gp_emulator Documentation

Release 1.6.10

J Gomez-Dans

Feb 21, 2019

Contents:

The *gp_emulator* library provides a simple pure Python implementations of Gaussian Processes (GPs), with a view of using them as **emulators** of complex computers code. In particular, the library is focused on radiative transfer models for remote sensing, although the use is general. The GPs can also be used as a way of regressing or interpolating datasets.

If you use this code, please cite both the code and the paper that describes it.

- JL Gómez-Dans, Lewis PE, Disney M. Efficient Emulation of Radiative Transfer Codes Using Gaussian Processes and Application to Land Surface Parameter Inferences. Remote Sensing. 2016; 8(2):119. DOI:10.3390/rs8020119
- José Gómez-Dans & Professor Philip Lewis. (2018, October 12). jgomezdans/gp_emulator (Version 1.6.5). Zenodo. DOI:10.5281/zenodo.1460970

Development of this code has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 687320, under project H2020 MULTIPLY.

CHAPTER 1

Gaussian Process Emulators

1.1 Introduction

Often, complex and numerically demanding computer codes are required in inverse modelling tasks. Such models might need to be invoked repeatedly as part of a minimisition task, or in order to provide some numerically-integrated quantity. This results in many applications being rendered impractical as the codes are too slow.

The concept of an emulator is simple: for a given model, let's provide a function that given the same inputs are the original model, gives *the same* output. Clearly, we need to qualify "the same", and maybe downgrade the expectations to "a very similar output". Ideally, with some metric of uncertainty on the prediction. So an emulator is just a fast, surrogate to a more established code.

Gaussian processes (GPs) have been used for this task for years, as they're very flexible throught he choice of covariance function that can be used, but also work remarkably well with models that are nonlinear and that have a reasonable number of inputs (10s).

We have used these techniques for emulation radiative transfer models used in Earth Observation, and we even wrote a nice paper about it: Gomez-Dans et al (2016). Read it, it's pure gold.

1.2 Installing the package

The package works on Python 3. With a bit of effort it'll probably work on Python 2.7. The only dependencies are scipy and numpy To install, use either **conda**:

```
conda install -c jgomezdans gp_emulator
```

or **pip**:

```
pip install gp_emulator
```

or just clone or download the source repository and invoke *setup.py* script:

```
python setup.py install
```

CHAPTER 2

Quickstart

2.1 Single output model emulation

Assume that we have two arrays, `X` and `y`. `y` is of size `N`, and it stores the `N` expensive model outputs that have been produced by running the model on the `N` input sets of `M` input parameters in `X`. We will try to emulate the model by learning from these two training sets:

```
gp = gp_emulator.GaussianProcess(inputs=X, targets=y)
```

Now, we need to actually do the training...

```
gp.learn_hyperparameters()
```

Once this process has been done, you're free to use the emulator to predict the model output for an arbitrary test vector `x_test` (size `M`):

```
y_pred, y_sigma, y_grad = gp.predict (x_test, do_unc=True,
                                      do_grad=True)
```

In this case, `y_pred` is the model prediction, `y_sigma` is the variance associated with the prediction (the uncertainty) and `y_grad` is an approximation to the Jacobian of the model around `x_test`.

Let's see a more concrete example. We create a damped sine, add a bit of Gaussian noise, and then subsample a few points (10 in this case), fit the GP, and predict the function over the entire range. We also plot the uncertainty from this prediction.

```
import numpy as np
import matplotlib.pyplot as plt

import gp_emulator

np.random.seed(42)
n_samples = 2000
x = np.linspace(0, 2, n_samples)
```

(continues on next page)

(continued from previous page)

```
y = np.exp(-0.7*x)*np.sin(2*np.pi*x/0.9)
y += np.random.randn(n_samples)*0.02

# Select a few random samples from x and y
isel = np.random.choice(n_samples, 10)
x_train = np.atleast_2d(x[isel]).T
y_train = y[isel]
fig = plt.figure(figsize=(12,4))

gp = gp_emulator.GaussianProcess(x_train, y_train)
gp.learn_hyperparameters(n_tries=25)

y_pred, y_unc, _ = gp.predict(np.atleast_2d(x).T,
                              do_unc=True, do_deriv=False)
plt.plot(x, y_pred, '-', lw=2., label="Predicted")
plt.plot(x, np.exp(-0.7*x)*np.sin(2*np.pi*x/0.9), '-', label="True")
plt.fill_between(x, y_pred-1.96*y_unc,
                    y_pred+1.96*y_unc, color="0.8")
plt.legend(loc="best")
```

We can see that the GP is doing an excellent job in predicting the function, even in the presence of noise, and with a handful of sample points. In situations where there is extrapolation, this is indicated by an increase in the predictive uncertainty.

2.2 Multiple output emulators

In some cases, we can emulate multiple outputs from a model. For example, hyperspectral data used in EO can be emulated by employing the SVD trick and emulating the individual principal component weights. Again, we use `X` and `y`. `y` is now of size `N, P`, and it stores the `N` expensive model outputs (size `P`) that have been produced by running the model on the `N` input sets of `M` input parameters in `X`. We will try to emulate the model by learning from these two training sets, but we need to select a variance level for the initial PCA (in this case, 99%)

```
gp = gp_emulator.MultivariateEmulator (X=y, y=X, thresh=0.99)
```

Now, we're ready to use on a new point `x_test` as above:

```
y_pred, y_sigma, y_grad = gp.predict (x_test, do_unc=True,
                                      do_grad=True)
```

A more concrete example: let's produce a signal that can be decomposed as a sum of scaled orthogonal basis functions...

```
import numpy as np

from scipy.fftpack import dct

import matplotlib.pyplot as plt
import gp_emulator


np.random.seed(1)

n_validate = 250
n_train = 100
```

(continues on next page)

(continued from previous page)

```
basis_functions = dct(np.eye(128), norm="ortho")[:, 1:4]

params=["w1", "w2", "w3"]
mins = [-1, -1, -1]
maxs = [1, 1, 1]


train_weights, dists = gp_emulator.create_training_set(params, mins, maxs,
                                                       n_train=n_train)
validation_weights = gp_emulator.create_validation_set(dists,
                                                     n_validate=n_validate)

training_set = (train_weights@basis_functions.T).T

training_set += np.random.randn(*training_set.shape)*0.0005
validation_set = (validation_weights@basis_functions.T).T

gp = gp_emulator.MultivariateEmulator (y=train_weights, X=training_set.T,
                                       thresh=0.973, n_tries=25)
y_pred = np.array([gp.predict(validation_weights[i])[0]
                        for i in range(n_validate)])

fig, axs = plt.subplots(nrows=1, ncols=2,sharey=True,figsize=(12, 4))
axs[0].plot(validation_set[:, ::25])
axs[1].plot(10.*(y_pred.T - validation_set))
axs[0].set_title("Samples from validation dataset")
axs[1].set_title("10*Mismatch between validation simulator and emulator")
```

CHAPTER 3

Emulating a typical radiative transfer model

This package was designed to emulate radiatie transfer models. The process entails the following steps:

1. Decide on what input parameters are required
2. Decide their ranges
3. Generate a training input parameter set
4. Run the model for each element of the input training set and store the outputs
5. Pass the input and output training pairs to the library, and let it fit the hyperparameters

We can show how this works with an example of the PROSPECT+SAIL model.

3.1 Setting the input parameter ranges

We can set the parameter names and their ranges simply by having lists with minimum and maximum values. This assumes a uniformly-distributed parameter distribution between those two boundaries, but other distributions are possible (we never had any reason to try them though!). We additionally set up the SRFs and other variables that need to be defined here... We train the model on 250 samples and test on (say) 100. 100 validation samples is probably too few, but for the sake of not waiting too much... ;-)

```
from functools import partial

import numpy as np

import gp_emulator

import prosail

# Spectral band definition. Just a top hat, with start and
# end wavlengths as an example
b_min = np.array( [ 620., 841, 459, 545, 1230, 1628, 2105] )
b_max = np.array( [ 670., 876, 479, 565, 1250, 1652, 2155] )
```

(continues on next page)

(continued from previous page)

```
wv = np.arange ( 400, 2501 )
passband = []

# Number of training and validation samples
n_train = 250
n_validate = 100
# Validation number is small, increase to a more realistic value
# if you want

# Define the parameter names and their ranges
# Note that we are working here in transformed coordinates...

# Define geometry. Each emulator is for one geometry
sza = 30.
vza = 0.
raa = 0. # in degrees

parameters = [ 'n', 'cab', 'car', 'cbrown', 'cw', 'cm', 'lai', 'ala', 'bsoil', 'psoil
↪']
min_vals = [ 0.8       ,  0.46301307,  0.95122942,  0.        ,  0.02829699,
        0.03651617,  0.04978707,  0.44444444,  0.        ,  0.]
max_vals = [ 2.5       ,  0.998002  ,  1.        ,  1.        ,  0.80654144,
        0.84366482,  0.99501248,  0.55555556,  2.    , 1     ]
```

We then require a function for calling the RT model. In the case of PROSAIL, we can do that easily from Python, in other models available in e.g. Fortran, you could have a function that calls the external model

```
def inverse_transform ( x ):
    """Inverse transform the PROSAIL parameters"""
    x_out = x*1.
    # Cab, posn 1
    x_out[1] = -100.*np.log ( x[1] )
    # Cab, posn 2
    x_out[2] = -100.*np.log ( x[2] )
    # Cw, posn 4
    x_out[4] = (-1./50.)*np.log ( x[4] )
    #Cm, posn 5
    x_out[5] = (-1./100.)*np.log ( x[5] )
    # LAI, posn 6
    x_out[6] = -2.*np.log ( x[6] )
    # ALA, posn 7
    x_out[7] = 90.*x[7]
    return x_out



def rt_model ( x, passband=None, do_trans=True ):
    """A coupled land surface/atmospheric model, predicting refl from
    land surface parameters. Thisfunction provides estimates of refl for
    a particular illumination geometry.

    The underlying land surface reflectance spectra is simulated using
    PROSAIL. The input parameter ``x`` is a vector with the following components:

        * ``n``
        * ``cab``
```

(continues on next page)

(continued from previous page)

```
        * ``car``
        * ``cbrown``
        * ``cw``
        * ``cm``
        * ``lai``
        * ``ala``
        * ``bsoil``
        * ``psoil``

    """
    x, sza, vza, raa = x

    # Invert parameter LAI
    if do_trans:
        x = inverse_transform ( x )
    ################# surface refl with prosail ####################

    surf_refl = prosail.run_prosail(x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7], 0.
↪01, sza, vza, raa,
                    rsoil=x[8], psoil=x[9])
    if passband is None:
        return surf_refl
    else:
        return surf_refl[passband].mean()
```

Now we loop over all the bands, and prepare the emulators. we do this by using the *create_emulator_validation* function, that does everything you'd want to do... We just stuff the emulator, training and validation sets in one list for convenience.

```
retval = []
for iband,bmin in enumerate ( b_min ):
    # Looping over the bands....
    print("Doing band %d" % (iband+1))
    passband = np.nonzero( np.logical_and ( wv >= bmin, wv <= b_max[iband] ) )
    # Define the SRF for the current band
    # Define the simulator for convenience
    simulator = partial (rt_model, passband=passband)
    # Actually create the training and validation parameter sets, train the emulators
    # and return all that
    x = gp_emulator.create_emulator_validation (simulator, parameters, min_vals, max_
↪vals,
                                n_train, n_validate, do_gradient=True,
                                n_tries=15, args=(30, 0, 0) )
    retval.append (x)
```

A simple validation visualisation looks like this

3.2 An spectral emulator of PROSAIL

For the case of a spectral emulator, the approach is the same, only that we just use the spectral emulator, which is a bit simpler.

```
n_train = 350
n_validate = 100
```

(continues on next page)

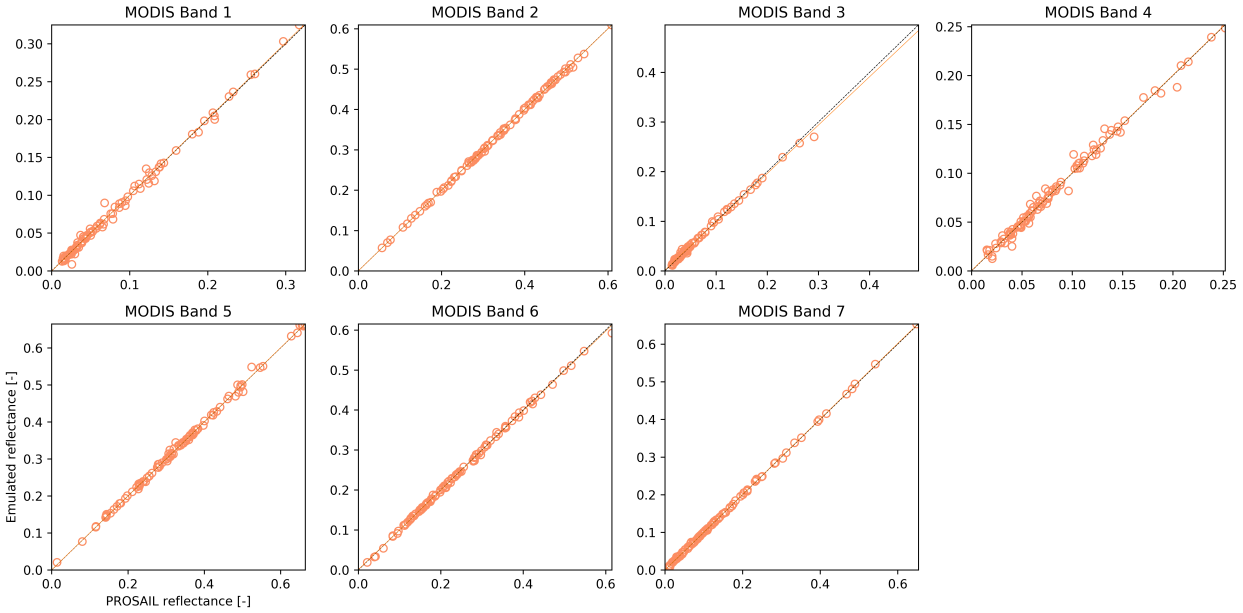

Fig. 1: Comparison between the simulated output and the corresponding emulator output for the validation dataset. Correlations (R2) are in all cases better than 0.99. Slope was between 0.97 and 1., whereas the bias term was smaller than 0.002.

(continued from previous page)

```
x = gp_emulator.create_emulator_validation ( rt_model, parameters, min_vals, max_vals,
                                    n_train, n_validate, do_gradient=True,
                                    n_tries=10, args=(30, 0, 0) )
```

The validation results looks like this:

We can also check that the gradient of the model is sensible, by comparing it with finite difference approximations from the original model, which is already carried out by *create_emulator_validation* if we set the *do_gradient* option.

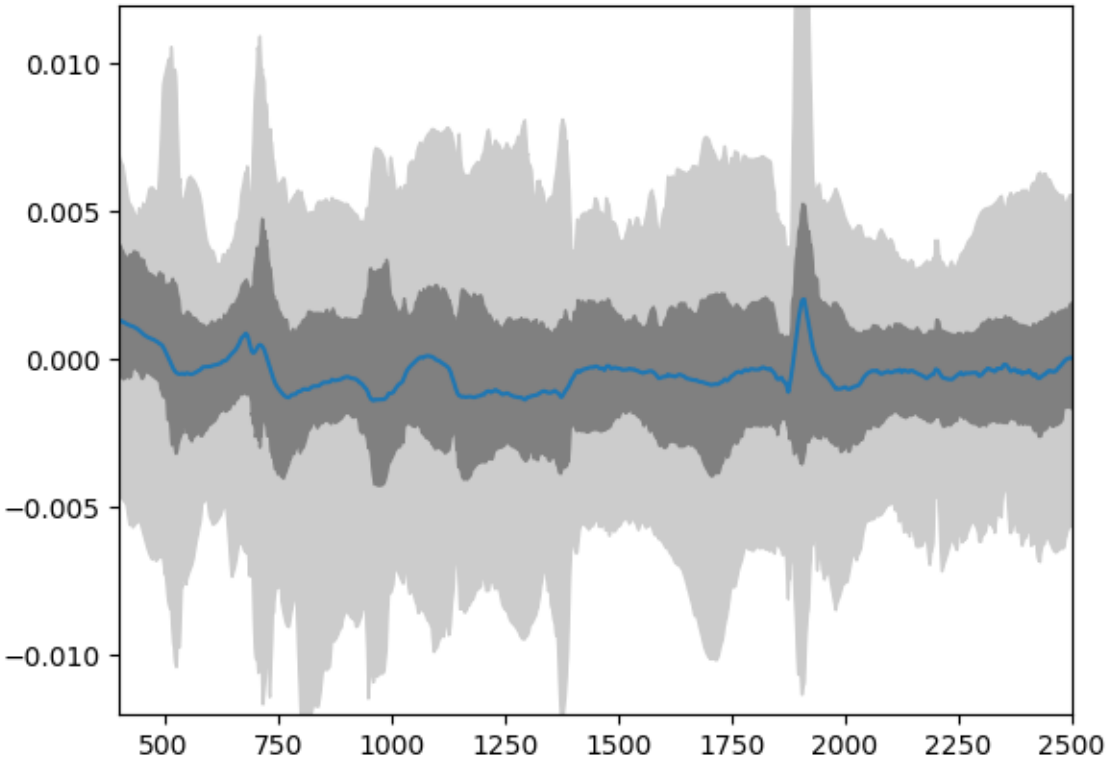

Fig. 2: Distribution of residuals derived from the difference of the emulator and simulator for PROSAIL.

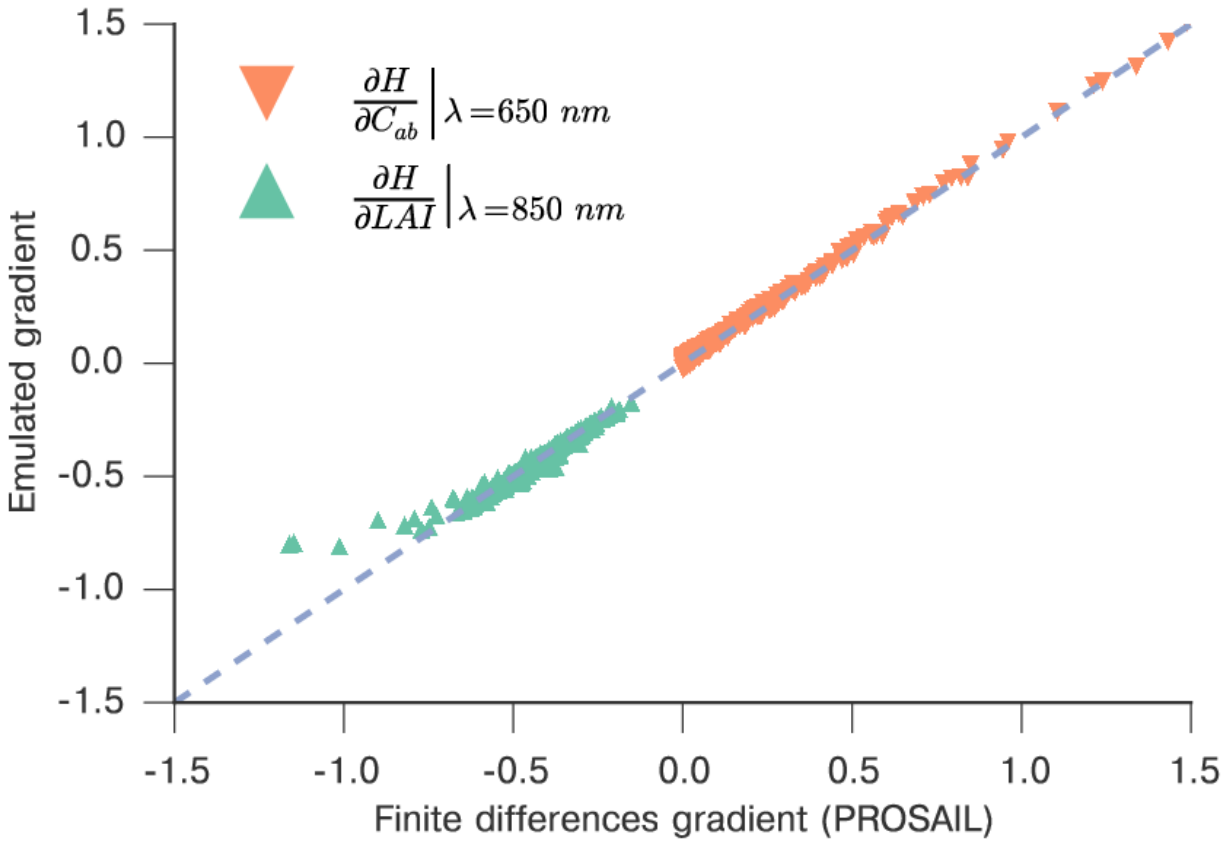

Fig. 3: Comparison of the emulated model gradient versus finite difference approximation for LAI and chlorophyll at different spectral regions.

CHAPTER 4

GPs as regressors for biophyiscal parameter inversion

GPs are a general regression technique, and can be used to regress some wanted magnitude from a set of inputs. This isn't as cool as other things you can do with them, but it's feasible to do... GPs are flexible for regression and interpolation, but given that this library has a strong remote sensing orientation, we'll consider their use for bionpysical parameter extraction from Sentinel-2 data (for example).

4.1 Retrieving bionpysical parameters for Sentinel-2

Let's assume that we want to retrieve leaf area index (LAI) from Sentinel-2 surface reflectance data. The regression problem can be stated as one where the inputs to the regressor are the spectral measurements of a pixel, and the output is the retrieved LAI. We can do this mapping by pairing in situ measurements, or we can just use a standard RT model to provide the direct mapping, and then learn the inverse mapping using the GP.

Although the problem is easy, we know that other parameters will have an effect in the measuremed reflectance, so we can only expect this to work over a limited spread of parameters other than LAI. Here, we show how to use the *gp_emulator* helper functions to create a suitable training set, and perform this.

```
import numpy as np

import scipy.stats

import gp_emulator
import prosail

import matplotlib.pyplot as plt

np.random.seed(42)
# Define number of training and validation samples
n_train = 200
n_validate = 500
# Define the parameters and their spread
parameters = ["n", "cab", "car", "cbrown", "cw", "cm", "lai", "ala"]
p_mins =     [1.6, 25,    5,     0.0,     0.01, 0.01, 0.,   32.]
```

(continues on next page)

(continued from previous page)

```
p_maxs =     [2.1, 90,    20,    0.4,     0.014, 0.016, 7.,   57.]

# Create the training samples
training_samples, distributions = gp_emulator.create_training_set(parameters, p_mins, p_maxs,
                        n_train=n_train)
# Create the validation samples
validation_samples = gp_emulator.create_validation_set(distributions, n_validate=n_validate)

# Load up the spectral response functions for S2
srf = np.loadtxt("S2A_SRS.csv", skiprows=1,
                 delimiter=",")[100:, :]
srf[:, 1:] = srf[:, 1:]/np.sum(srf[:, 1:], axis=0)
srf_land = srf[:, [ 2, 3, 4, 5, 6, 7, 8, 9, 12, 13]].T

# Generate the reflectance training set by running the RT model
# for each entry in the training set, and then applying the
# spectral basis functions.
training_s2 = np.zeros((n_train, 10))
for i, p in enumerate(training_samples):
    refl = prosail.run_prosail (p[0], p[1], p[2], p[3], p[4], p[5], p[6], p[7],
                                0.001, 30., 0, 0, prospect_version="D",
                                rsoil=0., psoil=0, rsoil0=np.zeros(2101))
    training_s2[i, :] = np.sum(refl*srf_land, axis=-1)

# Generate the reflectance validation set by running the RT model
# for each entry in the validation set, and then applying the
# spectral basis functions.
validation_s2 = np.zeros((n_validate, 10))
for i, p in enumerate(validation_samples):
    refl = prosail.run_prosail (p[0], p[1], p[2], p[3], p[4], p[5], p[6], p[7],
                                0.001, 30., 0, 0, prospect_version="D",
                                rsoil=0., psoil=0, rsoil0=np.zeros(2101))
    validation_s2[i, :] = np.sum(refl*srf_land, axis=-1)

# Define and train the emulator from reflectance to LAI
gp = gp_emulator.GaussianProcess(inputs=training_s2, targets=training_samples[:, 6])
gp.learn_hyperparameters(n_tries=15, verbose=False)

# Predict the LAI from the reflectance
ypred, _, _ = gp.predict(validation_s2)

# Plot
fig = plt.figure(figsize=(7,7))
plt.plot(validation_samples[:, 6], ypred, 'o', mfc="none")
plt.plot([p_mins[6], p_maxs[6]], [p_mins[6], p_maxs[6]],
        '--', lw=3)
x = np.linspace(p_mins[6], p_maxs[6], 100)

regress = scipy.stats.linregress(validation_samples[:, 6], ypred)
plt.plot(x, regress.slope*x + regress.intercept, '-')
plt.xlabel(r"Validation LAI $[m^{2}m^{-2}]$")
plt.ylabel(r"Retrieved LAI $[m^{2}m^{-2}]$")
plt.title("Slope=%8.4f, "%(regress.slope) +
          "Intercept=%8.4f, "%(regress.intercept) +
          "$R^2$=%8.3f" % (regress.rvalue**2))
```

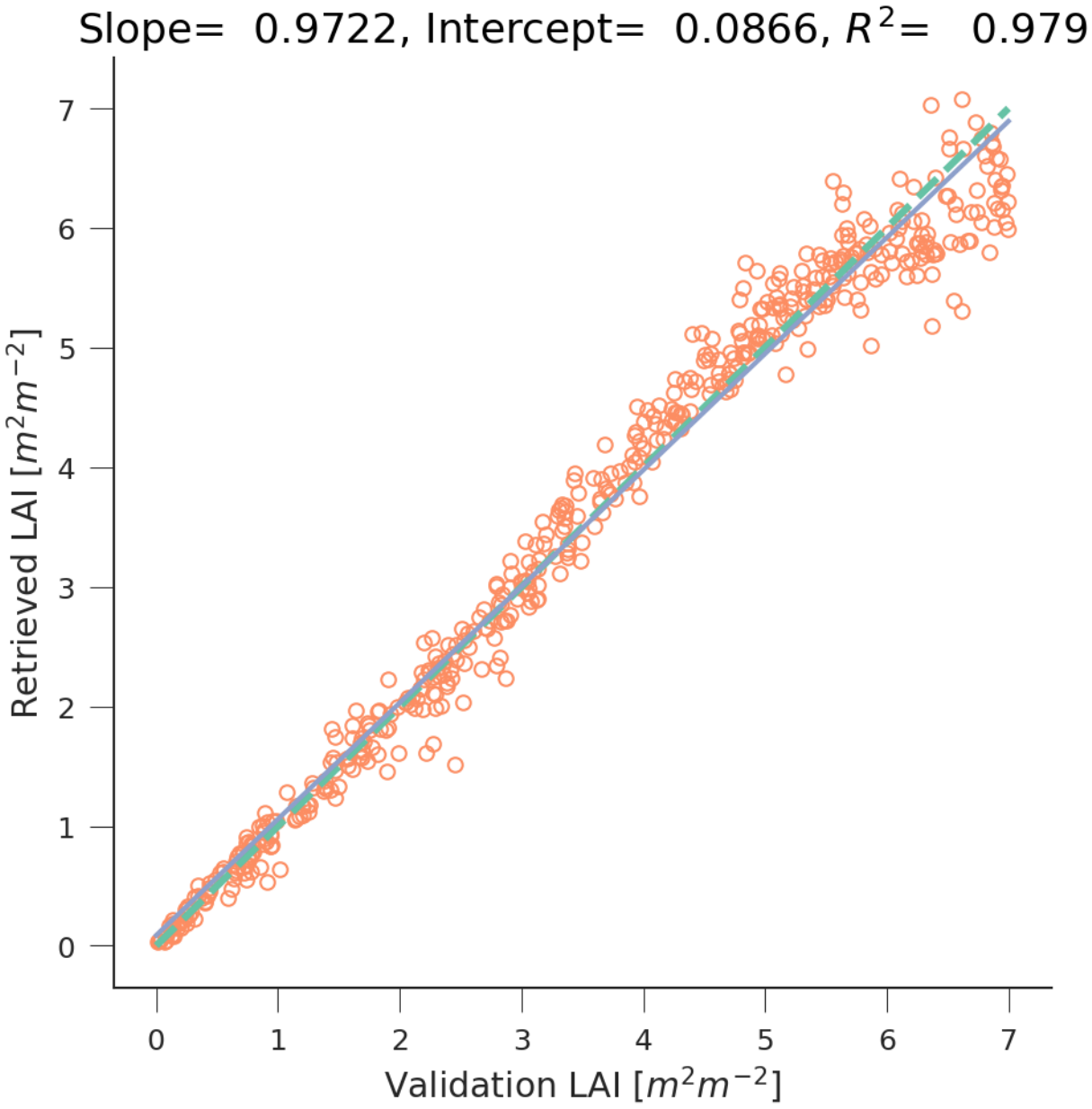

Fig. 1: Using Gaussian Processes to regress leaf area index (LAI) from Sentinel-2 data using the PROSAIL RT model. Comparison between the true LAI and retrieved LAI using the GPs.

The results are quite satisfactory. Another issue is whether these results will work as well on real Sentinel-2 data of random vegetation classes!!! One reason why they won't is because above I have assumed the soil to be black. While this won't matter for situations with large canopy cover, it will for low LAI.

CHAPTER 5

User Reference

5.1 The *GaussianProcess* class

class gp_emulator.**GaussianProcess**(*inputs=None*, *targets=None*, *emulator_file=None*)
Bases: object

A simple class for Gaussian Process emulation. Currently, it assumes a squared exponential covariance function, but other covariance functions ought to be possible and easy to implement.

hessian(*testing*)
Calculates the hessian of the GP for the testing sample. hessian returns a (nn by d by d) array.

learn_hyperparameters(*n_tries=15*, *verbose=False*, *x0=None*)
User method to fit the hyperparameters of the model, using random initialisations of parameters. The user should provide a number of tries (e.g. how many random starting points to avoid local minima), and whether it wants lots of information to be reported back.

n_tries: int, optional Number of random starting points

verbose: flag, optional How much information to parrot (e.g. convergence of the minimisation algorithm)

x0: array, optional If you want to start the learning process with a particular vector, set it up here.

loglikelihood(*theta*)
Calculates the loglikelihood for a set of hyperparameters `theta`. The size of `theta` is given by the dimensions of the input vector to the model to be emulated.

theta: array Hyperparameters

partial_devs(*theta*)
This function calculates the partial derivatives of the cost function as a function of the hyperameters, and is only needed during GP training.

theta: array Hyperparameter set

predict (*testing*, *do_deriv=True*, *do_unc=True*)
Make a prediction for a set of input vectors, as well as calculate the partial derivatives of the emulated model, and optionally, the "emulation uncertainty".

testing: array, size Npred * Ninputs The size of this array (and it must always be a 2D array!) is given by the number of input vectors that will be run through the emulator times the input vector size.

do_unc: flag, optional Calculate the uncertainty (if you don't set this flag, it can shave a few us.

do_deriv: flag, optional Whether to calculate the partial derivatives of the emulated model.

Three parameters (the mean, the variance and the partial derivatives) If some of those outputs have been left out, they are returned as *None* elements.

save_emulator (*emulator_file*)
Save emulator to disk as npz FileExistsError Saves an emulator to disk using an npz file.

5.2 The *MultivariateEmulator* class

class gp_emulator.**MultivariateEmulator** (*dump=None*, *X=None*, *y=None*, *hyperparams=None*, *model="*, *sza=0*, *vza=0*, *raa=0*, *thresh=0.98*, *n_tries=5*)

Bases: object

calculate_decomposition (*X*, *thresh*)
Does PCA decomposition

This simply does a PCA decomposition using the SVD. Note that if *X* is very large, more efficient methods of doing this might be required. The number of PCs to retain is selected as those required to estimate *thresh* of the total variance.

X: array (N_train, N_full) The modelled output array for training

thresh: float The threshold at where to cutoff the percentage of variance explained.

compress (*X*)
Project full-rank vector into PC basis

dump_emulator (*fname*, *model_name*, *sza*, *vza*, *raa*)
Save emulator to file for reuse

Saves the emulator to a file (*.npz* format) for reuse.

fname: str The output filename

hessian (*x*)
A method to approximate the Hessian. This method builds on the fact that the spectral emulators are a linear combination of individual emulators. Therefore, we can calculate the Hessian of the spectral emulator as the sum of the individual products of individual Hessians times the spectral basis functions.

predict (*y*, *do_unc=True*, *do_deriv=True*)
Prediction of input vector

The individual GPs predict the PC weights, and these are used to reconstruct the value of the function at a point *y*. Additionally, the derivative of the function is also calculated. This is returned as a *(N_params, N_full)* vector (i.e., it needs to be reduced along axis 1)

Parameters: y: array

The value of the prediction point

do_deriv: bool Whether derivatives are required or not

do_unc: bool Whether to calculate the uncertainty or not

Returns: A tuple with the predicted mean, predicted variance and patial derivatives. If any of the latter two elements have been switched off by *do_deriv* or *do_unc*, they'll be returned as *None*.

train_emulators(*X*, *y*, *hyperparams*, *n_tries=2*)
Train the emulators

This sets up the required emulators. If necessary (*hypeparams* is set to None), it will train the emulators.

X: array (N_train, N_full) The modelled output array for training

y: array (N_train, N_param) The corresponding training parameters for *X*

hyperparams: array (N_params + 2, N_PCs) The hyperparameters for the relevant GPs

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index