Journal of Machine Learning Research 16 (2015) 2611-2616 Submitted 2/14; Revised 12/14; Published 12/15

pyGPs – A Python Library for Gaussian Process Regression and Classification

Marion Neumann M.NEUMANN@WUSTL.EDU
Department of Computer Science and Engineering,
Washington University, St. Louis, MO 63130, United States

Shan Huang SCHAN.HUANG@GMAIL.COM
Fraunhofer IAIS, 53757 Sankt Augustin, Germany

Daniel E. Marthaler DAN.MARTHALER@GMAIL.COM
Sproutling, San Francisco, CA 94111, United States

Kristian Kersting KRISTIAN.KERSTING@CS.TU-DORTMUND.DE
Department of Computer Science, TU Dortmund University
44221 Dortmund, Germany

Editor: Antti Honkela

Abstract

We introduce pyGPs, an object-oriented implementation of Gaussian processes (GPs) for machine learning. The library provides a wide range of functionalities reaching from simple GP specification via mean and covariance and GP inference to more complex implementations of hyperparameter optimization, sparse approximations, and graph based learning. Using Python we focus on usability for both "users" and "researchers". Our main goal is to offer a *user-friendly and flexible* implementation of GPs for machine learning.

Keywords: Gaussian processes, Python, regression and classification

1. Introduction

pyGPs is a Python software project implementing Gaussian processes (GPs) for machine learning (ML). GPs have become a popular model for a wide variety of ML tasks (Rasmussen and Williams, 2006), such as standard regression and classification, as well as active learning (Freytag et al., 2013), graph-based and relational learning (Chu et al., 2006), and Bayesian optimization (Osborne et al., 2009). Besides the recent advances in ML research, GPs get more and more attention for applications in other fields such as animal behaviour research (Mann et al., 2011) or reconfigurable computing (Kurek et al., 2013). Existing procedural GP libraries are GPML (Rasmussen and Nickisch, 2010) and GPstuff (Vanhatalo et al., 2013). However, depending on their design procedural implementations can be hard to extend. Being an established object-oriented programming language Python has great support and is easy to use. There are a few existing Python implementations of GPs. GPs in scikit (Pedregosa et al., 2011) provide only very restricted functionality and they are difficult to extend. pyGP[1] is little developed in terms of documentation and developer interface. GPy (the GPy authors, 2014) was developed in parallel to pyGPs and the library focuses

1. Online at `https://github.com/PMBio/pygp`.

mainly on dimensionality reduction and multi-output learning, whereas our implementation provides extensions for graph-based learning including an implementation of propagation kernels (Neumann et al., 2012), as well as simple routines for multi-class classification, evaluation, and enhanced hyperparameter optimization.

pyGPs is both *user-friendly and flexible*. We explicitly want to bridge the gap between systems designed primarily for "users", who mainly want to apply GPs and need basic ML routines for model training, evaluation, and visualization, and expressive systems for "developers", who focus on extending the core GP functionalities as covariance and likelihood functions, as well as inference techniques. We provide a comprehensive and illustrative documentation including a lot of demos and an overview of functionalities providing an easy start with pyGPs. Further, we believe that utilizing object-oriented programming is the right direction towards our goal of developing user-friendly *and* flexible software.

2. Implementation and Documentation

pyGPs is released under the FreeBSD license and it can be downloaded from `http://mloss.org/software/view/509/` or `https://github.com/marionmari/pyGPs`. pyGPs requires Python 2.6 or 2.7 (`www.python.org`) and the `numpy` (`www.numpy.org`), `scipy` (`www.scipy.org`), and `Matplotlib` (`www.matplotlib.org/`) packages. The provided functionality follows roughly the GPML toolbox introduced in Rasmussen and Nickisch (2010), which is implemented in a procedural way in MATLAB. However, pyGPs has an object-oriented structure and it additionally supports useful routines for the practical use of GPs, such as cross validation functionalities for evaluation as well as basic routines for iterative restarts for GP hyperparameter optimization. The library also supports FITC sparse approximations (Snelson and Ghahramani, 2005), one-vs-one multi-class classification and kernels for graph-based and semi-supervised learning.[2]

pyGPs provides a comprehensive documentation in form of a pdf-manual including an API and an online documentation at `http://www-ai.cs.uni-dortmund.de/weblab/static/api_docs/pyGPs/`. This documentation guides the user through installation, offers a small tutorial on GPs, summarizes the functionalities of the library and walks the user through a lot of demos. There are demo implementations of basic and sparse regression, as well as of basic and sparse binary classification. Further, we show how to do multi-class classification in a one-vs-one fashion, how to perform k-fold cross validation and how to incorporate kernels on graphs and graph kernels. The documentation also gives instructions on how to develop customized kernel, mean, likelihood, or inference functions. pyGPs also includes unit tests and instructions on how to test newly developed functions.

3. Functionalities of pyGPs

Now we exemplify the use of pyGPs for regression and describe its functionalities in detail.

2. We also released the procedural version `pyGP_PR`, which consists of a subset of pyGPs routines and is intended for users familiar with the GPML toolbox. It provides all basic routines needed to follow the examples in Rasmussen and Williams (2006). Online at `https://github.com/marionmari/pyGP_PR`.

3.1 Basic Example

Given the training data (x, y), where $x \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^n$, we get the predictions $f_* = f(z)$ for test inputs $z \in \mathbb{R}^{m \times d}$ by invoking the following four lines:

```
1 model = pyGPs.GPR()      # specify model (GP regression)
2 model.getPosterior(x,y) # get default model (zero mean & rbf kernel)
3 model.optimize(x,y)      # optimize hyperparams (single run minimize)
4 model.predict(z)         # prediction for test cases
```

Besides the predictive mean $\bar{f}_*$ (`model.ym`) of the GP which is commonly used as point estimate for the input targets, the model contains the predictive variance (`model.ys2`) and the means and variances of the latent function (`model.fm` and `model.fs2`).

In the following, we give a more detailed description of the above routine. By specifying the model as GP regression, cf. line 1, we assume a prior GP $f \sim \mathcal{GP}\left(m(x), k(x, x')\right)$, where the default mean function is zero, $m(x) = 0$, and the default covariance is a radial basis function (RBF) kernel, $k(x, x') = \sigma^2 \exp(-\frac{\|x-x'\|^2}{2\ell^2})$, with hyperparameters $\theta = \{\sigma, \ell\}$; both of which have a default value of 1. Further, the default GP regression settings are a Gaussian likelihood function and exact inference. For hyperparameter optimization we use an optimizer introduced in Rasmussen (1996) commonly referred to as *minimize* as the default. We will describe and explain the use of non-default likelihoods, and inference and optimization methods in the next section. Non-default means such as a linear (`mean.Linear`) mean function and covariances such as polynomial (`cov.Poly`) or Matérn (`cov.Matern`) or sums (`+`) and products (`*`) thereof can be set by using `model.setPrior`. A list of implemented means and kernels is provided in Table 1.
The following lines show how to set composite mean and covariance functions:

```
5 m = pyGPs.mean.Linear(D=x.shape[1])+pyGPs.mean.Const()  # sum of means
6 k = pyGPs.cov.RBF() * pyGPs.cov.Linear()           # product of kernels
7 model.setPrior(mean=m, kernel=k)                   # non-default prior
```

After we have specified the GP for regression, we can fit the model to our training data, cf. line 2. Now, we get the current value of the negative log marginal likelihood (`model.nlZ`) and its partial derivatives w.r.t. each hyperparameter (`model.dnlZ`) and the (approximate) posterior (`model.posterior`) represented by $L = \text{cholesky}(K + \sigma_n^2 I)$ (`posterior.L`), $\alpha = L^\top \backslash (L \backslash y)$ (`posterior.alpha`) and σ_n (`posterior.sW`). So far, we performed inference with the default hyperparameters of the specified covariance function. For better results, however, we optimize the hyperparameters, cf. line 3. This means that we minimize the negative log marginal likelihood $-log\ p(y|x, \theta) = -\frac{1}{2}\, y^\top K^{-1} y - \frac{1}{2}\, log|K| - \frac{n}{2}\, log 2\pi$ and fit the model again with the learned hyperparameters. The hyperparameters can be accessed via `model.covfunc.hyp` and the posterior (`model.posterior`) and negative log marginal likelihood (`model.nlZ`) will be updated accordingly. Now, we can get the predictions with the optimal hyperparameters, cf. line 4, where $\bar{f}_*$ is the expected value of $f_*|x, y, z$ (`model.ym`) and $V(f_*)$ is the variance of $f_*|x, y, z$ (`model.ys2`).

3.2 Functionalities

The object-oriented implementation offers one base class `GP` for the general GP model and five base classes for the core GP functionality `Mean`, `Kernel`, `Likelihood`, `Inference`, and `Optimizer`. Tables 1 and 2 show lists of implemented functionalities in pyGPs. Due to

kernels	*kernels for graphs*	*means*	*optimization methods*	*evaluation measures*
CONSTANT	DIFFUSION	CONSTANT	MINIMIZE	ACC
LINEAR (ISO, ARD, ONE)	L+	LINEAR	BFGS	RMSE
RBF (ISO, ISO-UNIT, ARD)	REG LAPLACIAN	ONE	CG	PREC
MATERN (ISO, ARD)	RANDOM WALK	ZERO	SCG	RECALL
RQ (ISO, ARD)	VND			NLPD
PERIODIC	INVERSE COSINE			
POLYNOMIAL	PROPAGATION KERNEL			
PIECWISEPOLY (ISO)				
NOISE				
COMPOSITE: SUM (+), PRODUCT (∗), SCALE (∗)				

Table 1: pyGPs functionality: kernels, means, optimizers, evaluation measures

	likelihood		
inference	GAUSSIAN	LAPLACE	ERROR FUNCTION
EXACT	✓		
LAPLACE	✓		✓
EP	✓	✓	✓
FITC-EXACT	✓		
FITC-LAPLACE	✓		✓
FITC-EP	✓	✓	✓

Table 2: pyGPs functionality: inference methods, likelihoods

the intuitive class hierarchy it is easy to augment the classes by for instance customized covariance functions and likelihoods. This makes pyGPs suitable for researches in ML. Further, we provide functionalities to ease usability of GPs as a machine learning tool as for instance parameter optimization, evaluation, and one-vs-one multi-class classification. They are explained by detailed demos (`demo_GPMC.py`, `demo_Validation.py`) and in the documentation. In the following, we briefly describe the most important aspects of pyGPs.

Sparse Approximations. We support sparse approximations for large scale GPs for regression and classification. We implement the popular "fully independent training conditional" (FITC) approximation (Snelson and Ghahramani, 2005) for exact and approximate inference.

Optimizers. Beside minimize, other optimization methods included in pyGPs are scaled conjugate gradient optimization (SCG) and it is also possible to use built-in optimizers from `scipy` such as conjugate gradient (CG) or the quasi-Newton method BFGS.

Validation. We provide the most common technique for model evaluation, k-fold cross validation (`valid.py`). The implemented evaluation measures are root mean squared error (`RMSE`), accuracy (`ACC`), precision and recall (`Prec`, `Recall`) and the negative log predictive density (`NLPD`) to evaluate the quality of the whole predictive GP model.

GraphExtensions. pyGPs offers the possibility to perform GP inference on networked data. So far, we provide one example graph kernel (propagation kernel (Neumann et al., 2012)), kernels for graph-based and semi-supervised learning, and knn-graph creation.

Currently, we are working on time series modeling and Bayesian optimization with GPs, as well as the incorporation of more state-of-the-art graph kernels for structured data.We also plan to add multi-output GPs, active learning, further application support, and more likelihood and covariance functions in the near future.

Acknowledgments

We would like to thank the following persons for their help in improving this software: Roman Garnett, Maciej Kurek, Hannes Nickisch, Zhao Xu, and Alejandro Molina. This software project is partly supported by the Fraunhofer ATTRACT fellowship STREAM.

References

W. Chu, V. Sindhwani, Z. Ghahramani, and S.S. Keerthi. Relational Learning with Gaussian Processes. In *Advances in Neural Information Processing Systems (NIPS-06)*, pages 289–296. 2006.

A. Freytag, E. Rodner, P. Bodesheim, and J. Denzler. Labeling examples that matter: Relevance-based active learning with gaussian processes. In *Proceedings of the 35th German Conference on Pattern Recognition (GCPR)*, volume 8142 of *Lecture Notes in Computer Science*, pages 282–291. Springer, 2013.

M. Kurek, T. Becker, and W. Luk. Parametric Optimization of Reconfigurable Designs Using Machine Learning. In *Reconfigurable Computing: Architectures, Tools and Applications - 9th International Symposium (ARC-2013)*, pages 134–145, 2013.

R. Mann, R. Freeman, M. A. Osborne, R. Garnett, C. Armstrong, J. Meade, D. Biro, T. Guilford, and S. Roberts. Objectively identifying landmark use and predicting flight trajectories of the homing pigeon using Gaussian processes. *Journal of the Royal Society Interface*, 8(55):210–219, 2011.

M. Neumann, N. Patricia, R. Garnett, and K. Kersting. Efficient Graph Kernels by Randomization. In *Proceedings of the Machine Learning and Knowledge Discovery in Databases - European Conference (ECML/PKDD-12)*, pages 378–393, 2012.

M. A. Osborne, R. Garnett, and S. J. Roberts. Gaussian processes for global optimization. In *Proceedings of the 3rd Learning and Intelligent Optimization Conference (LION-09)*, 2009.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

C. E. Rasmussen. Function minimization using conjugate gradients: Conj, 1996.

C. E. Rasmussen and H. Nickisch. Gaussian Processes for Machine Learning (GPML) Toolbox. *Journal of Machine Learning Research*, 11:3011–3015, 2010.

C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

E. Snelson and Z. Ghahramani. Sparse Gaussian Processes using Pseudo-inputs. In *Advances in Neural Information Processing Systems (NIPS-05)*, pages 1257–1264, 2005.

the GPy authors. GPy: A Gaussian process framework in python, 2014. `https://github.com/SheffieldML/GPy`.

J. Vanhatalo, J. Riihimäki, J. Hartikainen, P. Jylänki, V. Tolvanen, and A. Vehtari. Gpstuff: Bayesian modeling with gaussian processes. *Journal of Machine Learning Research*, 14 (1):1175–1179, 2013.