pyGPs API

Release v1.2

M. Neumann, S. Huang, D. Marthaler, K. Kersting

July 02, 2014

CONTENTS

CHAPTER

ONE

PYGPS - A PACKAGE FOR GAUSSIAN PROCESSES

1.1 About the package

pyGPs is a library hosting Python implementations of Gaussian processes (GPs) for machine learning. pyGPs bridges the gap between systems designed primarily for users, who mainly want to apply gps and need basic machine learning routines for model training, evaluation, and visualiza- gp functiontion, and expressive systems for developers, who focus on extending the core functionalities as covariance and likelihood functions, as well as inference techniques.

The software package is released under the BSD 2-Clause (FreeBSD) License. `Copyright` (c) by Marion Neumann, Shan Huang, Daniel Marthaler, & Kristian Kersting, Feb.2014

Further, it includes implementations of

- minimize.py implemented in python by Roland Memisevic 2008, following minimize.m (Copyright (c) Carl Edward Rasmussen (1999-2006))
- scg.py (Copyright (c) Ian T Nabney (1996-2001))
- brentmin.py (Copyright (c) Hannes Nickisch 2010-01-10)
- FITC functionality (following matlab implementations under Copyright (c) by Ed Snelson, Carl Edward Rasmussen and Hannes Nickisch, 2011-11-02)

This is a stable release. If you observe problems or bugs, please let us know. You can also download a procedual implementation of GP functionality from Github. However, the procedual version will not be supported in future.

Authors:

- Marion Neumann [marion dot neumann at uni-bonn dot de]
- Shan Huang [shan dot huang at iais dot fraunhofer dot de]
- Daniel Marthaler [marthaler at ge dot com]
- Kristian Kersting [kristian dot kersting at cs dot tu-dortmund dot de]

The following persons helped to improve this software: Roman Garnett, Maciej Kurek, Hannes Nickisch, Zhao Xu, and Alejandro Molina.

This work is partly supported by the Fraunhofer ATTRACT fellowship STREAM.

1.2 Getting started

1.2.1 Installation

1. First, download the archive from github and extract it to any local directory.
2. You can either add the local directory to your PYTHONPATH

   ```
   export PYTHONPATH=$PYTHONPATH:/path/to/local/directory/../parent_folder_of_pyGPs
   ```

3. Or install the package using setup.py:

   ```
   sudo python setup.py install
   ```

Requirements

- python 2.6 or 2.7
- scipy, numpy, and matplotlib: open-source packages for scientific computing in Python.

Example installation on Ubuntu & Debian:

```
sudo apt-get install python2.7 python-numpy python-scipy python-matplotlib
```

Example installation on Mac via Macports (requires XCode and MacPorts):

```
sudo port install python27 py27-numpy py27-scipy py27-matplotlib
```

For other systems please check the installation instructions on the respective package web sites.

1.2.2 GPs & Functionality

Gaussian Processes (GPs) can conveniently be used for Bayesian supervised learning, such as regression and classification. In its simplest form, GP inference can be implemented in a few lines of code. However, in practice, things typically get a little more complicated: you might want to use expressive covariance and mean functions, learn good values for hyperparameters, use non-Gaussian likelihood functions (rendering exact inference intractable), use approximate inference algorithms, or combinations of many or all of the above.

A comprehensive introduction to Gaussian Processes for Machine Learning is provided in the GPML book by Rasmussen and Williams, 2006.

1.3 Tutorials

1.3.1 Demos

There are several demos exemplifying the use of pyGPs for various Gaussian process (GP) tasks. We recommend to first go through *Basic GP Regression* which introduces the GP regression model. Basic regression is the most intuitive and simplest learning task feasable with GPs. The other demos will then provide a general insight into more advanced functionalities of the package. You will also find the implementation of the demos in the source folder under pyGPs/Demo.

The Demos give some theoretical explanations. Further, it is useful to have a look at our documentation on Kernels & Means and Optimizers.

Regression

Basic Regression

The code shown in this tutorial can be executed by running *pyGPs/Demo/demo_GPR.py*

This demo will not only introduce the regression model, it also provides the general insight of how to use the package. This general information will not be repeated in the other demos.

Import packages

Once you installed pyGPs, the typical way to import it is:

```
from pyGPs.Core import *
import numpy as np
```

Load data

First, load the data for this demo. The data consists of $n = 20$ 1-d data points drawn from a unit Gaussian. This is the same data used in the GPML example (it is hardcoded in *data/regression_data.npz*).

```
demoData = np.load('data_for_demo/regression_data.npz')
x = demoData['x']      # training data
y = demoData['y']      # training target
z = demoData['xstar']  # test data
```

A five-line toy example

Now lets do regression with Gaussian processes. Using pyGPs for regression is really simple; here is the most basic example:

```
model = gp.GPR()     # specify model (GP regression)
model.fit(x, y)      # fit default model (mean zero & rbf kernel) with data
model.train(x, y)    # optimize hyperparamters (default optimizer: single run minimize)
model.predict(z)     # predict test cases
model.plot()         # and plot result
```

By default, GPR uses a zero mean, the rbf kernel and a Gaussian likelihood. Default optimizer is a single run of Rasmussen's minimize. You will see below how to set non-default values in another example.

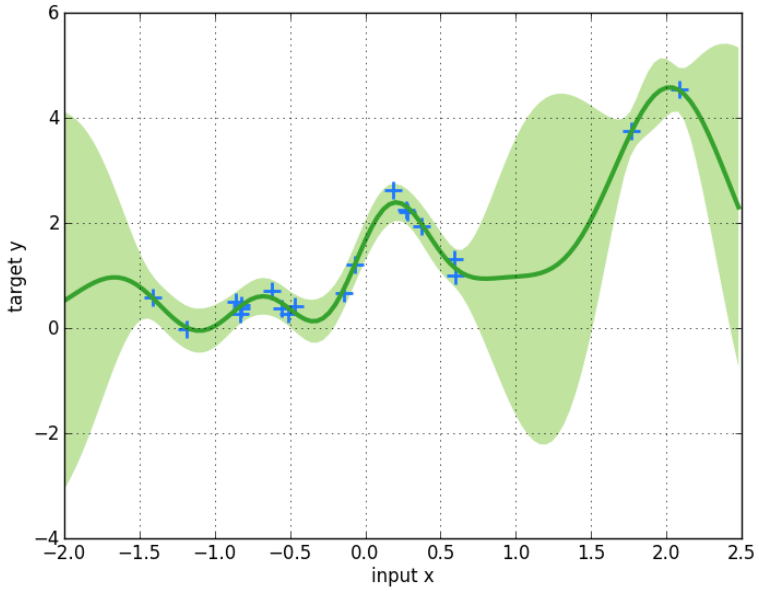
GPR.plot() will plot the result, where the dark line is the posterior mean and the green-shaded area is the posterior variance. Note, that *plot()* is not a general method as it is not trivial to visualize high dimensional data. Here, *GPR.plot()* works for 1-d data only, while *GPC.plot()* is a toy method visualising 2-d input data in a classification scenario.

A more complicated example

Now lets do another example to get insight into more advanced features of the toolbox.

You can specify non-default mean and covariance functions:

```
m = mean.Linear( D=x.shape[1] ) + mean.Const()
k = cov.RBF()
model.setPrior(mean=m, kernel=k)
```



Here, we use a composite mean as the sum of a linear and a constant function, and an rbf kernel. The initial hyperparameters are left to their default values. See Kernels & Means for a complete documentation of kernel/mean specification and custom kernel/mean construction. Once kernel and mean are specified, they are passed to the prior using *setPrior()*.

You can add the traning data to the model explicitly by using *setData()*. So, you avoid passing them into *fit()* or *train()* each time used. More importantly, the deafult mean will be adapted to the average value of the trainging labels y (if you do not specify mean function by your own).

Further, you can plot the data in the 1-d case:

```
model.setData(x, y)
model.plotData_1d()
```

You can specify a optimization method different from the default, which is a single run of Rasmussen's minimize. For example, you can choose to rerun the optimization method several times with different random initializations:

```
model.setOptimizer("Minimize", num_restarts=30)
```

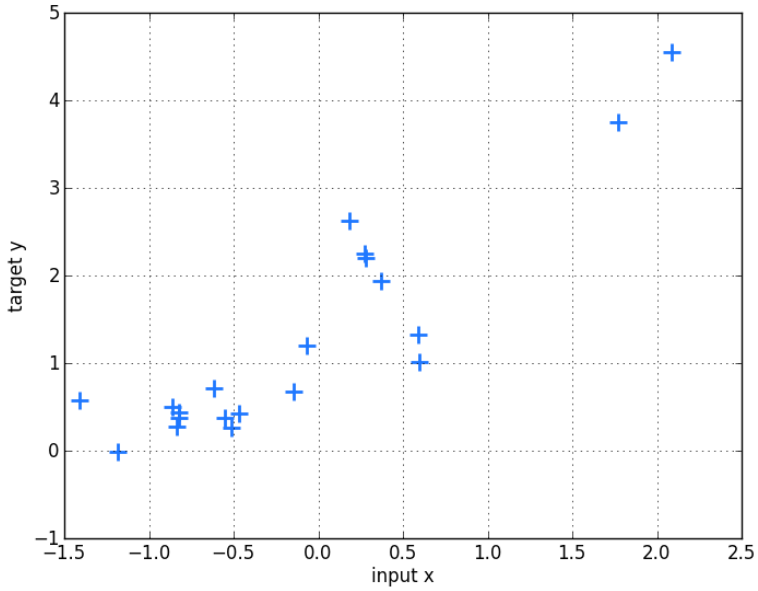
The optimized hyperparameters returned by *train()* are then set to be the ones obtained from the run with the best result. The whole functionality for optimization is introduced in detail in the documentation Optimizers.

Instead of *fit()*, which only fits data using given hyperparameters, *train()* will optimize hyperparamters based on marginal likelihood:

```
model.train()
```

There are several properties you can get from the model:

```
model.nlZ                   # negative log marginal likelihood
model.dnlZ.cov              # direvatives of negative log marginal likelihood
model.dnlZ.lik
model.dnlZ.mean
model.posterior.sW          # posterior structure
model.posterior.alpha
model.posterior.L
```




```
model.covfunc.hyp
model.meanfunc.hyp
model.likfunc.hyp
model.fm                    # latent mean
model.fs2                   # latent variance
model.ym                    # predictive mean
model.ys2                   # predictive variance
model.lp                    # log predictive probability
```

For example, to get the log marginal likelihood use:

```
print 'Optimized negative log marginal likelihood:', round(model.nlZ,3)
```

Prediction on the test data will return five values, which are output mean (ymu) resp. variance (ys2), latent mean (fmu) resp. variance (fs2), and log predictive probabilities (lp)

```
ym, ys2, fm, fs2, lp = model.predict(z)
```

Plot data. Note that *GPR.plot()* is a toy method only for visualising 1-d data. Here we got a different posterior by using a different prior other than in the default example.

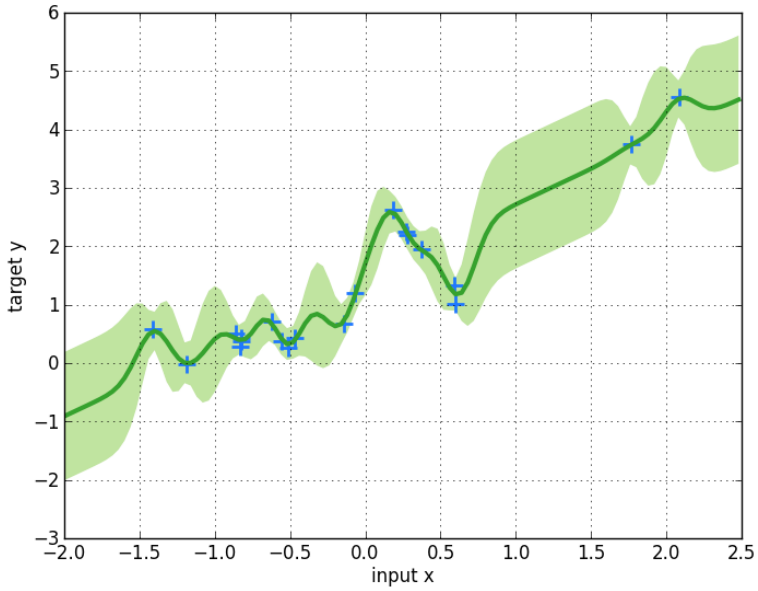
```
model.plot()
```

A bit more things you can do

[For all Models] Speed up computation time for prediction if you know posterior in advance. Posterior is passed as an object with three fields (attributes) post.alpha, post.sW and post.L. How to use these vectors to represent the posterior can be best seen from Algorithm 2.1 (page 19) in Chapeter 2 of the GPML book by Rasmussen and Williams, 2006.

```
post = myPosterior()        # known in advance
ym, ys2, fm, fs2, lp = model.predict_with_posterior( post,z )
```

[Only for Regression] Specify noise of data (with $\sigma = 0.1$ by default):




```
model.setNoise( log_sigma = np.log(0.1) )
```

You do not need to specify the noise parameter if you are optimizing the hyperparamters later anyhow.

All plotting methods have keyword axisvals. You can adjust plotting range if you want. For example:

```
model.plot(axisvals = [-1.9, 1.9, -0.9, 3.9])
```

Switch to other Inference and Likelihood functions.

```
model.useInference("EP")
model.useLikelihood("Laplace")
```

Sparse Regression

The code shown in this tutorial can be obtained by running *pyGPs/Demo/demo_GPR_FITC.py* This demo is more or less similar to the demo of FITC classification.

First example → default inducing points

First load the same data as in the GPR demo.

[Theory] In case the number of training inputs x exceeds a few hundred, approximate inference using Laplace approximation or expectation propagation takes too long. We offer the FITC approximation based on a low-rank plus diagonal approximation to the exact covariance to deal with these cases. The general idea is to use inducing points u and to base the computations on cross-covariances between training, test and inducing points only.

Okay, now the model is FITC regression:

```
model = gp.GPR_FITC()
```

The difference between the usage of basic GP regression is that we will have to specify inducing points. In the first example here, we will introduce you how to use the default settings.

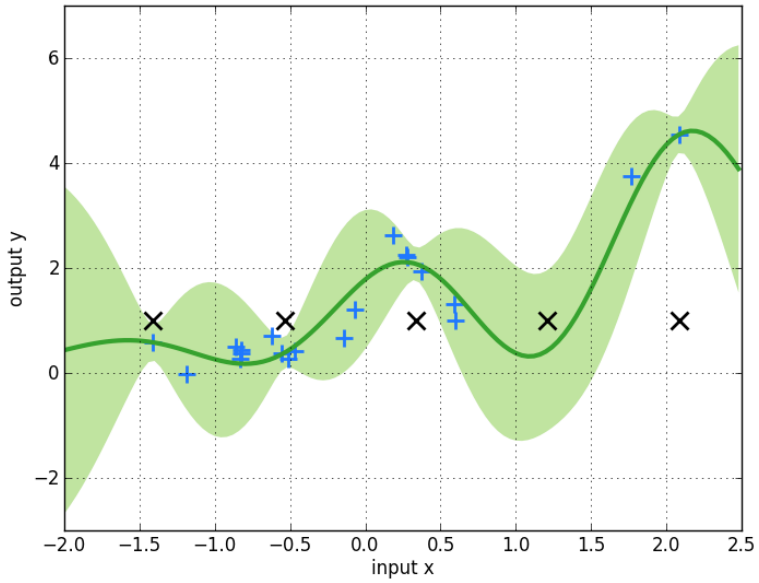
The default inducing points are a grid (hypercube for higher dimensions), where each dimension has 5 values in equidistant steps in $[min, max]$, where min and max are the minimum and maximum values of the input data by default. In order to specify the dimension of input data, we HAVE TO set data first:

```
model.setData(x, y)
```

The number of inducing points per axis is 5 per default.

Now, the regular training and prediction routines follow:

```
model.train()
model.predict(z)
model.plot()
```



The equidistant default inducing points u that are shown in the figure as black x's.

To change the number of inducing points per axis just specify a different value per axis:

```
model.setData(x, y, value_per_axis=10)
```

Second example → user-defined inducing points

Alternatively, a random subset of the training points can be used as inducing points. Note, that there are plenty of methods to set these inducing points. So, in the second example let us use a user-defined set of inducing points.

You can pick a set of fixed inducing points by hand:

```
u = np.array([[-1], [-0.8], [-0.5], [0.3],[1.]])
```

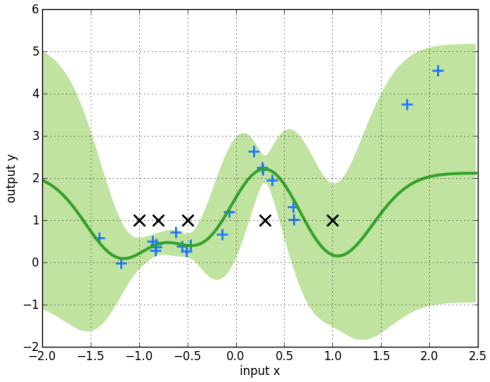
You can also use equidistant inducing points u, but without the values on the margin of the grid.(i.e. shrinking the range of values)

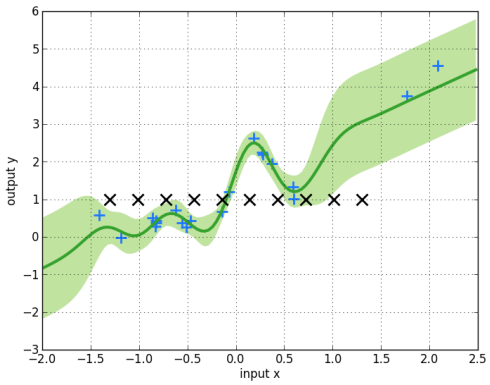
```
num_u = np.fix(x.shape[0]/2)
u = np.linspace(-1.3,1.3,num_u).T
u = np.reshape(u,(num_u,1))
```

Then pass u when specifying prior.

```
m = mean.Zero()
k = cov.RBFard(log_ell_list=[0.05,0.17], log_sigma=1.)
model.setPrior(mean=m, kernel=k, inducing_points=u)
```

The left figure below shows the result of fixed inducing points, and the right figure shows the result for equidistant u.





[Theory] Note that the predictive variance is overestimated outside the support of the inducing inputs. In a multivariate example where densely sampled inducing inputs are infeasible, one can also try to simply use a random subset of the training points.

A bit more things you can do

Switch to other Inference and Likelihood functions.

```
model.useInference("EP")
model.useLikelihood("Laplace")
```

Classification

Basic Classification

The demo shown in this tutorial can be obtained by running *pyGPs/Demo/demo_GPC.py*.

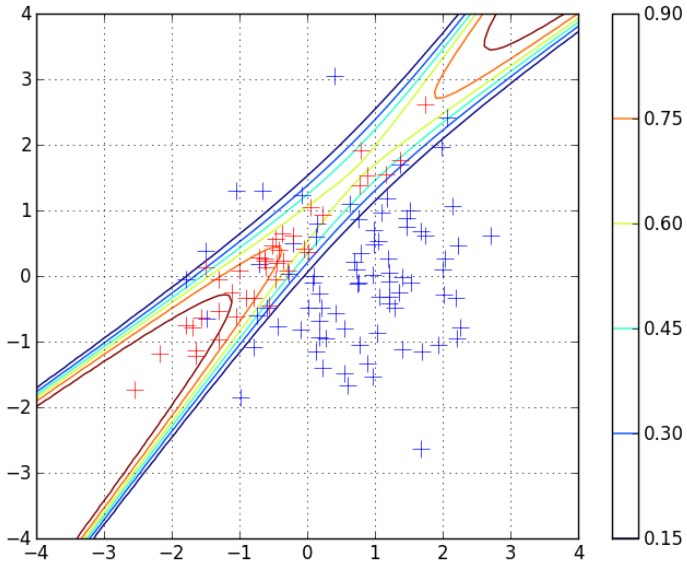
Load data

First, we import the data:

```
# GPC target class are +1 and -1
demoData = np.load('data_for_demo/classification_data.npz')
x = demoData['x']             # training data
y = demoData['y']             # training target
z = demoData['xstar']         # test data
```

The 120 data points were generated from two Gaussians with different means and covariances. One Gaussian is isotropic and contains $2/3$ of the data (blue), the other is highly correlated and contains $1/3$ of the points (red). Note, that the labels for the targets are specified to be ± 1 (and not $0/1$).

In the plot, we superimpose the data points with the posterior equi-probability contour lines for the probability of the second class given complete information about the generating mechanism.



First example → state default values

Again, lets see the simplest use of gp classification at first

```
model = gp.GPC()              # binary classification (default inference method: EP)
model.fit(x, y)               # fit default model (mean zero & rbf kernel) with data
model.train(x, y)             # optimize hyperparamters (default optimizer: single run minimize)
model.predict(z)              # predict test cases
```

Note, that inference is done via expectation propagation (EP) approximation by deault. How to set inference to Laplace approximation, see *A bit more things you can do*.

Second example → GP classification

So we first state the model to be GP classification now:

```
model = gp.GPC()
```

The rest is similar to GPR:

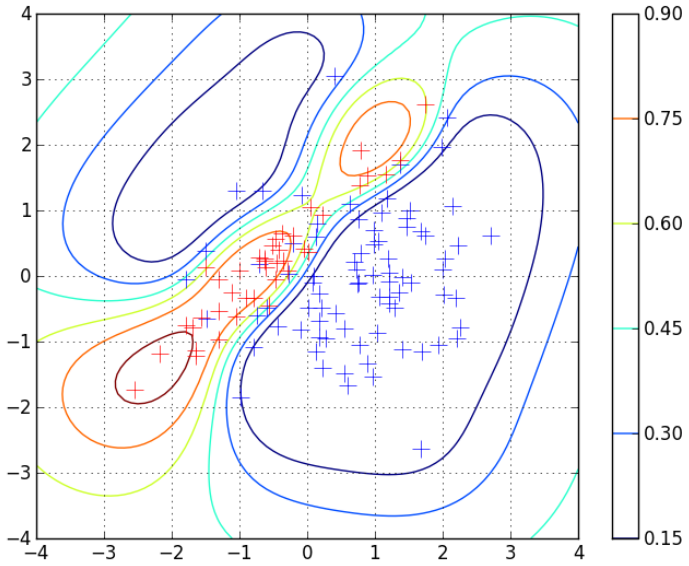
```
k = cov.RBFard(log_ell_list=[0.05,0.17], log_sigma=1.)
model.setPrior(kernel=k)

model.setData(x, y)
model.plotData_2d(x1,x2,t1,t2,p1,p2)

model.fit()
model.train()
model.predict(z, ys=np.ones((z.shape[0],1)))
```

[Theory] In this example, we used an RBF kernel (squared exponential covariance function) with automatic relevance determination (ARD). This covariance function has one characteristic length-scale parameter for each dimension of the input space (here 2 in total), and a signal magnitude parameter, resulting in a total of 3 hyperparameters. ARD with separate length-scales for each input dimension is a very powerful tool to learn which inputs are important for the predictions: if length-scales are short, input dimensions are very important, and when they grow very large (compared to the spread of the data), the corresponding input dimensions will be mostly ignored.

Note, *GPC.plot()* is a toy method for 2-d data:

```
model.plot(x1,x2,t1,t2)
```

The contour plot for the predictive distribution is shown below. Note, that the predictive probability is fairly close to the probabilities of the generating process in regions of high data density. Note also, that as you move away from the data, the probability approaches $1/3$, the overall class probability.



Examining the two ARD characteristic length-scale parameters after learning, you will find that they are fairly similar, reflecting the fact that for this data set, both input dimensions are important.

A bit more things you can do

GPC uses expectation propagation (EP) inference and Error function likelihood by default, you can explictly change to other methods:

```
model.useInference("Laplace")
```

Sparse Classification

The demo in this tutorial can be obtained by running *pyGPs/Demo/demo_GPC_FITC.py*. This demo is more or less a repetition of the demo of FITC regression.

First example → default inducing points

First load the same data as in the GPC demo.

[Theory] In case the number of training inputs x exceeds a few hundred, approximate inference using Laplacian Approximation or Expectation Propagation takes too long. As in regression, we offer the FITC approximation based on a low-rank plus diagonal approximation to the exact covariance to deal with these cases. The general idea is to use inducing points u and to base the computations on cross-covariances between training, test and inducing points only.

Okay, now the model is FITC classificiation:

```
model = gp.GPC_FITC()
```

The difference between the usage of basic GP is that we will have to specify inducing points. In our first example, we will introduce how to perform sparse GPC with the default settings.

The default inducing points form a grid (hypercube in higher dimension), where each dimension has 5 values in equidistant steps in $[min, max]$, where min and max are the minimum and maximum values of the input data by default. In order to specify the dimension of input data, we HAVE TO set data first:

```
model.setData(x, y)
```

The number of inducing points per axis is 5 per default. How to change this, see *A bit more things you can do*.

Then, the regular process follows:

```
model.train()
model.predict(z, ys=np.ones((z.shape[0],1)))
model.plot(x1,x2,t1,t2)
```

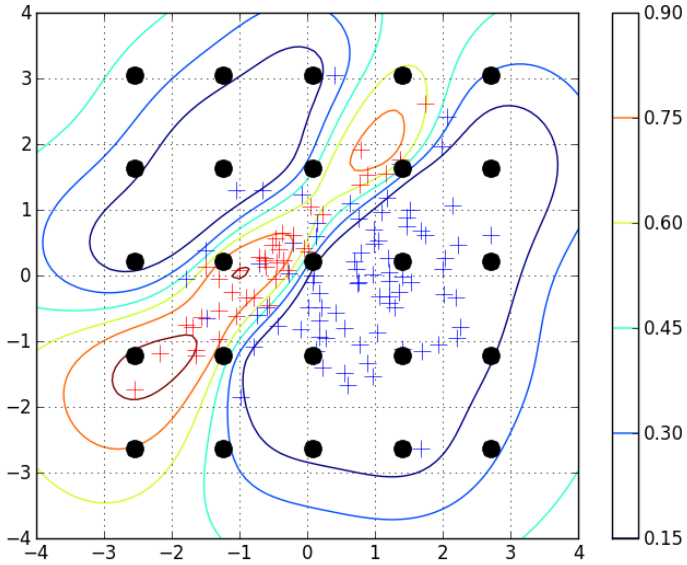
The equispaced default inducing points u are shown as black circles in the plot.

Second example → user-defined inducing points

Alternatively, a random subset of the training points can be used as inducing points. Note, that there are various different ways of how to set the inducing points. So, in the second example let us use a user-defined set of inducing points:

```
u1,u2 = np.meshgrid(np.linspace(-2,2,5),np.linspace(-2,2,5))
u = np.array(zip(np.reshape(u2,(np.prod(u2.shape),)),np.reshape(u1,(np.prod(u1.shape),))))
```

Here, we also use a grid euqually spaced, but without the values on the margin of the grid.(i.e. shrinking the grid) Then, we can just pass u when specifying prior:




```
m = mean.Zero()
k = cov.RBFard(log_ell_list=[0.05,0.17], log_sigma=1.)
model.setPrior(mean=m, kernel=k, inducing_points=u)
```

The prediction results for this set of inducing points are shown below:

A bit more things you can do

As in standard GPC, it is possible to use other inference/likelihood in the FITC method:

```
model.useInference("Laplace")
```

Change the number of inducing points per axis:

```
model.setData(x, y, value_per_axis=10)
```

Multi-class Classification

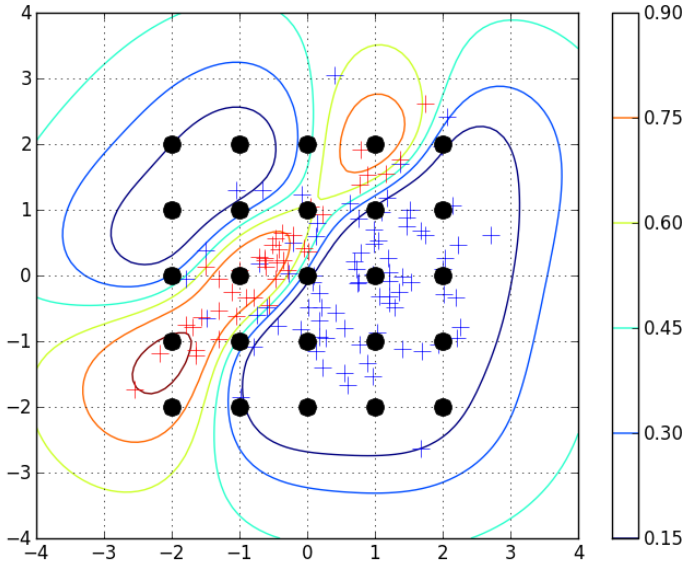
GPMC is NOT based on multi-class Laplace approximation. It works as a one vs. one classification wrapper. In other words, GPMC trains a GPC model for each pair of two classes, and uses a majority voting scheme over all results to determine the final class. The method only returns the predictive class with highest rating; no other values (such as variance) are returned.

Lets see a practical example to classify the 10 (0,1,2,...9) hand-written digits in the USPS digits dataset.

Load data

The USPS digits data were gathered at the Center of Excellence in Document Analysis and Recognition (CEDAR) at SUNY Buffalo, as part of a project sponsored by the US Postal Service. The dataset is described in [1].

[1] A Database for Handwritten Text Recognition Research, J. J. Hull, IEEE PAMI 16(5) 550-554, 1994.




```
data = loadmat('data_for_demo/usps_resampled.mat')
x = data['train_patterns'].T    # train patterns
y = data['train_labels'].T      # train labels
xs = data['test_patterns'].T    # test patterns
ys = data['test_labels'].T      # test labels
```

To be used in GPMC, labels should start from 0 to k (k = number of classes).

GPMC example

State model with 10-class classification problem:

```
model = gp.GPMC(10)
```

Pass data to model:

```
model.setData(x,y)
```

Train default GPC model for each binary classification problem, and decide label for test patterns of hand-writen digits. The return value *predictive_vote[i,j]* is the probability of being class *j* for test pattern *i*.

```
predictive_vote = model.trainAndPredict(xs)
predictive_class = np.argmax(predictive_vote, axis=1)
```

Just like we did for GP classification, you can use specific settings (other than default) for all binary classificiation problem for example by:

```
m = mean.Zero()
k = cov.RBF()
model.setPrior(mean=m,kernel=k)
model.useInference("Laplace")
```

For more information on how to use non-default settings see demo_GPC and demo_GPR.

Beside *trainAndPredict(xs)*, there is also an option to perform prediction without hyperparameter optimization:

```
model.fitAndPredict(xs)
```

Some examples for real-world data

K-fold Cross-Validation

In this demo, we'll show you the typical process of using GP for machine learning from loading data, preprocessing, training, predicting to validation and evaluation.

Load data

We use the ionosphere dataset [2] from Johns Hopkins University Ionosphere database. It is available in UCI machine learning repository. Then we need to do some data cleaning. Here we deal with label in ionosphere data, change "b" to"-1", and "g" to "+1". These preprocessing implementation are availabe in the source code.

Cross Validation

Now, lets focus on the use of cross-validation.

```
K = 10              # number of folds
for x_train, x_test, y_train, y_test in valid.k_fold_validation(x, y, K):
    # This is a binary classification problem
    model = gp.GPC()
    # Since no prior knowldege, leave everything default
    model.train(x_train, y_train)
    # Predition
    ymu, ys2, fmu, fs2, lp = model.predict(x_test, ys=y_test)
    # ymu for classification is a continuous value over -1 to +1
    # If you want predicting result to either one of the classes, take a sign of ymu.
    ymu_class = np.sign(ymu)
    # Evluation
    acc = valid.ACC(ymu_class, y_test)       # accuracy
    rmse = valid.RMSE(ymu_class, y_test)     # root-mean-square error
```

Evaluation measures

We implemented some classical evaluation measures.

- RMSE - root mean squared error
- ACC - classification/regression accuracy
- Prec - classification precision for class +1
- Recall - classification recall for class +1
- NLPD - negative log predictive density in transformed observation space

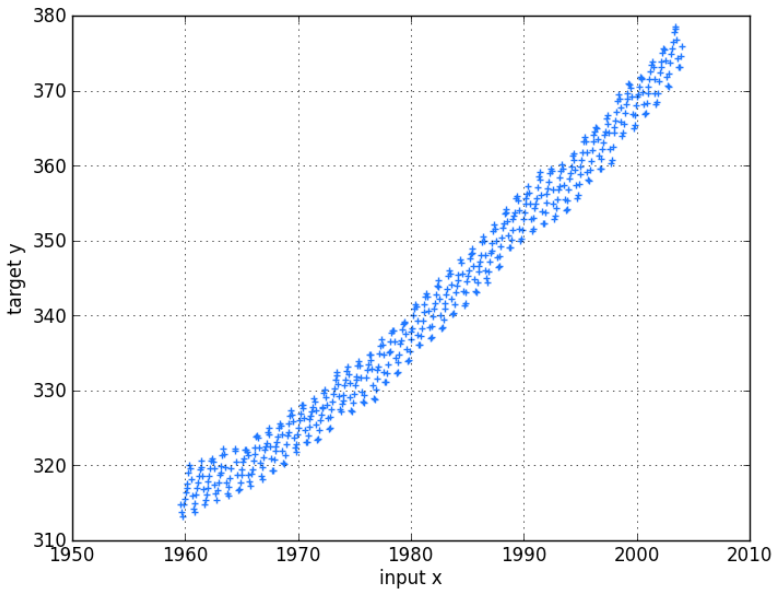
[2] Sigillito, V. G., Wing, S. P., Hutton, L. V., & Baker, K. B. (1989). Classification of radar returns from the ionosphere using neural networks. Johns Hopkins APL Technical Digest, 10, 262-266.

Regression on Mauna Loa data

This example does regression on the Hawaiian Mauna Loa data (example taken from chapter 5 of the GPML book by Rasmussen and Williams, 2006)

We will use a modelling problem concerning the concentration of CO_2 in the atmosphere to illustrate how the marginal likelihood can be used to set multiple hyperparameters in hierarchical Gaussian process models. A complex covariance function is derived by combining several different kinds of simple covariance functions, and the resulting model provides an excellent fit to the data as well as insight into its properties by interpretation of the adapted hyperparameters. Although the data is one-dimensional, and therefore easy to visualize, a total of 11 hyperparameters are used, which in practice rules out the use of cross-validation for setting parameters, except for the gradient-based LOO-CV procedure.

The data [3] consists of monthly average atmospheric CO_2 concentrations (in parts per million by volume (ppmv)) derived from *in-situ* air samples collected at the Mauna Loa Observatory, Hawaii, between 1958 and 2003 (with some missing values) [2].



The data is shown in the above plot. Our goal is to model the CO_2 concentration as a function of time t. Several features are immediately apparent: a long term rising trend, a pronounced seasonal variation and some smaller irregularities. In the following, contributions to a combined covariance function which takes care of these individual properties are suggessted. This is meant primarily to illustrate the power and flexibility of the Gaussian process framework—it is possible that other choices would be more appropriate for this data set.

To model the long term smooth rising trend, a squared exponential (SE) covariance term with two hyperparameters controlling the amplitude θ_1 and characteristic length-scale θ_2 is used:

$$k_1(x, x') = \theta_1^2 \exp\left(-\frac{(x - x')^2}{2\theta_2^2}\right).$$

Note that we just use a smooth trend; actually enforcing the trend *a priori* to be increasing is probably not so simple and (hopefully) not desirable. We can use the periodic covariance function with a period of one year to model the

[3] Keeling, C. D. and Whorf, T. P. (2004). Atmospheric CO_2 Records from Sites in the SIO Air Sampling Network. In Trends: A Compendium of Data on Global Change. Carbon Dioxide Information Analysis Center, Oak Ridge National Laboratory, Oak Ridge, Tenn., U.S.A.

seasonal variation. However, it is not clear that the seasonal trend is exactly periodic, so we modify it by taking the product with a squared exponential component to allow a decay away from exact periodicity:

$$k_2(x,x') = \theta_3^2 \exp\left(-\frac{(x-x')^2}{2\theta_4^2}\frac{2\sin^2(\pi(x-x'))}{\theta_5^2}\right).$$

where θ_3 gives the magnitude, θ_4 the decay-time for the periodic component, and θ_5 the smoothness of the periodic component; the period has been fixed to one (year). The seasonal component in the data is caused primarily by different rates of CO_2 uptake for plants depending on the season, and it is probably reasonable to assume that this pattern may itself change slowly over time, partially due to the elevation of the CO_2 level itself; if this effect turns out not to be relevant, then it can be effectively removed at the fitting stage by allowing θ_4 to become very large.

To model the (small) medium term irregularities, a rational quadratic term is used:

$$k_3(x,x') = \theta_6^2\left(1+\frac{(x-x')^2}{2\theta_8\theta_7^2}\right)^{\theta_8}.$$

where θ_6 is the magnitude, θ_7 is the typical length-scale and θ_8 is the shape parameter determining diffuseness of the length-scales.

One could also have used a squared exponential form for this component, but it turns out that the rational quadratic works better (gives higher marginal likelihood), probably because it can accommodate several length-scales simultaneously.

Finally we specify a noise model as the sum of a squared exponential contrubition and an independent component:

$$k_4(x_p,x_q) = \theta_9^2 \exp\left(-\frac{(x_p-x_q)^2}{2\theta_{10}^2}\right) + \theta_{11}^2\delta_{pq}.$$

where θ_9 is the magnitude of the correlated noise component, θ_{10} is its length scale and θ_{11} is the magnitude of the independent noise component. Noise in the series could be caused by measurement inaccuracies, and by local short-term weather phenomena, so it is probably reasonable to assume at least a modest amount of correlation in time. Notice that the correlated noise component, the first term has an identical expression to the long term component in the trend covariance. When optimizing the hyperparameters, we will see that one of these components becomes large with a long length-scale (the long term trend), while the other remains small with a short length-scale (noise). The fact that we have chosen to call one of these components 'signal' and the other one 'noise' is only a question of interpretation. Presumably, we are less interested in very short-term effect, and thus call it noise; if on the other hand we were interested in this effect, we would call it signal.

The final covariance function is:

$$k(x,x') = k_1(x,x') + k_2(x,x') + k_3(x,x') + k_4(x,x')$$

with hyperparameters $\theta = (\theta_1,\ldots,\theta_{11})$

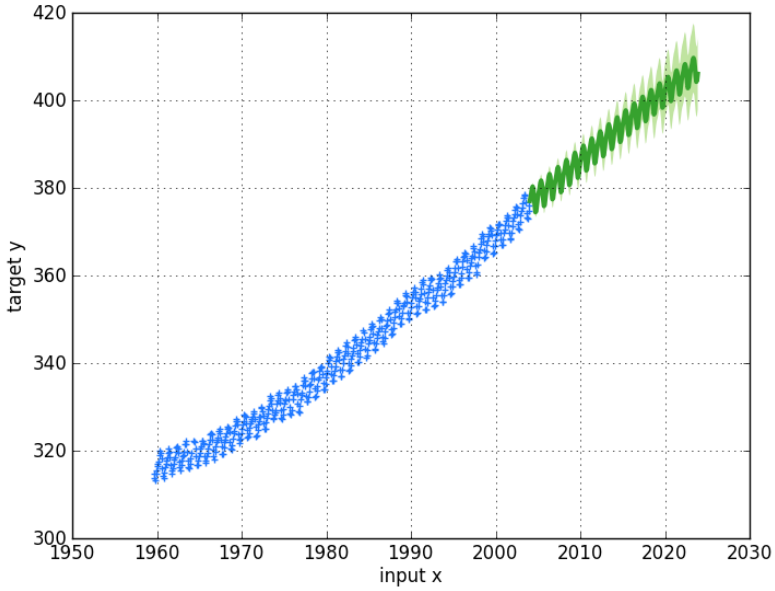
```
# DEFINE parameterized covariance function
k1 = cov.RBF(np.log(67.), np.log(66.))
k2 = cov.Periodic(np.log(1.3), np.log(1.0), np.log(2.4)) * cov.RBF(np.log(90.), np.log(2.4))
k3 = cov.RQ(np.log(1.2), np.log(0.66), np.log(0.78))
k4 = cov.RBF(np.log(1.6/12.), np.log(0.18)) + cov.Noise(np.log(0.19))
k  = k1 + k2 + k3 + k4
```

After running the minimization,

```
t0 = clock()
model.train(x,y)
t1 = clock()
model.predict(xs)
```

The extrapolated data looks like:

and the optimized values of the hyperparameters allow for a principled analysis of different components driving the model.



Regression on UCI Housing data

Boston Housing is a fairly standard dataset used for testing regression problems. It contains 506 data points with 12 numeric attributes, and one binary categorical attribute. The goal is to determine median home values, based on various census attributes. This dataset is available at the UCI Repository.

The demo follows that in [4]. The data set was preprocessed as follows: each continuous feature was transformed to zero mean and unit variance (The categorical variable was dropped). The data was partitioned into 481 points for training and 25 points for testing.

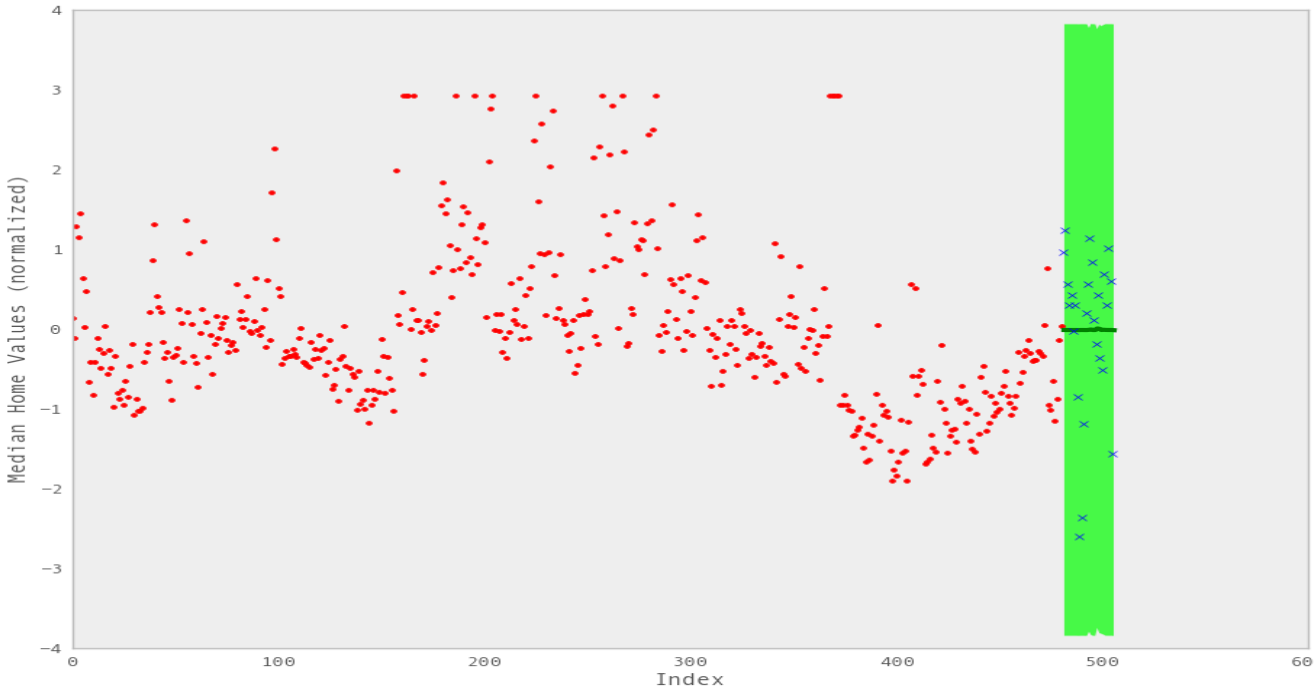
The mean function used was `src.Core.means.meanZero()` and the covariance (using the `src.Core.kernels.covSum()` function) was a composite of `src.Core.kernels.covSEiso()` and `src.Core.kernels.covNoise()`. The initial values of the hyperparameters were selected randomly from a zero-mean, unit-variance normal distribtion. The actual values were: $[-0.75,\ 0.59,\ -0.45]$. The initial likelihood hyperparameter was -2.30. The regression started with initial negative log marginal likelihood of :math:‘ 752.46‘. Note the initial zero mean and the variance that is uniform over the test set.

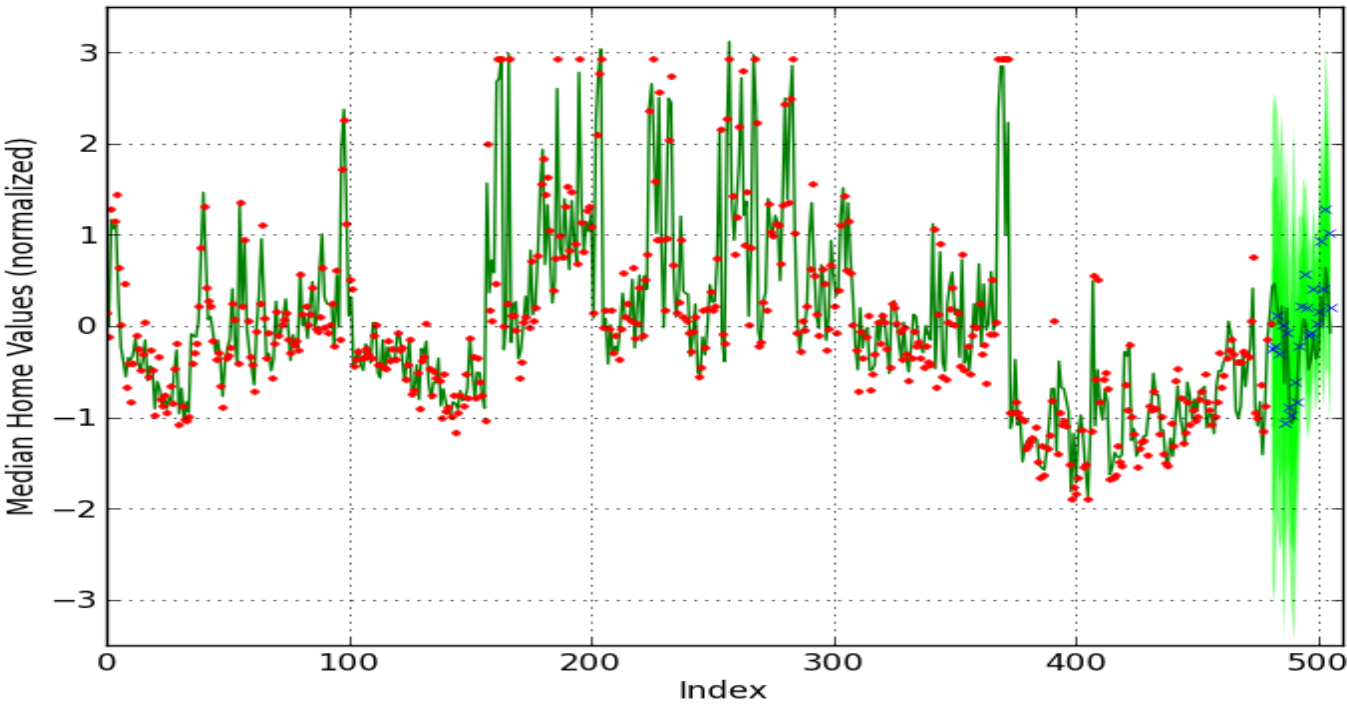
```
model = gp.GPR()
model.train(x,y)
ym, ys2, fm, fs2, lp = model.predict(xs)
xa  = np.concatenate((data[:,:4],data[:,5:-1]),axis=1)
xa = (xa - np.mean(xa,axis=0))/(np.std(xa,axis=0)+1.e-16)
ya, ys2a, fma, fs2a, lpa = model.predict(xa)
```

After hyperparameter optimization, the covariance hyperparameters were $[1.17,\ 0.45,\ -1.41]$ and the likelihood hyperparameter was -2.27. The final negative log marginal likelihood (optimized) was 214.46.

[4]

20. Suttorp and C. Igel, Approximation of Gaussian process regression models after training. In M. Verleysen (Hrsg.), Proceedings of the 16th European Symposium on Artificial Neural Networks (ESANN 2008) , pp. 427–432 (2008).


Median Home Values (normalized)
Index
4
3
2
1
0
−1
−2
−3
−4
0
100
200
300
400
500
60


Median Home Values (normalized)
Index
3
2
1
0
−1
−2
−3
0
100
200
300
400
500

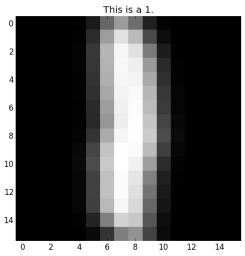
Semi-supervised Learning with Graphs

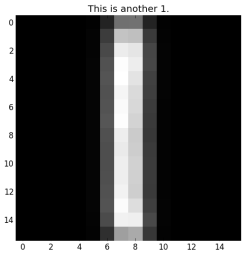
The code shown in this tutorial can be executed by running *pyGPs/Demo/demo_KernelOnGraph.py*

Load data

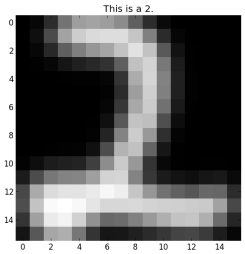
We used the same dataset from GPMC example. i.e. The USPS digits dataset [5]. Each digit of $16 * 16$ pixels is flattened into a 256 dimension vector. For the simplicity of demo, we only selected digits 1 s and 2 s such that we have a binary classification problem where digit 1 for class +1 and digit 2 for class -1. We also reduced the dataset into 100 samples per digit, where the original dataset consist of thousands of samples for each digit.

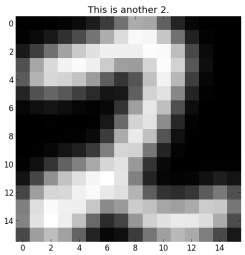
Here are samples for two digits for 1





and samples for two digits for 2





[5] A Database for Handwritten Text Recognition Research, J. J. Hull, IEEE PAMI 16(5) 550-554, 1994.

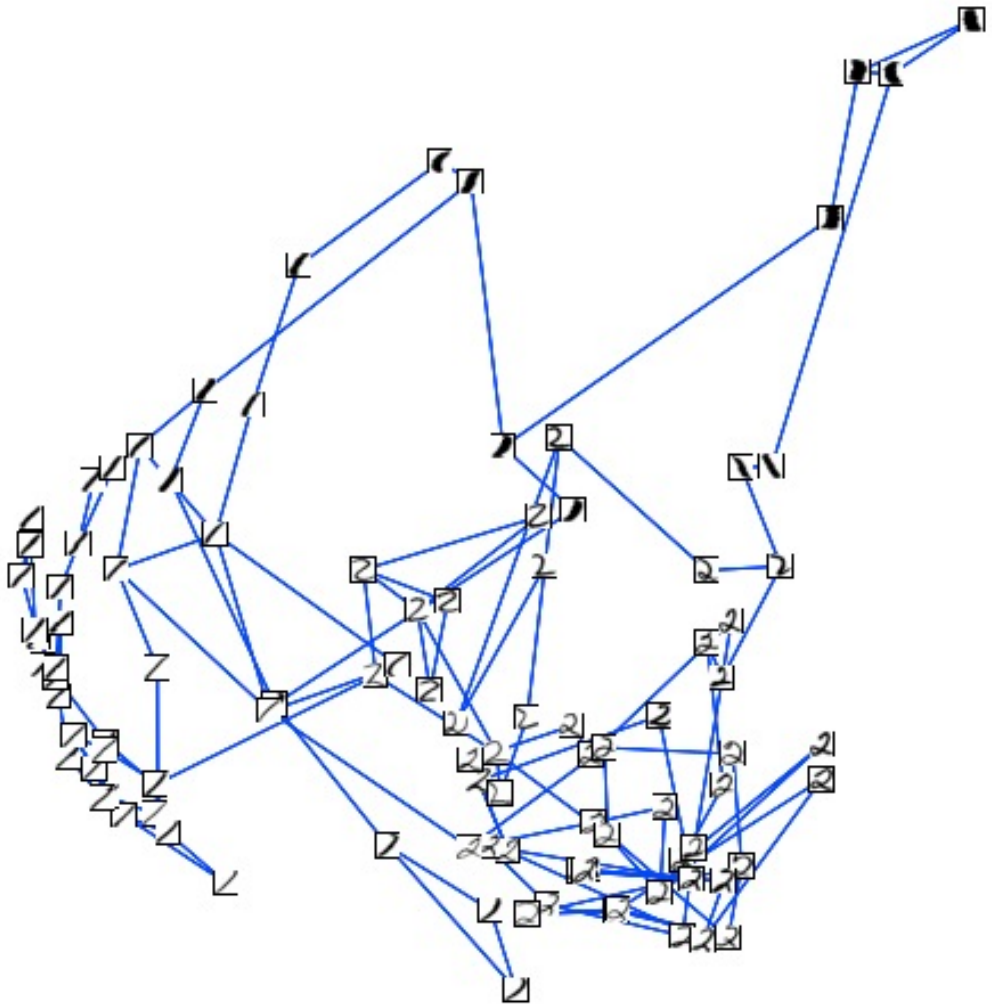
Form a nearest neighbour graph

We form a nearest-neighbor graph based on Euclidean distance of the vector representation of digits. Neighboring images have small Euclidean distance. Each digit is a node in the graph. There is an edge if digit i is the k-nearest neighbour of digit j. We form a symmetrized graph such that we connect nodes j, i if i is in j's kNN and vice versa, and therefore a node can have more than k edges. You should import the corresponding module from *pyGPs.GraphStuff*

```
x,y = load_binary(1,2,reduce=True)
A = form_knn_graph(x,2)
```

A is the adjacency matrix of this $2-NN$ graph.

Below shows an example of such symmetrized Euclidean $2-NN$ graph on some 1s and 2s taking from Xiaojin Zhu's doctoral thesis [6].



[6] Semi-Supervised Learning with Graphs, Xiaojin Zhu, CMU-LTI-05-192, 2005

Kernel on graph

Several classical kernels on graph described in Structured Kernels can be built from adjacency matrix A. We use diffusion kernel for this example to get the precomputed kernel matrix.

```
Matrix = diffKernel(A)
```

This a big square matrix with all rows and columns of the number of data points. By specifying the indice of training data and test data, we will form two matrix M1 and M2 with the exact format which *pyGPs.Core.cov.Pre* needed.

```
M1,M2 = form_kernel_matrix(Matrix, indice_train, indice_test)
```

M1 is a matrix with shape number of training points plus 1 by number of test points

- cross covariances matrix (train by test)
- last row is self covariances (diagonal of test by test)

M2 is a square matrix with number of training points for each dimension

- training set covariance matrix (train by train)

GP classification

Every ingredients for a basic semi-supervised learning is prepared now. Lets see how to proceed for GP classification. First, the normal way with rbf kernel we have seen several times

```
model = gp.GPC()
k = cov.RBF()
model.setPrior(kernel=k)
```

Then lets use our kernel precomputed matrix. If you only use precomputed kernel matrix, there is no training data. However you still need to specify x just to fit in the usage of pyGPs for generality reason. You can create any x as long as the dimension is correct.

```
x = np.zeros((n,1))
k = cov.Pre(M1,M2) + cov.RBF()
model.setPrior(kernel=k)
```

Moreover, you can composite a kernel for both precomputed matrix and regular kernel function if necessary.

```
k = cov.Pre(M1,M2) + cov.RBFunit()
model.setPrior(kernel=k)
```

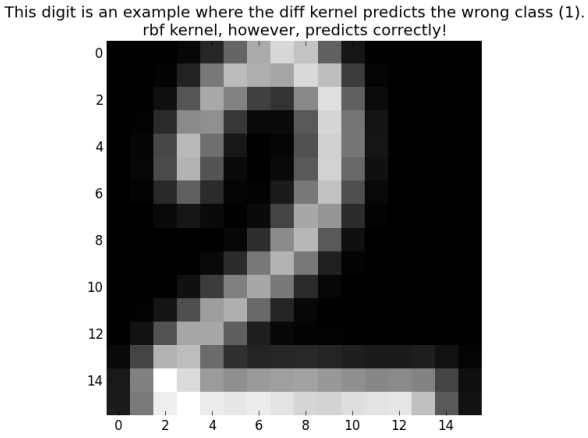
The rest way of using pyGPs is exactly the same as the demo of GP classification.

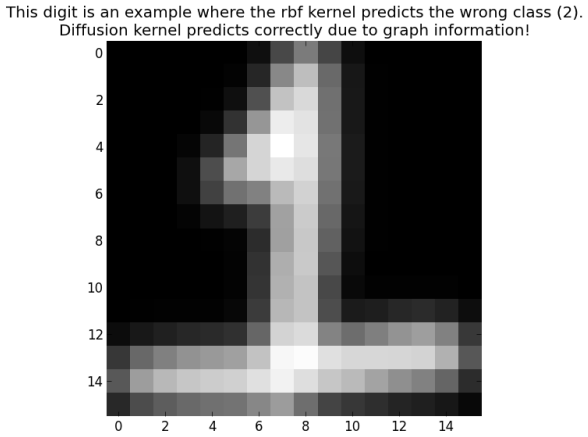
Result

For our manually created graph data, an rbf kernel works better than a diffusion kernel on the graph (higher accuracy). The performance in general should depend on the application as well as features of data.

The left image shows the digit that using diffusion kernel will predict the wrong result (should be 2), but rbf kernel does the job fine. The right image shows the digit that rbf kernel predicts the wrong class, diffusion kernel on the other hand, predicts correctly due to graph information! (should be 1).

Interestingly, using a composite kernel with diffusion kernel on graph and an rbf kernel together. All test cases including the following are predicted correctly.





1.4 GP functionality

1.4.1 Kernels & Means

Simple Kernel & Mean

You may already seen, we can specify a kernel function like this(same for mean fucntions):

```
k = cov.RBF( log_ell=-1., log_sigma=0. )
```

There are several points need to be noticed:

1. Most parameters are initilized in their logorithms. This is because we need to make sure they are positive during optimization. e.g. Here length scale and signal variance should always be positive.
2. Most kernel functions have a scalar in front, namely signal variance(set by log_sigma)
3. If you will do optimization later anyway, you can just leave parameters to be default

Some Special Cases

1. For some kernels/means, number of hyperparameters depends on the dimension of input data. You can either enter the dimension, which use default values:

   ```
   m = mean.Linear( D=x.shape[1] )
   ```

   or you can initialze with the exact hyperparameters, you should enter as a list, one element for each dimension

   ```
   m = mean.Linear( alpha_list=[0.2, 0.4, 0.3] )
   ```

   All these "hyp-dim-dependent" functions are:

   - *mean.Linear*
   - *cov.RBFard*
   - *cov.LINard*
   - *cov.RQard*

2. For linear kernel, there is NO signal variance(scalar) in front of the function.

   If you want to add a scalar for it, you can use:

   ```
   k = 0.5 * cov.LIN()
   ```

   If you also want to add a bias term:

   ```
   k = 0.5 * cov.LIN() + cov.Const(c=1.)
   ```

   Note 0.5 will also be treated as a hyperparameter. This also applies in *cov.LINard*.

3. For *cov.RBFunit()*, its signal variance is always 1 (because of unit magnitude). Therefore this function do not have a hyperparameter of "signal variance".

4. ***cov.Poly()* has three parameters, where hyperparameters are:**

   - c -> inhomogeneous offset
   - sigma -> signal deviation

   however,

   - d -> order of polynomial will be treated as normal parameter, i.e. will not be trained

5. Explicitly set *cov.Noise* is not necessary, because noise are already added in likelihood.

Composite Kernels & Meams

Adding and muliplying Kernels(Means) is really simple:

```
k = cov.Periodic() * cov.RBF()
k = 0.5*cov.LIN() + cov.Periodic()
```

Scalar will also be treated as a hyperparameter. For example, k = s1 * k1 + s2 * k2, then the list of hyperparameters is hyp = [s1, k1.hyp, s2, k2.hyp]. Scalar is passed in logorithm domain such that it will always be positive during optimization.

Except linear kernel, all kernel functions have a scalar (signal variance) as hyperparameter. Therefore, the only explict scalar might be added to cov.LIN()

Beside + / * , there is also a power operator for mean functions:

```
m = ( mean.One()+mean.Linear(alpha_list=[0.2]) )**2
```

Precomputed Kernel Matrix

In certain cases, you may have a precomputed kernel matrix, but its non-trivial to write down the exact formula of kernel functions. Then you can specify your kernel in the following way. A precomputed kernel also fits with other kernels. In other words, it can also be composited as the way other kernels functions do.

```
k = cov.Pre(M1, M2)
```

M1 and M2 are your precomputed kernel matrix,

where,

M1 is a matrix with shape number of training points plus 1 by number of test points

- cross covariances matrix (train by test)
- last row is self covariances (diagonal of test by test)

M2 is a square matrix with number of training points for each dimension

- training set covariance matrix (train by train)

A precomputed kernel can also be composited with other kernels. Similar to *cov.LIN()*, you need to explictly add scalar for *cov.Pre()*.

```
k = 0.5*cov.Pre(M1, M2) + cov.RBF()
```

Customizing Kernel & Mean

We also support you to create your own kernel/mean class, your customized kernel class need to follow the structure template as below:

```
# Your kernel class needs to inherit base class Kernel,
# which is in the module of Core.cov
class MyKernel(Kernel):

    def __init__(self, para1=0., para2=0., para3=0.):
        self.hyp = [para1, para2]     # hyperparameters that can be trained
        self.para = [para3]           # static parameters

    def proceed(self, x=None, z=None, der=None):
        ''' x is n by D training patterns matrix, and z is nn by D test case matrix'''
        return A
```

where the returning matrix A depends on the input:

- if *z* == *None*, A is covariance matrix of x with shape (n,n)
- elif *z* == *'diag'*, A is self covariance matrix with shape (n,1)
- else *z is a matrix (given test points)*, A is covariance between data sets x and z with shape (n,nn)
- if *der* == *None*, return A as defined previously.
- else *der !*= *None*, i.e. given der as an integer der = k, return the derivative matrix wrt. to k_{th} hyperparameter.

and for customized mean class:

```
# Your mean class needs to inherit base class Mean,
# which is in the module of Core.mean
class MyMean(Mean):

    def __init__(self, para1=0., para2=0., para3=0.):
        self.hyp = [para1, para2]     # hyperparameters that can be trained
        self.para = [para3]           # static parameters

    def proceed(self, x=None, der=None):
        ''' x is n by D training patterns matrix'''
        return A
```

where the returning matrix A depends on the input:

- if *der* == *None*, return A as the mean of x
- else *der !*= *None*, return the derivative of mean wrt. to k_{th} hyperparameter.

1.4.2 Likelihoods & Inference

Changing Likelihood & Inference

By default,

- GPR uses Gaussian likelihood and exact inference.
- GPC uses Error functionlikelihood and EP inference.
- FITC model uses same default with corresponding FITC inference.
- GPMC calls GPC and thus uses the default setting of GPC

You can change to other likelihood or inference methods using:

```
model.useLikelihood(newLik)
model.useInference(newInf)
```

***newLik* and *newInf* are Strings. Currently the options are:**

1. Regression model
   - newLik: **"Laplace"**. Note this will force inference method to be EP.
   - newInf: **"EP"**, **"Laplace"**.
2. Classification model (including GPMC)
   - newInf: **"Laplace"**

List of Likelihoods

class `pyGPs.Core.lik.`**`Erf`**

Error function or cumulative Gaussian likelihood function for binary classification or probit regression.

$Erf(t) = \frac{1}{2}(1 + erf(\frac{t}{\sqrt{2}})) = normcdf(t)$

class `pyGPs.Core.lik.`**`Gauss`**(*log_sigma=-2.3025850929940455*)

Gaussian likelihood function for regression.

$Gauss(t) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t-y)^2}{2\sigma^2}}$, where y is the mean and σ is the standard deviation.

hyp = [log_sigma]

class pyGPs.Core.lik.**Laplace**(*log_sigma=-2.3025850929940455*)
Laplacian likelihood function for regression. ONLY works with EP inference!

$Laplace(t) = \frac{1}{2b}e^{-\frac{|t-y|}{b}}$ where $b = \frac{\sigma}{\sqrt{2}}$, y is the mean and σ is the standard deviation.

hyp = [log_sigma]

class pyGPs.Core.lik.**Likelihood**
Base function for Likelihood function

List of Inference

class pyGPs.Core.inf.**EP**
Expectation Propagation approximation to the posterior Gaussian Process.

class pyGPs.Core.inf.**Exact**
Exact inference for a GP with Gaussian likelihood. Compute a parametrization of the posterior, the negative log marginal likelihood and its derivatives w.r.t. the hyperparameters.

class pyGPs.Core.inf.**FITC_EP**
FITC-EP approximation to the posterior Gaussian process. The function is equivalent to infEP with the covariance function: Kt = Q + G; G = diag(g); g = diag(K-Q); Q = Ku' * inv(Kuu + snu2 * eye(nu)) * Ku; where Ku and Kuu are covariances w.r.t. to inducing inputs xu and snu2 = sn2/1e6 is the noise of the inducing inputs. We fixed the standard deviation of the inducing inputs snu to be a one per mil of the measurement noise's standard deviation sn. In case of a likelihood without noise parameter sn2, we simply use snu2 = 1e-6. For details, see The Generalized FITC Approximation, Andrew Naish-Guzman and Sean Holden, NIPS, 2007.

class pyGPs.Core.inf.**FITC_Exact**
FITC approximation to the posterior Gaussian process. The function is equivalent to infExact with the covariance function: Kt = Q + G; G = diag(g); g = diag(K-Q); Q = Ku' * inv(Quu) * Ku; where Ku and Kuu are covariances w.r.t. to inducing inputs xu, snu2 = sn2/1e6 is the noise of the inducing inputs and Quu = Kuu + snu2*eye(nu).

class pyGPs.Core.inf.**FITC_Laplace**
FITC-Laplace approximation to the posterior Gaussian process. The function is equivalent to infLaplace with the covariance function: Kt = Q + G; G = diag(g); g = diag(K-Q); Q = Ku' * inv(Kuu + snu2 * eye(nu)) * Ku; where Ku and Kuu are covariances w.r.t. to inducing inputs xu and snu2 = sn2/1e6 is the noise of the inducing inputs. We fixed the standard deviation of the inducing inputs snu to be a one per mil of the measurement noise's standard deviation sn. In case of a likelihood without noise parameter sn2, we simply use snu2 = 1e-6.

class pyGPs.Core.inf.**Inference**
Base class for inference. Defined several tool methods in it.

class pyGPs.Core.inf.**Laplace**
Laplace's Approximation to the posterior Gaussian process.

1.4.3 Optimizers

Optimization Methods

As you may have already seen in the demos, the optimizer is initialized in the following way:

GP.**setOptimizer**(*method*, *num_restarts=None*, *min_threshold=None*, *meanRange=None*, *covRange=None*, *likRange=None*)
This method is used to sepecify optimization configuration. By default, gp uses a single run "minimize".

Parameters

- **method** – Optimization methods. Possible values are:

  "Minimize" -> minimize by Carl Rasmussen (python implementation of "minimize" in GPML)

  "CG" -> conjugent gradient

  "BFGS" -> quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS)

  "SCG" -> scaled conjugent gradient (faster than CG)

- **num_restarts** – Set if you want to run mulitiple times of optimization with different initial guess. It specifys the maximum number of runs/restarts/trials.
- **min_threshold** – Set if you want to run mulitiple times of optimization with different initial guess. It specifys the threshold of objective function value. Stop optimization when this value is reached.
- **meanRange** – The range of initial guess for mean hyperparameters. e.g. meanRange = [(-2,2), (-5,5), (0,1)]. Each tuple specifys the range (low, high) of this hyperparameter, This is only the range of initial guess, during optimization process, optimal hyperparameters may go out of this range.

  (-5,5) for each hyperparameter by default.

- **covRange** – The range of initial guess for kernel hyperparameters. Usage see meanRange
- **likRange** – The range of initial guess for likelihood hyperparameters. Usage see meanRange

1.5 Extensions

1.5.1 Kernels for Graph Data

You can refer to our demo of semi-supervised learning for a simple usage of kernels for graph data.

Kernels on Graph

pyGPs.GraphStuff.kernels_on_graph.**VNDKernel**(*A*, *alpha=0.5*)
Von Neumann Diffusion Kernel on graph (Zhou et al., 2004) (also label spreading kernel)

K = (I - alpha*S)^-1, where S = D^-1/2*A*D^-1/2

Parameters

- **A** – adjacency matrix
- **alpha** – (hyper)parameter(s)

pyGPs.GraphStuff.kernels_on_graph.**cosKernel**(*A*)
Cosine Kernel (also Inverse Cosine Kernel)

K = cos (L*pi/4), where L is the normalized Laplacian

Parameters **A** – adjacency matrix

pyGPs.GraphStuff.kernels_on_graph.**diffKernel**(*A*, *beta=0.5*)
Diffusion Process Kernel

K = exp(beta * H), where H = -L = A-D

K = Q exp(beta * Lambda) Q.T

Parameters

- **A** – adjacency matrix
- **beta** – (hyper)parameter(s)

pyGPs.GraphStuff.kernels_on_graph.**normLap**(*A*)
normalized Laplacian

pyGPs.GraphStuff.kernels_on_graph.**psInvLapKernel**(*A*)
Pseudo inverse of the normalized Laplacian.

Parameters **A** – adjacency matrix

pyGPs.GraphStuff.kernels_on_graph.**regLapKernel**(*A*, *sigma=1*)
Regularized Laplacian Kernel

Parameters

- **A** – adjacency matrix
- **sigma** – (hyper)parameter(s)

pyGPs.GraphStuff.kernels_on_graph.**rwKernel**(*A*, *p=1*, *a=2*)
p-step Random Walk Kernel with a>1

K = (aI-L)^p, p>1 and L is the normalized Laplacian

Parameters

- **A** – adjacency matrix
- **p** – step parameter
- **a** – (hyper)parameter(s)

Graph Kernels

tbd.

CHAPTER

TWO

API

2.1 pyGPs

2.1.1 pyGPs Package

pyGPs Package

Subpackages

Core Package

Core Package

cov Module

class pyGPs.Core.cov.**Const**(*log_sigma=0.0*)

Bases: pyGPs.Core.cov.Kernel

Constant kernel. hyp = [log_sigma]

Parameters **log_sigma** – signal deviation.

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**FITCOfKernel**(*cov*, *inducingInput*)

Bases: pyGPs.Core.cov.Kernel

Covariance function to be used together with the FITC approximation. The function allows for more than one output argument and does not respect the interface of a proper covariance function. Instead of outputing the full covariance, it returns cross-covariances between the inputs x, z and the inducing inputs xu as needed by infFITC

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

getHyp()

hyp

setHyp(*hyp*)

class pyGPs.Core.cov.**Kernel**
Bases: object

This is a base class of Kernel functions there is no computation in this class, it just defines rules about a kernel class should have each covariance function will inherit it and implement its own behaviour

fitc(*inducingInput*)

sq_dist(*a*, *b=None*)

class pyGPs.Core.cov.**LINard**(*D=None*, *log_ell_list=None*)
Bases: pyGPs.Core.cov.Kernel

Linear covariance function with Automatic Relevance Detemination. hyp = log_ell_list

Parameters

- **D** – dimension of training data. Set if you want default ell, which is 1 for each dimension.
- **log_ell_list** – characteristic length scale for each dimension.

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**Linear**
Bases: pyGPs.Core.cov.Kernel

Linear kernel. No hyperparameters.

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**Matern**(*log_ell=0.0*, *d=3*, *log_sigma=0.0*)
Bases: pyGPs.Core.cov.Kernel

Matern covariance function with nu = d/2 and isotropic distance measure. For d=1 the function is also known as the exponential covariance function or the Ornstein-Uhlenbeck covariance in 1d. d will be rounded to 1, 3, or 5. hyp = [log_ell, log_sigma]

Parameters

- **d** – d is 2 times nu. Can only be 1,3 or 5.
- **log_ell** – characteristic length scale.
- **log_sigma** – signal deviation.

dfunc(*d*, *t*)

dmfunc(*d*, *t*)

func(*d*, *t*)

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

mfunc(*d*, *t*)

class pyGPs.Core.cov.**Noise**(*log_sigma=0.0*)
Bases: pyGPs.Core.cov.Kernel

Independent covariance function, i.e "white noise", with specified variance. Normally NOT used anymore since noise is now added in liklihood. hyp = [log_sigma]

Parameters **log_sigma** – signal deviation.

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**Periodic**(*log_ell=0.0*, *log_p=0.0*, *log_sigma=0.0*)
Bases: pyGPs.Core.cov.Kernel

Stationary kernel for a smooth periodic function. hyp = [log_ell, log_p, log_sigma]

Parameters

- **log_p** – period.
- **log_ell** – characteristic length scale.
- **log_sigma** – signal deviation.

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**PiecePoly**(*log_ell=0.0*, *v=2*, *log_sigma=0.0*)
Bases: pyGPs.Core.cov.Kernel

Piecewise polynomial kernel with compact support. hyp = [log_ell, log_sigma]

Parameters

- **log_ell** – characteristic length scale.
- **log_sigma** – signal deviation.
- **v** – degree v will be rounded to 0,1,2,or 3. (not treated as hyperparameter, i.e. will not be trained).

dfunc(*v*, *r*, *j*)

dpp(*r*, *j*, *v*, *func*, *dfunc*)

func(*v*, *r*, *j*)

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

pp(*r*, *j*, *v*, *func*)

ppmax(*A*, *B*)

class pyGPs.Core.cov.**Poly**(*log_c=0.0*, *d=2*, *log_sigma=0.0*)
Bases: pyGPs.Core.cov.Kernel

Polynomial covariance function. hyp = [log_c, log_sigma]

Parameters

- **log_c** – inhomogeneous offset.
- **log_sigma** – signal deviation.
- **d** – degree of polynomial (not treated as hyperparameter, i.e. will not be trained).

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**Pre**(*M1*, *M2*)
Bases: pyGPs.Core.cov.Kernel

Precomputed kernel matrix. No hyperparameters and thus nothing will be optimised.

Parameters

- **M1** – cross covariances matrix(train+1 by test). last row is self covariances (diagonal of test by test)
- **M2** – training set covariance matrix (train by train)

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**ProductOfKernel**(*cov1*, *cov2*)
Bases: pyGPs.Core.cov.Kernel

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

gethyp()

hyp

sethyp(*hyp*)

class pyGPs.Core.cov.**RBF**(*log_ell=0.0*, *log_sigma=0.0*)
Bases: pyGPs.Core.cov.Kernel

Squared Exponential kernel with isotropic distance measure. hyp = [log_ell, log_sigma]

Parameters

- **log_ell** – characteristic length scale.
- **log_sigma** – signal deviation.

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**RBFard**(*D=None*, *log_ell_list=None*, *log_sigma=0.0*)
Bases: pyGPs.Core.cov.Kernel

Squared Exponential kernel with Automatic Relevance Determination. hyp = log_ell_list + [log_sigma]

Parameters

- **D** – dimension of pattern. set if you want default ell, which is 1 for each dimension.
- **log_ell_list** – characteristic length scale for each dimension.
- **log_sigma** – signal deviation.

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**RBFunit**(*log_ell=0.0*)
Bases: pyGPs.Core.cov.Kernel

Squared Exponential kernel with isotropic distance measure with unit magnitude. i.e signal variance is always 1. hyp = [log_ell]

Parameters **log_ell** – characteristic length scale.

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**RQ**(*log_ell=0.0*, *log_sigma=0.0*, *log_alpha=0.0*)
Bases: pyGPs.Core.cov.Kernel

Rational Quadratic covariance function with isotropic distance measure. hyp = [log_ell, log_sigma, log_alpha]

Parameters

- **log_ell** – characteristic length scale.
- **log_sigma** – signal deviation.
- **log_alpha** – shape parameter for the RQ covariance.

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**RQard**(*D=None*, *log_ell_list=None*, *log_sigma=0.0*, *log_alpha=0.0*)
Bases: pyGPs.Core.cov.Kernel

Rational Quadratic covariance function with Automatic Relevance Detemination (ARD) distance measure. hyp = log_ell_list + [log_sigma, log_alpha]

Parameters

- **D** – dimension of pattern. set if you want default ell, which is 0.5 for each dimension.
- **log_ell_list** – characteristic length scale for each dimension.
- **log_sigma** – signal deviation.
- **log_alpha** – shape parameter for the RQ covariance.

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

class pyGPs.Core.cov.**ScaleOfKernel**(*cov*, *scalar*)
Bases: pyGPs.Core.cov.Kernel

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

gethyp()

hyp

sethyp(*hyp*)

class pyGPs.Core.cov.**SumOfKernel**(*cov1*, *cov2*)
Bases: pyGPs.Core.cov.Kernel

getCovMatrix(*x=None*, *z=None*, *mode=None*)

getDerMatrix(*x=None*, *z=None*, *mode=None*, *der=None*)

gethyp()

hyp

sethyp(*hyp*)

gp Module

class pyGPs.Core.gp.**GP**

Bases: object

Base class for GP model

fit(*x=None*, *y=None*, *der=True*)

fit the training data @return [nlZ, post] if der = False @return [nlZ, dnlZ, post] if der = True (default)

where nlZ is the negative log marginal likelihood dnlZ is partial derivatives of nlZ w.r.t. each hyperparameter post is struct representation of the (approximate) posterior post consists of post.alpha, post.L, post.sW

optimize(*x=None*, *y=None*)

train optimal hyperparameters adjust to all mean/cov/lik functions

plotData_1d(*axisvals=None*)

plotData_2d(*x1*, *x2*, *t1*, *t2*, *p1*, *p2*, *axisvals=None*)

predict(*xs*, *ys=None*)

prediction according to given inputs

@param xs test input @param ys test target(optional)

@return ymu, ys2, fmu, fs2, lp

where ymu is predictive output means ys2 is predictive output variances fm2 is predictive latent means fs2 is predictive latent variances lp is log predictive probabilities(if ys is given, otherwise is None)

predict_with_posterior(*post*, *xs*, *ys=None*)

prediction with provided posterior (i.e. you already have the posterior and thus don't need fitting/training phases)

@param post posterior structcture @param xs test input @param ys test target(optional)

@return ymu, ys2, fmu, fs2, lp

where ymu is predictive output means ys2 is predictive output variances fm2 is predictive latent means fs2 is predictive latent variances lp is log predictive probabilities(if ys is given, otherwise is None)

setData(*x*, *y*)

setOptimizer(*method*, *num_restarts=None*, *min_threshold=None*, *meanRange=None*, *covRange=None*, *likRange=None*)

This method is used to sepecify optimization configuration. By default, gp uses a single run "minimize".

Parameters

- **method** – Optimization methods. Possible values are:

  "Minimize" -> minimize by Carl Rasmussen (python implementation of "minimize" in GPML)

  "CG" -> conjugent gradient

  "BFGS" -> quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS)

  "SCG" -> scaled conjugent gradient (faster than CG)

- **num_restarts** – Set if you want to run mulitiple times of optimization with different initial guess. It specifys the maximum number of runs/restarts/trials.

- **min_threshold** – Set if you want to run mulitiple times of optimization with different initial guess. It specifys the threshold of objective function value. Stop optimization when this value is reached.
- **meanRange** – The range of initial guess for mean hyperparameters. e.g. meanRange = [(-2,2), (-5,5), (0,1)]. Each tuple specifys the range (low, high) of this hyperparameter, This is only the range of initial guess, during optimization process, optimal hyperparameters may go out of this range.

  (-5,5) for each hyperparameter by default.
- **covRange** – The range of initial guess for kernel hyperparameters. Usage see meanRange
- **likRange** – The range of initial guess for likelihood hyperparameters. Usage see meanRange

setPrior(*mean=None*, *kernel=None*)
set prior mean and cov

class pyGPs.Core.gp.**GPC**
Bases: pyGPs.Core.gp.GP

Gaussian Process Classification

plot(*x1*, *x2*, *t1*, *t2*, *axisvals=None*)

setOptimizer(*method*, *num_restarts=None*, *min_threshold=None*, *meanRange=None*, *covRange=None*, *likRange=None*)

useInference(*newInf*)

useLikelihood(*newLik*)

class pyGPs.Core.gp.**GPC_FITC**
Bases: pyGPs.Core.gp.GP_FITC

Gaussian Process Classification FITC

plot(*x1*, *x2*, *t1*, *t2*, *axisvals=None*)

setOptimizer(*method*, *num_restarts=None*, *min_threshold=None*, *meanRange=None*, *covRange=None*, *likRange=None*)

useInference(*newInf*)

useLaplace_FITC()
use Laplace approxiamation other than EP

useLikelihood(*newLik*)

class pyGPs.Core.gp.**GPMC**(*n_class*)
Bases: object

This is a one vs. one classification wrapper for GPC

createBinaryClass(*i*, *j*)
create data points x,y which only contains class i and j class_i is +1 and class_j is -1

fitAndPredict(*xs*)
predictive_vote is a matrix where row i -> each test point i column j -> probability for being eahc class j

optimizeAndPredict(*xs*)
predictive_vote is a matrix where row i -> each test point i column j -> probability for being eahc class j

setData(*x*, *y*)
for multi-class, data is x_all and y_all

setPrior(*mean=None*, *kernel=None*)
set prior mean and cov

useInference(*newInf*)

useLikelihood(*newLik*)

class pyGPs.Core.gp.**GPR**
Bases: pyGPs.Core.gp.GP

Gaussian Process Regression

plot(*axisvals=None*)

setNoise(*log_sigma*)
explicitly set noise variance other than default

setOptimizer(*method*, *num_restarts=None*, *min_threshold=None*, *meanRange=None*, *covRange=None*, *likRange=None*)

useInference(*newInf*)

useLikelihood(*newLik*)

class pyGPs.Core.gp.**GPR_FITC**
Bases: pyGPs.Core.gp.GP_FITC

Gaussian Process Regression FITC

plot(*axisvals=None*)

setNoise(*log_sigma*)
explicitly set noise variance other than default

setOptimizer(*method*, *num_restarts=None*, *min_threshold=None*, *meanRange=None*, *covRange=None*, *likRange=None*)

useInference(*newInf*)

useLikelihood(*newLik*)

class pyGPs.Core.gp.**GP_FITC**
Bases: pyGPs.Core.gp.GP

GP_FITC base class

setData(*x*, *y*, *value_per_axis=5*)
set Data and derive deault inducing_points

value_per_axis is number of value in each dimension... ...when using a default inducing point grid

setPrior(*mean=None*, *kernel=None*, *inducing_points=None*)
set prior and inducing_points

inf Module

class pyGPs.Core.inf.**EP**
Bases: pyGPs.Core.inf.Inference

Expectation Propagation approximation to the posterior Gaussian Process.

proceed(*meanfunc*, *covfunc*, *likfunc*, *x*, *y*, *nargout=1*)

class pyGPs.Core.inf.**Exact**
Bases: pyGPs.Core.inf.Inference

Exact inference for a GP with Gaussian likelihood. Compute a parametrization of the posterior, the negative log marginal likelihood and its derivatives w.r.t. the hyperparameters.

proceed(*meanfunc*, *covfunc*, *likfunc*, *x*, *y*, *nargout=1*)

class pyGPs.Core.inf.**FITC_EP**
Bases: pyGPs.Core.inf.Inference

FITC-EP approximation to the posterior Gaussian process. The function is equivalent to infEP with the covariance function: Kt = Q + G; G = diag(g); g = diag(K-Q); Q = Ku' * inv(Kuu + snu2 * eye(nu)) * Ku; where Ku and Kuu are covariances w.r.t. to inducing inputs xu and snu2 = sn2/1e6 is the noise of the inducing inputs. We fixed the standard deviation of the inducing inputs snu to be a one per mil of the measurement noise's standard deviation sn. In case of a likelihood without noise parameter sn2, we simply use snu2 = 1e-6. For details, see The Generalized FITC Approximation, Andrew Naish-Guzman and Sean Holden, NIPS, 2007.

proceed(*meanfunc*, *covfunc*, *likfunc*, *x*, *y*, *nargout=1*)

class pyGPs.Core.inf.**FITC_Exact**
Bases: pyGPs.Core.inf.Inference

FITC approximation to the posterior Gaussian process. The function is equivalent to infExact with the covariance function: Kt = Q + G; G = diag(g); g = diag(K-Q); Q = Ku' * inv(Quu) * Ku; where Ku and Kuu are covariances w.r.t. to inducing inputs xu, snu2 = sn2/1e6 is the noise of the inducing inputs and Quu = Kuu + snu2*eye(nu).

proceed(*meanfunc*, *covfunc*, *likfunc*, *x*, *y*, *nargout=1*)

class pyGPs.Core.inf.**FITC_Laplace**
Bases: pyGPs.Core.inf.Inference

FITC-Laplace approximation to the posterior Gaussian process. The function is equivalent to infLaplace with the covariance function: Kt = Q + G; G = diag(g); g = diag(K-Q); Q = Ku' * inv(Kuu + snu2 * eye(nu)) * Ku; where Ku and Kuu are covariances w.r.t. to inducing inputs xu and snu2 = sn2/1e6 is the noise of the inducing inputs. We fixed the standard deviation of the inducing inputs snu to be a one per mil of the measurement noise's standard deviation sn. In case of a likelihood without noise parameter sn2, we simply use snu2 = 1e-6.

proceed(*meanfunc*, *covfunc*, *likfunc*, *x*, *y*, *nargout=1*)

class pyGPs.Core.inf.**Inference**
Bases: object

Base class for inference. Defined several tool methods in it.

Psi_line(*s*, *dalpha*, *alpha*, *K*, *m*, *likfunc*, *y*, *inffunc*)

Psi_lineFITC(*s*, *dalpha*, *alpha*, *V*, *d0*, *m*, *likfunc*, *y*, *inffunc*)

epComputeParams(*K*, *y*, *ttau*, *tnu*, *likfunc*, *m*, *inffunc*)

epfitcRefresh(*d0*, *P0*, *R0*, *R0P0*, *w*, *b*)

epfitcUpdate(*d*, *P_i*, *R*, *nn*, *gg*, *w*, *b*, *ii*, *w_i*, *b_i*, *m*, *d0*, *P0*, *R0*)

epfitcZ(*d*, *P*, *R*, *nn*, *gg*, *ttau*, *tnu*, *d0*, *R0*, *P0*, *y*, *likfunc*, *m*, *inffunc*)

fitcRefresh(*d0*, *P0*, *R0*, *R0P0*, *w*)

logdetA(*K*, *w*, *nargout*)

mvmK(*al*, *V*, *d0*)

mvmZ(*x*, *RVdd*, *t*)

proceed()

class pyGPs.Core.inf.**Laplace**
Bases: pyGPs.Core.inf.Inference

Laplace's Approximation to the posterior Gaussian process.

proceed(*meanfunc*, *covfunc*, *likfunc*, *x*, *y*, *nargout=1*)

class pyGPs.Core.inf.**dnlZStruct**(*m*, *c*, *l*)
Bases: object

class pyGPs.Core.inf.**postStruct**
Bases: object

lik Module

class pyGPs.Core.lik.**Erf**
Bases: pyGPs.Core.lik.Likelihood

Error function or cumulative Gaussian likelihood function for binary classification or probit regression.

$Erf(t) = \frac{1}{2}(1 + erf(\frac{t}{\sqrt{2}})) = normcdf(t)$

cumGauss(*y=None*, *f=None*, *nargout=1*)

gauOverCumGauss(*f*, *p*)

logphi(*z*, *p*)

proceed(*y=None*, *mu=None*, *s2=None*, *inffunc=None*, *der=None*, *nargout=1*)

class pyGPs.Core.lik.**Gauss**(*log_sigma=-2.3025850929940455*)
Bases: pyGPs.Core.lik.Likelihood

Gaussian likelihood function for regression.

$Gauss(t) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t-y)^2}{2\sigma^2}}$, where y is the mean and σ is the standard deviation.

hyp = [log_sigma]

proceed(*y=None*, *mu=None*, *s2=None*, *inffunc=None*, *der=None*, *nargout=1*)

class pyGPs.Core.lik.**Laplace**(*log_sigma=-2.3025850929940455*)
Bases: pyGPs.Core.lik.Likelihood

Laplacian likelihood function for regression. ONLY works with EP inference!

$Laplace(t) = \frac{1}{2b} e^{-\frac{|t-y|}{b}}$ where $b = \frac{\sigma}{\sqrt{2}}$, y is the mean and σ is the standard deviation.

hyp = [log_sigma]

proceed(*y=None*, *mu=None*, *s2=None*, *inffunc=None*, *der=None*, *nargout=1*)

class pyGPs.Core.lik.**Likelihood**
Bases: object

Base function for Likelihood function

proceed()

mean Module

class pyGPs.Core.mean.**Const**(*c=5.0*)
Bases: pyGPs.Core.mean.Mean

Constant mean function. hyp = [c]

Parameters **c** – constant value for mean

getDerMatrix(*x=None*, *der=None*)

getMean(*x=None*)

class pyGPs.Core.mean.**Linear**(*D=None*, *alpha_list=None*)
Bases: pyGPs.Core.mean.Mean

Linear mean function. self.hyp = alpha_list

Parameters **D** – dimension of training data. Set if you want default alpha, which is 0.5 for each dimension.

Alpha_list scalar alpha for each dimension

getDerMatrix(*x=None*, *der=None*)

getMean(*x=None*)

class pyGPs.Core.mean.**Mean**
Bases: object

The base function for mean function

class pyGPs.Core.mean.**One**
Bases: pyGPs.Core.mean.Mean

One mean.

getDerMatrix(*x=None*, *der=None*)

getMean(*x=None*)

class pyGPs.Core.mean.**PowerOfMean**(*mean*, *d*)
Bases: pyGPs.Core.mean.Mean

getDerMatrix(*x=None*, *der=None*)

getMean(*x=None*)

gethyp()

hyp

sethyp(*hyp*)

class pyGPs.Core.mean.**ProductOfMean**(*mean1*, *mean2*)
Bases: pyGPs.Core.mean.Mean

getDerMatrix(*x=None*, *der=None*)

getMean(*x=None*)

gethyp()

hyp

sethyp(*hyp*)

class pyGPs.Core.mean.**ScaleOfMean**(*mean*, *scalar*)
Bases: pyGPs.Core.mean.Mean

getDerMatrix(*x=None*, *der=None*)

getMean(*x=None*)

gethyp()

hyp

sethyp(*hyp*)

class pyGPs.Core.mean.**SumOfMean**(*mean1*, *mean2*)
Bases: pyGPs.Core.mean.Mean

getDerMatrix (*x=None*, *der=None*)

getMean (*x=None*)

gethyp ()

hyp

sethyp (*hyp*)

class pyGPs.Core.mean.**Zero**
Bases: pyGPs.Core.mean.Mean

Zero mean.

getDerMatrix (*x=None*, *der=None*)

getMean (*x=None*)

opt Module

class pyGPs.Core.opt.**BFGS** (*model*, *searchConfig=None*)
Bases: pyGPs.Core.opt.Optimizer

findMin (*x*, *y*)

class pyGPs.Core.opt.**CG** (*model*, *searchConfig=None*)
Bases: pyGPs.Core.opt.Optimizer

findMin (*x*, *y*)

class pyGPs.Core.opt.**Minimize** (*model*, *searchConfig=None*)
Bases: pyGPs.Core.opt.Optimizer

findMin (*x*, *y*)

class pyGPs.Core.opt.**Optimizer**
Bases: object

apply_in_objects (*hypInArray*)

convert_to_array ()

dnlml (*hypInArray*)

nlml (*hypInArray*)

nlzAnddnlz (*hypInArray*)

class pyGPs.Core.opt.**SCG** (*model*, *searchConfig=None*)
Bases: pyGPs.Core.opt.Optimizer

findMin (*x*, *y*)

tools Module

pyGPs.Core.tools.**brentmin** (*xlow*, *xupp*, *Nitmax*, *tol*, *f*, *nout=None*, **args*)
pyGPs.Core.tools.**cholupdate** (*R*, *x*, *sgn='+'*)

pyGPs.Core.tools.**solve_chol** (*L*, *B*)

pyGPs.Core.tools.**unique** (*x*)

GraphExtension Package

GraphExtension Package

graphKernels Module Created on April, 2014

@author: marion, shan

pyGPs.GraphExtension.graphKernels.**propagationKernel**(*A*, *l*, *gr_id*, *h_max*, *w*, *p*, *ktype=None*, *SUM=True*, *VIS=False*, *showEachStep=False*)

INPUT: A adjacency matrix (num_nodes x num_nodes) l label array (num_nodes x 1); values [1,...,k] or -1 for unlabeled nodes

...OR label array (num_nodes x num_labels); values [0,1], unlabeled nodes have only 0 entries

gr_id graph indicator array (num_nodes x 1); values [0,..,n] h_max number of iterations w bin widths parameter p distance ('tv', 'hellinger', 'L1', 'L2') ktype type of propagation kernel ['diffusion', 'label_propagation', 'label_spreading', 'belief_propagation']

RETURN: K kernel matrix

ToDO: SPLIT Varible label array and label probability

CAUTION: number of labels (k) > 1 !!!

graphUtil Module

pyGPs.GraphExtension.graphUtil.**formKernelMatrix**(*M*, *indice_train*, *indice_test*)

format precomputed kernel matrix into two matrix, which fit the parameters of method in pyGP

INPUT: M n by n precomputed kernel matrix indice_train list of indice of training examples indice_test list of indice of test examples

OUTPUT: M1 train+1 by test matrix

where the last row is the diagonal of test-test covariance

M2 train by train matrix

pyGPs.GraphExtension.graphUtil.**formKnnGraph**(*pc*, *k*)

INPUT: pc n by D data matrix where n is num_points

D is num_dimensions

OUTPUT: A adjacency matrix

pyGPs.GraphExtension.graphUtil.**normalizeKernel**(*K*)

INPUT: K n by D kernel matrix

OUTPUT: K_norm n by D normalized kernel matrix(correlation matrix)

nodeKernels Module

pyGPs.GraphExtension.nodeKernels.**VNDKernel**(*A*, *alpha=0.5*)

Von Neumann Diffusion Kernel on graph (Zhou et al., 2004) (also label spreading kernel)

K = (I - alpha*S)^-1, where S = D^-1/2*A*D^-1/2

Parameters

- **A** – adjacency matrix
- **alpha** – (hyper)parameter(s)

pyGPs.GraphExtension.nodeKernels.**cosKernel**(*A*)

Cosine Kernel (also Inverse Cosine Kernel)

K = cos (L*pi/4), where L is the normalized Laplacian

Parameters **A** – adjacency matrix

pyGPs.GraphExtension.nodeKernels.**diffKernel**(*A*, *beta=0.5*)
Diffusion Process Kernel

K = exp(beta * H), where H = -L = A-D

K = Q exp(beta * Lambda) Q.T

Parameters

- **A** – adjacency matrix
- **beta** – (hyper)parameter(s)

pyGPs.GraphExtension.nodeKernels.**normLap**(*A*)
normalized Laplacian

pyGPs.GraphExtension.nodeKernels.**normalizeKernel**(*K*)

pyGPs.GraphExtension.nodeKernels.**psInvLapKernel**(*A*)
Pseudo inverse of the normalized Laplacian.

Parameters **A** – adjacency matrix

pyGPs.GraphExtension.nodeKernels.**regLapKernel**(*A*, *sigma=1*)
Regularized Laplacian Kernel

Parameters

- **A** – adjacency matrix
- **sigma** – (hyper)parameter(s)

pyGPs.GraphExtension.nodeKernels.**rwKernel**(*A*, *p=1*, *a=2*)
p-step Random Walk Kernel with a>1

K = (aI-L)^p, p>1 and L is the normalized Laplacian

Parameters

- **A** – adjacency matrix
- **p** – step parameter
- **a** – (hyper)parameter(s)

Optimization Package

Optimization Package

conf Module

class pyGPs.Optimization.conf.**random_init_conf**(*mean*, *cov*, *lik*)
Bases: object

covRange

getcr()

getlr()

getmr()

likRange

meanRange

setcr(*value*)

setlr(*value*)

setmr(*value*)

minimize Module This module contains a function that performs unconstrained gradient based optimization using nonlinear conjugate gradients.

The function is a straightforward Python-translation of Carl Rasmussen's Matlab-function minimize.m

pyGPs.Optimization.minimize.**run**(*f*, *X*, *args=()*, *length=None*, *red=1.0*, *verbose=False*)

scg Module

pyGPs.Optimization.scg.**run**(*f*, *x*, *args=()*, *niters=100*, *gradcheck=False*, *display=0*, *flog=False*, *pointlog=False*, *scalelog=False*, *tolX=1e-08*, *tolO=1e-08*, *eval=None*)

Valid Package

Valid Package

valid Module

pyGPs.Valid.valid.**ACC**(*predict*, *target*)
Classification accuracy

pyGPs.Valid.valid.**NLPD**(*y*, *MU*, *S2*)
Calculate evaluation measure NLPD in transformed observation space.

INPUT y observed targets MU vector of predictions/predicted means S2 vector of 'self' variances

OUTPUT nlpd Negative Log Predictive Density.

pyGPs.Valid.valid.**Prec**(*predict*, *target*)
Precision for class +1

pyGPs.Valid.valid.**RMSE**(*predict*, *target*)
Root mean squared error

pyGPs.Valid.valid.**Recall**(*predict*, *target*)
Recall for class +1

pyGPs.Valid.valid.**k_fold_index**(*n*, *K=10*)
Similar to k_fold_validation, but only return the indice instead of data

n is number of instances of data K is number of folder

pyGPs.Valid.valid.**k_fold_validation**(*x*, *y*, *K=10*, *randomise=False*)
Generates K (training, validation) pairs from the items in X.

The validation iterables are a partition of X, and each validation iterable is of length len(X)/K. Each training iterable is the complement (within X) of the validation iterable, and so each training iterable is of length (K-1)*len(X)/K.

Validation Package

Validation Package

valid Module

pyGPs.Validation.valid.**ACC**(*predict*, *target*)
Classification accuracy

pyGPs.Validation.valid.**NLPD**(*y*, *MU*, *S2*)
Calculate evaluation measure NLPD in transformed observation space.

INPUT y observed targets MU vector of predictions/predicted means S2 vector of 'self' variances

OUTPUT nlpd Negative Log Predictive Density.

pyGPs.Validation.valid.**Prec**(*predict*, *target*)
Precision for class +1

pyGPs.Validation.valid.**RMSE**(*predict*, *target*)
Root mean squared error

pyGPs.Validation.valid.**Recall**(*predict*, *target*)
Recall for class +1

pyGPs.Validation.valid.**k_fold_index**(*n*, *K=10*)
Similar to k_fold_validation, but only return the indice instead of data

n is number of instances of data K is number of folder

pyGPs.Validation.valid.**k_fold_validation**(*x*, *y*, *K=10*, *randomise=False*)
Generates K (training, validation) pairs from the items in X.

The validation iterables are a partition of X, and each validation iterable is of length len(X)/K. Each training iterable is the complement (within X) of the validation iterable, and so each training iterable is of length (K-1)*len(X)/K.

PYTHON MODULE INDEX

p

INDEX

H

I

K

L

S

U

V

Z