boldsymbol{\theta}^*, \boldsymbol{\Sigma}_\theta)$ is approximated with a Gaussian distribution around the best-fit model parameter values $\boldsymbol{\theta}^*$. In order to compute the model parameter covariance $\boldsymbol{\Sigma}_\theta$, we use a Laplacian approximation

```
from GPdoemd.param_covar import laplace_approximation
M.Sigma = laplace_approximation(M, Xdata)
```

4.8 Computing approximate marginal predictive distribution

We can get the approximate marginal predictive distributions at locations `xnew` (`nn` $\times$ `dim_x` numpy array) by calling

```
from GPdoemd.marginal import taylor_first_order
mu, S = taylor_first_order( M, xnew )
```

which produces the mean `mu` (`nn` $\times$ `num_outputs` numpy array) and covariance `S` (`nn` $\times$ `num_outputs` $\times$ `num_outputs` numpy array). An alternative to the first-order Taylor approximation is the second-order Taylor approximation `taylor_second_order`.

4.9 Computing design criterion

In order to find the next optimal experiment, we maximise some design utility function, also known as a design criterion. We begin by computing the means `mu` and covariances `s2` of the marginal predictive distributions for all models `Ms` = $\{$`M`$_i\}$ at all test points `xnew`

```
import numpy as np
n  = len(xnew)
m  = len(Ms)
E  = Ms[0].num_outputs
mu = np.zeros(( n, m, E ))
s2 = np.zeros(( n, m, E, E ))
for i, M in enumerate( Ms ):
    mu[:, i], s2[:, i] = taylor_first_order( M, xnew )
    # Alternatively, for a potentially more accurate mean
    mu[:, i] = np.array([ M.call( x, M.pmean) for x in Xdata ])
```

We can then compute the design criterion at all design test points `xnew`, and find the test point `xnext` with the highest score

```
from GPdoemd.design_criteria import JR
measvar = Ms[0].meas_noise_var          # Measurement noise covariance
pps     = [ 1. / m for _ in range(m) ]  # Prior probabilities
dc      = JR( mu, s2, measvar, pps )    # Compute design criterion
xnext   = xnew[ np.argmax( dc ) ]       # Find next design
```

There are multiple design criteria implemented in the `GPdoemd` package:

Design criterion	Reference	Requires	
		`measvar`	`pps`
`HR`	Hunter and Reiner [1965]		
`BH`	Box and Hill [1967]	✓	✓
`BF`	Buzzi-Ferraris et al. [1990]	✓	
`AW`	Michalik et al. [2010]	✓	✓
`JR`	-	✓	✓

If `pps` is not set, it will be assumed that the prior probability is equal for all models.

4.10 Discriminating between models

When a new experiment experiment has been performed, and the result incorporated into the data set `Xdata`, `Ydata`, we can attempt model discrimination. First, the model parameters should be re-tuned.

```
for M in Ms:
    M.param_estim( Xdata, Ydata, diff_evol, M.p_bounds )
```

We may want to generate new training data `Z`, `Y` for the GP surrogates models, and retrain them.

```
for M in Ms:
    M.gp_surrogate( Z=Z, Y=Y )
    M.gp_optimise()
    M.Sigma = laplace_approximation( M, Xdata )
```

Then we want to compute the marginal predictive distributions for the design in our experimental data set.

```
mu = np.zeros(( n, m, E ))
s2 = np.zeros(( n, m, E, E ))
for i, M in enumerate( Ms ):
    mu[:, i], s2[:, i] = taylor_first_order( M, Xdata )
    # Alternatively, for a potentially more accurate mean
    mu[:, i] = np.array([ M.call( x, M.pmean) for x in Xdata ])
```

The last thing we need is an array `D = np.array([ M.dim_p for M in Ms ])` with the number of model parameters for each model. Now we are ready to determine which models adequately describe the experimental data, or which model fits the data best.

We will use the following mathematical notation when defining the discrimination criteria

Symbol	Python	Notes
$\boldsymbol{X}$	`Xdata`	$\boldsymbol{X}^\top = [\boldsymbol{x}_1^\top, \ldots, \boldsymbol{x}_N^\top]$
$\boldsymbol{Y}$	`Ydata`	$\boldsymbol{Y}^\top = [\boldsymbol{y}_1^\top, \ldots, \boldsymbol{y}_N^\top]$
$\breve{\mu}_i(\boldsymbol{x}_j)$	`mu[j-1,i-1]`	Marginal predictive mean for model i and design $\boldsymbol{x}_j$
$\breve{\Sigma}_i(\boldsymbol{x}_j)$	`s2[j-1,i-1]`	Marginal predictive covariance for model i and design $\boldsymbol{x}_j$
$\boldsymbol{\Sigma}$	`measvar`	Measurement noise covariance
E	`E`, `M.num_outputs`	Number of target dimensions
D_i	`Ms[i-1].dim_p`	Number of model parameters for model i

4.10.1 Gaussian likelihood

Box and Hill [1967] suggests using the (normalised) Gaussian likelihoods π_i for model selection. The Gaussian likelihood for model i is defined as

$$\pi_i = \frac{p(\boldsymbol{Y} \mid \boldsymbol{X}, \breve{\mu}_i, \breve{\Sigma}_i)}{\sum_{j=1}^{m} p(\boldsymbol{Y} \mid \boldsymbol{X}, \breve{\mu}_j, \breve{\Sigma}_j)}, \quad \text{with } p(\boldsymbol{Y} \mid \boldsymbol{X}, \breve{\mu}_i, \breve{\Sigma}_i) = \prod_{n=1}^{N} \mathcal{N}(\boldsymbol{y}_n \mid \breve{\mu}_i(\boldsymbol{x}_n), \breve{\Sigma}_i(\boldsymbol{x}_n) + \boldsymbol{\Sigma}).$$

Clearly, $\sum_i \pi_i = 1$. In order to compute the the Gaussian likelihoods `pps`, we call

```
from GPdoemd.discrimination_criteria import gaussian_likelihood
s2 += measvar
pps = gaussian_likelihood(Ydata, mu, s2)
```

We can set a threshold ξ (e.g. $\xi = 10^{-8}$), where a model i is discarded if $\pi_i < \xi$.

If we have computed the Gaussian likelihoods $\pi_i = \pi_{i,N}$ for the first N data points, Box and Hill [1967] suggests updating the Gaussian likelihood after each additional data point in the following way

$$\pi_{i,N+1} = \frac{\pi_{i,N} \mathcal{N}(\boldsymbol{y}_{N+1} \mid \breve{\mu}_i(\boldsymbol{x}_{N+1}), \breve{\Sigma}_i(\boldsymbol{x}_{N+1}) + \boldsymbol{\Sigma})}{\sum_{j=1}^{m} \pi_{j,N} \mathcal{N}(\boldsymbol{y}_{N+1} \mid \breve{\mu}_j(\boldsymbol{x}_{N+1}), \breve{\Sigma}_j(\boldsymbol{x}_{N+1}) + \boldsymbol{\Sigma})}.$$

This is done by calling

```
from GPdoemd.discrimination_criteria import gaussian_likelihood_update
s2_new += measvar
pps_new = gaussian_likelihood_update(y_new, mu_new, s2_new, pps)
```

where `mu_new` and `s2_new` are the mean and covariances for the models at the latest design point. Note that the value of the updated Gaussian likelihood $\pi_{i,\tilde{N}}$ will depend on the order in which the $\tilde{N}$ data points are observed.

4.10.2 χ^2 test

Buzzi-Ferraris et al. [1990] advocates using the χ^2 test for model discrimination. This does not rank the models against each other, but models are discarded when they are no longer deemed adequate to describe the observed data. The χ^2 test metric q_i for model i is defined as

$$q_i = 1 - \int_0^{\epsilon} F_{k_i}^2(\boldsymbol{z}) \mathrm{d}z$$

where F_k^2 is the cumulative χ^2 distribution with $k = N \times E - D_i$ degrees of freedom, and ϵ is the sum of weighted errors given by

$$\epsilon = \sum_{n=1}^{N} (\breve{\mu}_i(\boldsymbol{x}_n) - \boldsymbol{y}_n)^\top \breve{\Sigma}_i^{-1}(\boldsymbol{x}_n)(\breve{\mu}_i(\boldsymbol{x}_n) - \boldsymbol{y}_n).$$

We set a threshold ξ (e.g. $\xi = 0.01$). If $q_i < \xi$, then model i is said to have failed the χ^2 test. However, [Buzzi-Ferraris et al., 1990] argues against discarding the model completely, but simply ignore it when designing the next experiment.

In `GPdoemd` we can compute the models χ^2 test scores q_i by calling

```
from GPdoemd.discrimination_criteria import chi2
s2 += measvar
qs  = chi2( Y, mu, s2, D )
```

4.10.3 Akaike information criterion

Michalik et al. [2010] argues in favour of using the Akaike information criterion for model selection. This is similar to the Gaussian likelihoods, but adds a penalty term for the number of tunable model parameters, such that

$$\tilde{p}(\boldsymbol{Y} \mid \boldsymbol{X}, \breve{\mu}_i, \breve{\Sigma}_i) = \exp\left(2 \log p(\boldsymbol{Y} \mid \boldsymbol{X}, \breve{\mu}_i, \breve{\Sigma}_i) - 2D_i\right).$$

The probability terms $\tilde{\pi}_i$ are then computed in the same way as π_i, but using $\tilde{p}(\cdot)$.

References

G. E. P. Box and W. J. Hill. Discrimination among mechanistic models. *Technometrics*, 9(1):57–71, 1967.

G. Buzzi-Ferraris, P. Forzatti, and P. Canu. An improved version of a sequential design criterion for discriminating among rival multiresponse models. *Chem Eng Sci*, 45(2):477–481, 1990.

GPy. GPy: A Gaussian process framework in python. `http://github.com/SheffieldML/GPy`, since 2012.

W. G. Hunter and A. M. Reiner. Designs for discriminating between two rival models. *Technometrics*, 7(3):307–323, 1965.

C. Michalik, M. Stuckert, and W. Marquardt. Optimal experimental design for discriminating numerous model candidates: The AWDC criterion. *Ind Eng Chem Res*, 49:913–919, 2010.

S. Olofsson, M. P. Deisenroth, and R. Misener. Design of experiments for model discrimination hybridising analytical and data-driven approaches. In *ICML '18: Proceedings of the International Conference on Machine Learning*, Stockholm, Sweden, PMLR 80, 2018.