esem

Release v0.2.0

D Watson-Parris

Sep 20, 2022

CONTENTS:

CHAPTER

ONE

INSTALLING ESEM

1.1 Using PyPi

It is straightforward to install esem using pip, this will automatically include tensorflow (with GPU support):

```
$ pip install esem
```

Optionally also install GPFlow, keras or scikit-learn

```
$ pip install esem[gpflow]
```

Or

```
$ pip install esem[gpflow,keras,scikit-learn]
```

1.2 Using conda

In order to make the most of the support for Iris and CIS creating a specific conda environment is recommended. If you don't already have conda, you must first download and install it. Anaconda is a free conda package that includes Python and many common scientific and data analysis libraries, and is available here. Further documentation on using Anaconda and the features it provides can be found at http://docs.continuum.io/anaconda/index.html.

Having installed (mini-) conda - and ideally within a fresh environment - you can easily install CIS (and Iris) with the following command:

```
$ conda install -c conda-forge cis
```

It is then straightforward to install esem in to this environment using pip as above.

1.3 Dependencies

If you choose to install the dependencies yourself, use the following command to check the required dependencies are present:

```
$ python setup.py checkdep
```

CHAPTER

TWO

WHAT'S NEW IN ESEM

2.1 What's new in ESEm 1.1

This page documents the new features added, and bugs fixed in ESEm since version 1.0. For more detail see all changes here: https://github.com/duncanwp/ESEm/compare/1.0.0...1.1.0

2.1.1 ESEm 1.1 features

- We have added this What's New page for tracking the latest developments in ESEm!
- We have dropped the mandatory requirement of Iris to make installation of ESEm easier. We have also added support for xarray DataArrays so that users can use their preferred library for data processing.
- The `esem.emulator.Emulator.predict()` and `esem.emulator.Emulator.batch_stats()` methods can now accept pd.DataFrames to match the training interface. The associated doc-strings and signatures have been extended to reflect this.

2.1.2 Bugs fixed

- Use tqdm.auto to automatically choose the appropriate progress bar for the context
- Fix *plot_validation* handling of masked data

CHAPTER

THREE

EMULATING WITH ESEM

ESEm provides a simple and streamlined interface to emulate earth system datasets, denoted Y in the following documentation. These datasets can be provided as iris Cubes, xarray DataArrays or numpy arrays and the resulting emulation results will preserve any associated metadata. The corresponding predictors (X) can be provided as a numpy array or pandas *DataFrame*. This emulation is essentially just a regression estimating the functional form:

$$Y \approx f(X)$$

and can be performed using a variety of techniques using the same API.

3.1 Gaussian processes emulation

Gaussian processes (GPs) are a popular choice for model emulation due to their simple formulation and robust uncertainty estimates, particularly in cases where there is limited training data. Many excellent texts are available to describe their implementation and use (Rasmussen and Williams, 2005) and we only provide a short description here. Briefly, a GP is a stochastic process (a distribution of continuous functions) and can be thought of as an infinite dimensional normal distribution (hence the name).

The ESEm GP emulation module provides a thin wrapper around the GPFlow implementation. Please see their documentation for a detailed description of the implementation.

An important consideration when using GP regression is the form of the covariance matrix, or kernel. Typical kernels include: constant; linear; radial basis function (RBF; or squared exponential); and Matérn 3/2 and 5/2 which are only once and twice differentiable respectively. Kernels can also be designed to represent any aspect of the functions of interest such as non-stationarity or periodicity. This choice can often be informed by the physical setting and provides greater control and interpretability of the resulting model compared to e.g., Neural Networks.

Note: By default, ESEm uses a combination of linear, RBF and polynomial kernels which are suitable for the smooth and continuous parameter response expected for the examples used in this paper and related problems. However, given the importance of the kernel for determining the form of the functions generated by the GP we have also included the ability for users to specify combinations of other common kernels. See e.g., Duvenaud, 2011 for a clear description of some common kernels and their combinations, as well as work towards automated methods for choosing them.

The framework provided by GPFlow also allows for multi-output GP regression and ESEm takes advantage of this to automatically provide regression over each of the output features provided in the training data. E.g. Y can be of arbitrary dimensionality. It will be automatically flattened and reshaped before being passed to GPFlow.

The most convenient way to setup a GPFlow emulator is using the `esem.gp_model()` function which can be imported directly:

```
from esem import gp_model
```

This creates a regression model with a default kernel as described above but provides a convenient interface for defining arbitrary kernels through addition and multiplication. For example, to initialize a model with a `Linear+Cosine()` kernel:

```
from esem import gp_model

# X_train and Y_train are our predictors and outputs, respectively.
model = gp_model(X_train, Y_train, kernel=['Linear', 'Cosine'], kernel_op='add')
```

Further details are described in the function description *esem.gp_model()*.

Examples of emulation using Gaussian processes can be found in Emulating_using_GPs.ipynb and CMIP6_emulator.ipynb.

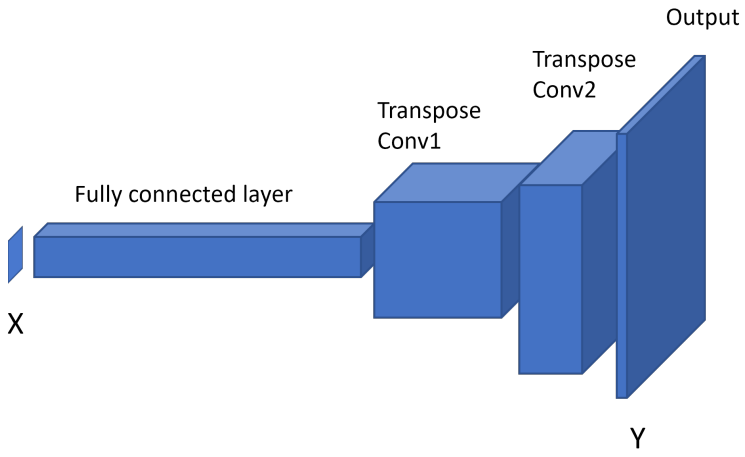
3.2 Neural network emulation

While fully connected neural networks have been used for many years, even in climate science, the recent surge in popularity has been powered by the increases in expressibility provided by deep, convolutional neural networks (CNNs) and the regularisation techniques which prevent these huge models from over-fitting the large amounts of training data required to train them.

Many excellent introductions can be found elsewhere but, briefly, a neural network consists of a network of nodes connecting (through a variety of architectures) the inputs to the target outputs via a series of weighted activation functions. The network architecture and activation functions are typically chosen a-priori and then the model weights are determined through a combination of back-propagation and (batch) gradient descent until the outputs match (defined by a given loss function) the provided training data. As previously discussed, the random dropping of nodes (by setting the weights to zero), termed dropout, can provide estimates of the prediction uncertainty of such networks.

The computational efficiency of such networks and the rich variety of architectures available have made them the tool of choice in many machine learning settings, and they are starting to be used in climate sciences for emulation, although the large amounts of training data required (especially compared to GPs) have so far limited their use somewhat.

ESEm provides a baseline CNN architecture based on the Keras library which essentially acts as a decoder - transforming input parameters into predicted 2(or 3) dimensional output fields:



This model can be easily constructed using the *esem.cnn_model()* function. It is possible to use any Keras model in this way though and there are many potential ways of improving / developing this simple model.

An example of emulation using this convolution neural network can be found in Emulating_using_ConvNets.ipynb.

3.3 Random forest emulation

ESEm also provides the option for emulation with Random Forests using the open-source implementation provided by scikit-learn. Random Forest estimators are comprised of an ensemble of decision trees; each decision tree is a recursive binary partition over the training data and the predictions are an average over the predictions of the decision trees.

As a result of this architecture, Random Forests (along with other algorithms built on decision trees) have two main attractions. Firstly, they require very little pre-processing of the inputs as the binary partitions are invariant to monotonic rescaling of the training data. Secondly, and of particular importance for climate problems, they are unable to extrapolate outside of their training data because the predictions are averages over subsets of the training dataset.

This model can be constructed using the *esem.rf_model()* function. All of the relevant scikit-learn arguments and keyword-arguments can be provided through this interface.

An example of emulation using the random forest can be found in CRM_Emulation_with_RandomForest.ipynb.

3.4 Data processing

Many of the above approaches make assumptions about, or simply perform better when, the training data is structured or distributed in a certain way. These transformations are purely to help the emulator fit the training data, and can complicate comparison with e.g. observations during calibration. ESEm provides a simple and transparent way of transforming the datasets for training, and this automatically un-transforms the model predictions to aid in observational comparison.

Where these transformations are strictly necessary for a given model then it will be included in the wrapper function. Other choices are left to the user to apply as required.

For example, to 'whiten' the data (that is, remove the mean and normalise by the standard deviation):

```
import esem
from esem import gp_model

# X_train and Y_train are our predictors and outputs, respectively.
model = gp_model(X_train, Y_train, data_processors=[esem.data_processors.Whiten()])
```

A full list of the data processors can be found in the API documentation.

3.5 Feature selection

ESEm includes a simple utility function that wraps the scikit-learn LassoLarsIC regression tool in order to enable an initial feature (parameter) selection. This can be useful to reduce the dimensionality of the input space. Either the Akaike information criterion (AIC) or the Bayes Information criterion (BIC) can be used, although BIC is the default.

For example,

```
from esem import gp_model
from esem.utils import get_param_mask

# X and Y are our model parameters and outputs respectively.
```

(continues on next page)

(continued from previous page)

```
active_params = get_param_mask(X, y)

# The model parameters can then be subsampled either directly
X_sub = X[:, active_params]

# Or by specifying the GP active_dims
active_dims, = np.where(active_params)
model = gp_model(X, y, active_dims=active_dims)
```

Note, this estimate only applies to one-dimensional outputs. Feature selection for higher dimension outputs is a much harder task beyond the scope of this package.

CHAPTER

FOUR

CALIBRATING WITH ESEM

Having trained a fast, robust emulator this can be used to calibrate our model against available observations. The following description closely follows that in our model description paper but with explicit links to the ESEm algorithms to aid clarity. Generally, this problem involves estimating the model parameters which could give rise to, or best match, the available observations.

In other words, we would like to know the posterior probability distribution of the input parameters: $p\left(\theta\middle|Y^0\right)$.

Using Bayes' theorem, we can write this as:

$$p\left(\theta\middle|Y^0\right) = p\left(Y^0\middle|\theta\right) p()p(Y0) \tag{4.1}$$

Where the probability of an output given the input parameters, $p\left(Y^0\middle|\theta\right)$, is referred to as the likelihood. While the model is capable of sampling this distribution, generally the full distribution is unknown and intractable, and we must approximate this likelihood. Depending on the purpose of the calibration and assumptions about the form of $p\left(Y^0\middle|Y\right)$, different techniques can be used. In order to determine a (conservative) estimate of the parametric uncertainty in the model for example, we can use approximate Bayesian computation (ABC) to determine those parameters which are plausible given a set of observations. Alternatively, we may wish to know the optimal parameters to best match a set of observations and Markov-Chain Monte-Carlo based techniques might be more appropriate. Both of these sampling strategies are available in ESEm and we describe each of them here.

4.1 Using approximate Bayesian computation (ABC)

The simplest ABC approach seeks to approximate the likelihood using only samples from the simulator and a discrepancy function ρ:

$$p\left(\theta\middle|Y^0\right) \propto p\left(Y^0\middle|Y\right) p\left(Y|\theta\right) p(\theta) \approx \int \mathbb{I}(\rho\left(Y^0, Y\right) \leq \epsilon)\ \ p\left(Y|\theta\right)\ \ p(\theta)\ dY \tag{4.2}$$

where the indicator function $\mathbb{I}(x) = 1$ if x is true and $\mathbb{I}(x) = 0$ if x is false, and ϵ is a small discrepancy. This can then be integrated numerically using e.g., Monte-Carlo sampling of $p(\theta)$. Any of those parameters for which $\rho\left(Y^0, Y\right) \leq \epsilon$ are accepted and those which do not are rejected. As $\epsilon \to \infty$ therefore, all parameters are accepted and we recover $p(\theta)$. For $\epsilon = 0$, it can be shown that we generate samples from the posterior $p\left(\theta\middle|Y^0\right)$ exactly.

In practice however the simulator proposals will never exactly match the observations and we must make a pragmatic choice for both ρ and ϵ. ESEm includes an implementation of the 'implausibility metric' which defines the discrepancy in terms of the standardized Cartesian distance:

$$\rho(Y^0, Y(\theta)) = \frac{\left|Y^0 - Y(\theta)\right|}{\sqrt{\sigma_E^2 + \sigma_Y^2 + \sigma_R^2 + \sigma_S^2}} = \rho(Y^0, \theta) \tag{4.3}$$

where the total standard deviation is taken to be the squared sum of the emulator variance (σ_E^2) and the uncertainty in the observations (σ_Y^2) and due to representation (σ_R^2) and structural model uncertainties (σ_S^2) as described in the paper.

Framed in this way, ϵ, can be thought of as representing the number of standard deviations the (emulated) model value is from the observations. While this can be treated as a free parameter and may be specified in ESEm, it is common to choose $\epsilon = 3$ since it can be shown that for unimodal distributions values of 3σ correspond to a greater than 95% confidence bound. This approach is implemented in the `esem.abc_sampler.ABCSampler` class where ϵ is referred to as a threshold since it defines the cut-off for acceptance.

In the general case, multiple ($\mathcal{N}$) observations can be used and ρ can be written as a vector of implausibilities, $\rho(Y_i^O, \theta)$ or simply $\rho_i(\theta)$, and a modified method of rejection or acceptance must be used. A simple choice is to require $\rho_i < \epsilon \; \forall \; i \; \in \mathcal{N}$, however this can become restrictive for large $\mathcal{N}$ due to the curse of dimensionality. The first step should be to reduce $\mathcal{N}$ through the use of summary statistics, such as averaging over regions, or stations, or by performing an e.g. Principle Component Analysis (PCA) decomposition.

An alternative is to introduce a tolerance (T) such that only some proportion of ρ_i need to be smaller than ϵ : $\sum_{i=0}^{\mathcal{N}} H(\rho_i \; - \; \epsilon) < T$, where H is the Heaviside function. This tolerance can be specified when sampling using the `esem.abc_sampler.ABCSampler`. An efficient implementation of this approach whereby the acceptance is calculated in batches on the GPU can be particularly useful when dealing with high-dimensional outputs `esem.abc_sampler.ABCSampler`. It is recommended however, to choose $T = 0$ as a first approximation and then identify any particular observations which generate a very large implausibilities, since this provides a mechanism for identifying potential structural (or observational) errors.

A useful way of identifying such observations is using the `esem.abc_sampler.ABCSampler.get_implausibility()` method which returns the full implausibility matrix ρ_i. Note this may be very large (N-samples x N-observations) so it is recommended that only a subset of the full sample space be requested. The offending observations can then be removed and noted for further investigation.

Examples of the ABC sampling can be found in Calibrating_GPs_using_ABC.ipynb.

4.2 Using Markov-chain Monte-Carlo

The ABC method described above is simple and powerful, but somewhat inefficient as it repeatedly samples from the same prior. In reality each rejection or acceptance of a set of parameters provides us with extra information about the 'true' form of $p\left(\theta \middle| Y^0\right)$ so that the sampler could spend more time in plausible regions of the parameter space. This can then allow us to use smaller values of ϵ and hence find better approximations of $p\left(\theta \middle| Y^0\right)$.

Given the joint probability distribution described by Eq. 2 and an initial choice of parameters θ' and (emulated) output Y', the acceptance probability r of a new set of parameters (θ) is given by:

$$r = \frac{p\left(Y^0 \middle| Y'\right) p\left(\theta' | \theta\right) p(\theta')}{p\left(Y^0 | Y\right) p\left(\theta | \theta'\right) p(\theta)} \tag{4.4}$$

The `esem.sampler.MCMCSampler` class uses the TensorFlow-probability implementation of Hamiltonian Monte-Carlo (HMC) which uses the gradient information automatically calculated by TensorFlow to inform the proposed new parameters θ. For simplicity, we assume that the proposal distribution is symmetric: $p\left(\theta' | \theta\right) \; = \; p\left(\theta | \theta'\right)$, which is implemented as a zero log-acceptance correction in the initialisation of the TensorFlow target distribution. The target log probability provided to the TensorFlow HMC algorithm is then:

$$log(r) = log(p\left(Y^0 \middle| Y'\right)) \; + \; log(p(\theta')) \; - \; log(p\left(Y^0 \middle| Y\right)) \; - \; log(p(\theta)) \tag{4.5}$$

Note, that for this implementation the distance metric ρ must be cast as a probability distribution with values [0, 1]. We therefore assume that this discrepancy can be approximated as a normal distribution centred about zero, with standard deviation equal to the sum of the squares of the variances as described in Eq. 3:

$$p\left(Y^0 \middle| Y\right) \approx \frac{1}{\sigma_t \sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{Y^0 - Y}{\sigma_t}\right)^2}, \;\; \sigma_t = \sqrt{\sigma_E^2 + \sigma_Y^2 + \sigma_R^2 + \sigma_S^2} \tag{4.6}$$

The *esem.sampler.MCMCSampler.sample()* method will then return the requested number of accepted samples as well as reporting the acceptance rate, which provides a useful metric for tuning the algorithm. It should be noted that MCMC algorithms can be sensitive to a number of key parameters, including the number of burn-in steps used (and discarded) before sampling occurs and the step size. Each of these can be controlled via keyword arguments to the *esem.sampler.MCMCSampler.sample()* method.

This approach can provide much more efficient sampling of the emulator and provide improved parameter estimates, especially when used with informative priors which can guide the sampler.

Examples of the MCMC sampling can be found in Calibrating_GPs_using_MCMC.ipynb and CMIP6_emulator.ipynb.

CHAPTER

FIVE

EXAMPLES

5.1 Emulating using GPs

[1]:
```
import os
## Ignore my broken HDF5 install...
os.putenv("HDF5_DISABLE_VERSION_CHECK", '1')
```

[2]:
```
import iris
from utils import get_bc_ppe_data

from esem import gp_model
from esem.utils import get_random_params

import iris.quickplot as qplt
import matplotlib.pyplot as plt
%matplotlib inline
```

```
C:\Users\duncan\miniconda3\envs\gcem_dev\lib\site-packages\h5py\__init__.py:40:␣
↪UserWarning: h5py is running against HDF5 1.10.6 when it was built against 1.10.5,␣
↪this may cause problems
  '{0}.{1}.{2}'.format(*version.hdf5_built_version_tuple)
```

5.1.1 Read in the parameters and observables

[4]:
```
ppe_params, ppe_aaod = get_bc_ppe_data()
```

```
C:\Users\duncan\miniconda3\envs\gcem_dev\lib\site-packages\iris\__init__.py:249:␣
↪IrisDeprecation: setting the 'Future' property 'netcdf_promote' is deprecated and will␣
↪be removed in a future release. Please remove code that sets this property.
  warn_deprecated(msg.format(name))
C:\Users\duncan\miniconda3\envs\gcem_dev\lib\site-packages\iris\__init__.py:249:␣
↪IrisDeprecation: setting the 'Future' property 'netcdf_promote' is deprecated and will␣
↪be removed in a future release. Please remove code that sets this property.
  warn_deprecated(msg.format(name))
```

[5]:
```
n_test = 5

X_test, X_train = ppe_params[:n_test], ppe_params[n_test:]
Y_test, Y_train = ppe_aaod[:n_test,0], ppe_aaod[n_test:,0]
```

5.1.2 Setup and run the models

[9]:
```
model = gp_model(X_train, Y_train)
```

[15]:
```
model.train()
```

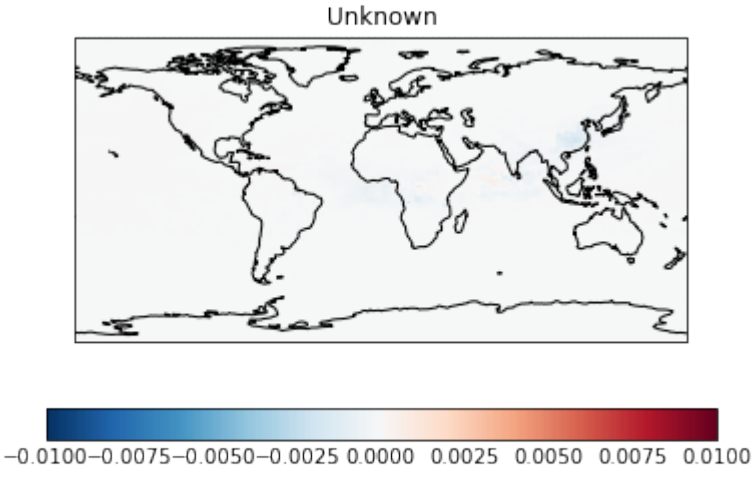
[16]:
```
m, v = model.predict(X_test.values)
```

[17]:
```
## validation_plot(Y_test.data.flatten(), m.data.flatten(), v.data.flatten())
```

[18]:
```
qplt.pcolormesh((m.collapsed('sample', iris.analysis.MEAN)-Y_test.collapsed('job', iris.
↪analysis.MEAN)), cmap='RdBu_r', vmin=-0.01, vmax=0.01)
plt.gca().coastlines()
```

```
C:\Users\duncan\miniconda3\envs\gcem_dev\lib\site-packages\iris\coords.py:1410:␣
↪UserWarning: Collapsing a non-contiguous coordinate. Metadata may not be fully␣
↪descriptive for 'sample'.
  warnings.warn(msg.format(self.name()))
C:\Users\duncan\miniconda3\envs\gcem_dev\lib\site-packages\iris\coords.py:1410:␣
↪UserWarning: Collapsing a non-contiguous coordinate. Metadata may not be fully␣
↪descriptive for 'job'.
  warnings.warn(msg.format(self.name()))
C:\Users\duncan\miniconda3\envs\gcem_dev\lib\site-packages\iris\coords.py:1193:␣
↪UserWarning: Coordinate 'longitude' is not bounded, guessing contiguous bounds.
  'contiguous bounds.'.format(self.name()))
C:\Users\duncan\miniconda3\envs\gcem_dev\lib\site-packages\iris\coords.py:1193:␣
↪UserWarning: Coordinate 'latitude' is not bounded, guessing contiguous bounds.
  'contiguous bounds.'.format(self.name()))
```

[18]:
```
<cartopy.mpl.feature_artist.FeatureArtist at 0x19ea2c99388>
```



[19]:
```
## Note the model variance is constant across the outputs
v.data.max()
```

[19]:
```
1.175408691451335e-06
```

[15]:
```
get_random_params(3, int(1e5)).shape
```

[15]:
```
(100000, 3)
```

[16]:
```
m, sd = model.batch_stats(get_random_params(3, int(1e3)))
```

```
100%|##########| 1000/1000 [00:09<00:00, 149.00sample/s]
```

[17]:
```
m, sd = model.batch_stats(get_random_params(3, int(1e4)), batch_size=10)
```

```
100%|##########| 10000/10000 [00:07<00:00, 1459.07sample/s]
```

[19]:
```
m, sd = model.batch_stats(get_random_params(3, int(1e6)), batch_size=10000)
```

```
100%|##########| 1000000/1000000 [00:09<00:00, 122569.61sample/s]
```

[ ]:
```

```

5.2 Emulating using CNNs

[1]:
```
import iris

from utils import get_bc_ppe_data

from esem import cnn_model
from esem.utils import get_random_params

import iris.quickplot as qplt
import matplotlib.pyplot as plt
%matplotlib inline
```

5.2.1 Read in the parameters and data

[2]:
```
ppe_params, ppe_aaod = get_bc_ppe_data()
```

[3]:
```
## Ensure the time dimension is last - this is treated as the color 'channel'
ppe_aaod.transpose((0,2,3,1))
```

[4]:
```
n_test = 5

X_test, X_train = ppe_params[:n_test], ppe_params[n_test:]
Y_test, Y_train = ppe_aaod[:n_test], ppe_aaod[n_test:]
```

[5]:
```
Y_train
```

[5]:
```
<iris 'Cube' of Absorption optical thickness - total 550nm / (1) (job: 34; latitude: 96;␣
↪longitude: 192; time: 12)>
```

5.2.2 Setup and run the models

[6]:
```
model = cnn_model(X_train, Y_train)
```

[7]:
```
model.train()
```

```
Epoch 1/100
4/4 [==============================] - 3s 71ms/step - loss: 1.1407 - val_loss: 0.4622
Epoch 2/100
4/4 [==============================] - 0s 19ms/step - loss: 1.1396 - val_loss: 0.4620
Epoch 3/100
4/4 [==============================] - 0s 17ms/step - loss: 1.1391 - val_loss: 0.4618
Epoch 4/100
4/4 [==============================] - 0s 19ms/step - loss: 1.1382 - val_loss: 0.4613
Epoch 5/100
4/4 [==============================] - ETA: 0s - loss: 1.048 - 0s 18ms/step - loss: 1.
↪1360 - val_loss: 0.4577
Epoch 6/100
4/4 [==============================] - 0s 17ms/step - loss: 1.1256 - val_loss: 0.4536
Epoch 7/100
4/4 [==============================] - 0s 18ms/step - loss: 1.1114 - val_loss: 0.4446
Epoch 8/100
4/4 [==============================] - 0s 19ms/step - loss: 1.0957 - val_loss: 0.4372
Epoch 9/100
4/4 [==============================] - 0s 20ms/step - loss: 1.0746 - val_loss: 0.4246
Epoch 10/100
4/4 [==============================] - 0s 18ms/step - loss: 1.0569 - val_loss: 0.4152
Epoch 11/100
4/4 [==============================] - 0s 20ms/step - loss: 1.0326 - val_loss: 0.4073
Epoch 12/100
4/4 [==============================] - 0s 19ms/step - loss: 1.0055 - val_loss: 0.4009
Epoch 13/100
4/4 [==============================] - 0s 20ms/step - loss: 0.9815 - val_loss: 0.3892
Epoch 14/100
4/4 [==============================] - 0s 21ms/step - loss: 0.9567 - val_loss: 0.3797
Epoch 15/100
4/4 [==============================] - 0s 22ms/step - loss: 0.9285 - val_loss: 0.3679
Epoch 16/100
4/4 [==============================] - 0s 22ms/step - loss: 0.9015 - val_loss: 0.3472
Epoch 17/100
4/4 [==============================] - 0s 21ms/step - loss: 0.8915 - val_loss: 0.3343
Epoch 18/100
4/4 [==============================] - 0s 20ms/step - loss: 0.8643 - val_loss: 0.3255
Epoch 19/100
4/4 [==============================] - 0s 19ms/step - loss: 0.8362 - val_loss: 0.3176
Epoch 20/100
4/4 [==============================] - 0s 19ms/step - loss: 0.8201 - val_loss: 0.3009
Epoch 21/100
4/4 [==============================] - 0s 19ms/step - loss: 0.7953 - val_loss: 0.2862
Epoch 22/100
4/4 [==============================] - 0s 19ms/step - loss: 0.7777 - val_loss: 0.2778
Epoch 23/100
4/4 [==============================] - 0s 18ms/step - loss: 0.7578 - val_loss: 0.2747
```

(continues on next page)

(continued from previous page)

```
Epoch 24/100
4/4 [==============================] - 0s 19ms/step - loss: 0.7465 - val_loss: 0.2592
Epoch 25/100
4/4 [==============================] - 0s 22ms/step - loss: 0.7223 - val_loss: 0.2570
Epoch 26/100
4/4 [==============================] - 0s 19ms/step - loss: 0.7036 - val_loss: 0.2337
Epoch 27/100
4/4 [==============================] - 0s 20ms/step - loss: 0.6804 - val_loss: 0.2253
Epoch 28/100
4/4 [==============================] - 0s 18ms/step - loss: 0.6666 - val_loss: 0.2308
Epoch 29/100
4/4 [==============================] - 0s 18ms/step - loss: 0.6502 - val_loss: 0.2109
Epoch 30/100
4/4 [==============================] - 0s 19ms/step - loss: 0.6315 - val_loss: 0.1978
Epoch 31/100
4/4 [==============================] - 0s 18ms/step - loss: 0.6138 - val_loss: 0.1819
Epoch 32/100
4/4 [==============================] - 0s 19ms/step - loss: 0.5970 - val_loss: 0.1738
Epoch 33/100
4/4 [==============================] - 0s 21ms/step - loss: 0.5801 - val_loss: 0.1640
Epoch 34/100
4/4 [==============================] - 0s 22ms/step - loss: 0.5622 - val_loss: 0.1643
Epoch 35/100
4/4 [==============================] - 0s 18ms/step - loss: 0.5502 - val_loss: 0.1508
Epoch 36/100
4/4 [==============================] - 0s 18ms/step - loss: 0.5405 - val_loss: 0.1625
Epoch 37/100
4/4 [==============================] - 0s 18ms/step - loss: 0.5311 - val_loss: 0.1517
Epoch 38/100
4/4 [==============================] - 0s 18ms/step - loss: 0.5191 - val_loss: 0.1331
Epoch 39/100
4/4 [==============================] - 0s 19ms/step - loss: 0.5063 - val_loss: 0.1264
Epoch 40/100
4/4 [==============================] - 0s 18ms/step - loss: 0.4976 - val_loss: 0.1217
Epoch 41/100
4/4 [==============================] - 0s 18ms/step - loss: 0.4865 - val_loss: 0.1159
Epoch 42/100
4/4 [==============================] - 0s 18ms/step - loss: 0.4715 - val_loss: 0.1112
Epoch 43/100
4/4 [==============================] - 0s 19ms/step - loss: 0.4604 - val_loss: 0.1131
Epoch 44/100
4/4 [==============================] - 0s 19ms/step - loss: 0.4498 - val_loss: 0.1002
Epoch 45/100
4/4 [==============================] - 0s 21ms/step - loss: 0.4404 - val_loss: 0.0969
Epoch 46/100
4/4 [==============================] - 0s 20ms/step - loss: 0.4320 - val_loss: 0.0938
Epoch 47/100
4/4 [==============================] - 0s 18ms/step - loss: 0.4242 - val_loss: 0.0905
Epoch 48/100
4/4 [==============================] - 0s 18ms/step - loss: 0.4173 - val_loss: 0.0874
Epoch 49/100
4/4 [==============================] - 0s 19ms/step - loss: 0.4105 - val_loss: 0.0903
```

(continues on next page)

(continued from previous page)

```
Epoch 50/100
4/4 [==============================] - 0s 18ms/step - loss: 0.4032 - val_loss: 0.0825
Epoch 51/100
4/4 [==============================] - 0s 21ms/step - loss: 0.3931 - val_loss: 0.0872
Epoch 52/100
4/4 [==============================] - 0s 22ms/step - loss: 0.3891 - val_loss: 0.0839
Epoch 53/100
4/4 [==============================] - 0s 22ms/step - loss: 0.3813 - val_loss: 0.0768
Epoch 54/100
4/4 [==============================] - 0s 22ms/step - loss: 0.3767 - val_loss: 0.0764
Epoch 55/100
4/4 [==============================] - 0s 21ms/step - loss: 0.3720 - val_loss: 0.0776
Epoch 56/100
4/4 [==============================] - 0s 21ms/step - loss: 0.3671 - val_loss: 0.0723
Epoch 57/100
4/4 [==============================] - 0s 20ms/step - loss: 0.3601 - val_loss: 0.0719
Epoch 58/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3570 - val_loss: 0.0708
Epoch 59/100
4/4 [==============================] - 0s 20ms/step - loss: 0.3562 - val_loss: 0.0704
Epoch 60/100
4/4 [==============================] - 0s 20ms/step - loss: 0.3519 - val_loss: 0.0699
Epoch 61/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3488 - val_loss: 0.0689
Epoch 62/100
4/4 [==============================] - 0s 20ms/step - loss: 0.3444 - val_loss: 0.0727
Epoch 63/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3427 - val_loss: 0.0795
Epoch 64/100
4/4 [==============================] - 0s 20ms/step - loss: 0.3420 - val_loss: 0.0691
Epoch 65/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3360 - val_loss: 0.0678
Epoch 66/100
4/4 [==============================] - 0s 21ms/step - loss: 0.3336 - val_loss: 0.0664
Epoch 67/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3303 - val_loss: 0.0689
Epoch 68/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3297 - val_loss: 0.0669
Epoch 69/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3246 - val_loss: 0.0659
Epoch 70/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3246 - val_loss: 0.0682
Epoch 71/100
4/4 [==============================] - 0s 20ms/step - loss: 0.3217 - val_loss: 0.0660
Epoch 72/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3196 - val_loss: 0.0656
Epoch 73/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3201 - val_loss: 0.0667
Epoch 74/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3166 - val_loss: 0.0653
Epoch 75/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3151 - val_loss: 0.0666
```

(continues on next page)

(continued from previous page)

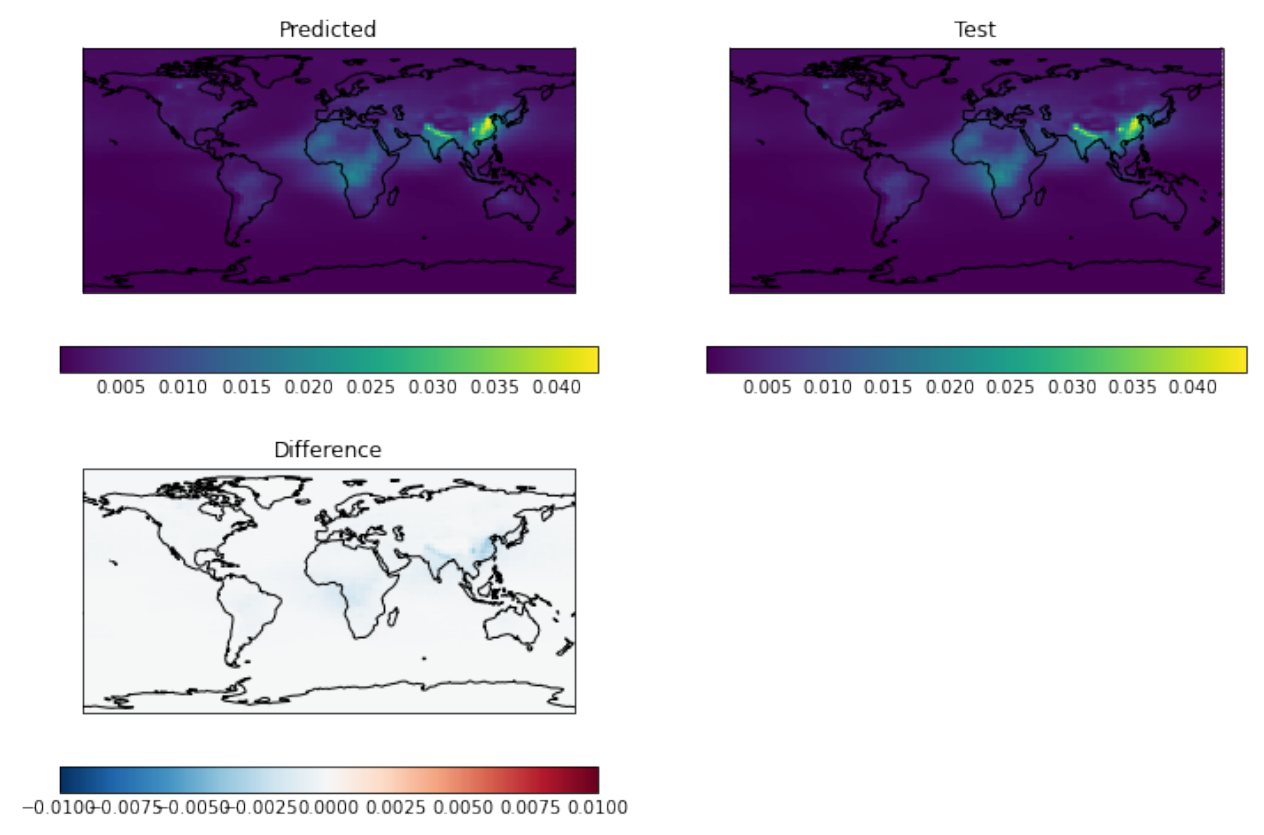
```
Epoch 76/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3137 - val_loss: 0.0657
Epoch 77/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3143 - val_loss: 0.0648
Epoch 78/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3112 - val_loss: 0.0647
Epoch 79/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3084 - val_loss: 0.0674
Epoch 80/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3087 - val_loss: 0.0650
Epoch 81/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3056 - val_loss: 0.0709
Epoch 82/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3082 - val_loss: 0.0651
Epoch 83/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3048 - val_loss: 0.0644
Epoch 84/100
4/4 [==============================] - 0s 19ms/step - loss: 0.3036 - val_loss: 0.0669
Epoch 85/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3032 - val_loss: 0.0662
Epoch 86/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3030 - val_loss: 0.0672
Epoch 87/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3016 - val_loss: 0.0691
Epoch 88/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3014 - val_loss: 0.0706
Epoch 89/100
4/4 [==============================] - 0s 18ms/step - loss: 0.3006 - val_loss: 0.0648
Epoch 90/100
4/4 [==============================] - 0s 18ms/step - loss: 0.2988 - val_loss: 0.0670
Epoch 91/100
4/4 [==============================] - 0s 17ms/step - loss: 0.2985 - val_loss: 0.0670
Epoch 92/100
4/4 [==============================] - 0s 21ms/step - loss: 0.2977 - val_loss: 0.0635
Epoch 93/100
4/4 [==============================] - 0s 20ms/step - loss: 0.2967 - val_loss: 0.0638
Epoch 94/100
4/4 [==============================] - 0s 23ms/step - loss: 0.2969 - val_loss: 0.0654
Epoch 95/100
4/4 [==============================] - 0s 19ms/step - loss: 0.2954 - val_loss: 0.0648
Epoch 96/100
4/4 [==============================] - 0s 21ms/step - loss: 0.2973 - val_loss: 0.0638
Epoch 97/100
4/4 [==============================] - 0s 23ms/step - loss: 0.2949 - val_loss: 0.0668
Epoch 98/100
4/4 [==============================] - 0s 24ms/step - loss: 0.2945 - val_loss: 0.0641
Epoch 99/100
4/4 [==============================] - 0s 22ms/step - loss: 0.2936 - val_loss: 0.0648
Epoch 100/100
4/4 [==============================] - 0s 23ms/step - loss: 0.2932 - val_loss: 0.0651
```

[8]:
```
m, v = model.predict(X_test.to_numpy())
```

[9]:
```
## TODO: Tidy this up a bit
plt.figure(figsize=(12, 8))
plt.subplot(2,2,1)
qplt.pcolormesh(m[0].collapsed('time', iris.analysis.MEAN))
plt.gca().set_title('Predicted')
plt.gca().coastlines()

plt.subplot(2,2,2)
qplt.pcolormesh(Y_test[0].collapsed('time', iris.analysis.MEAN))
plt.gca().set_title('Test')
plt.gca().coastlines()

plt.subplot(2,2,3)
qplt.pcolormesh((m.collapsed(['sample', 'time'], iris.analysis.MEAN)-Y_test.collapsed([
↪'job', 'time'], iris.analysis.MEAN)), cmap='RdBu_r', vmin=-0.01, vmax=0.01)
plt.gca().coastlines()
plt.gca().set_title('Difference')
```

```
C:\Users\duncan\miniconda3\envs\climatebench\lib\site-packages\iris\coords.py:1803:␣
↪UserWarning: Coordinate 'longitude' is not bounded, guessing contiguous bounds.
  warnings.warn(
C:\Users\duncan\miniconda3\envs\climatebench\lib\site-packages\iris\coords.py:1803:␣
↪UserWarning: Coordinate 'latitude' is not bounded, guessing contiguous bounds.
  warnings.warn(
C:\Users\duncan\miniconda3\envs\climatebench\lib\site-packages\iris\coords.py:1803:␣
↪UserWarning: Coordinate 'longitude' is not bounded, guessing contiguous bounds.
  warnings.warn(
C:\Users\duncan\miniconda3\envs\climatebench\lib\site-packages\iris\coords.py:1803:␣
↪UserWarning: Coordinate 'latitude' is not bounded, guessing contiguous bounds.
  warnings.warn(
C:\Users\duncan\miniconda3\envs\climatebench\lib\site-packages\iris\coords.py:1979:␣
↪UserWarning: Collapsing a non-contiguous coordinate. Metadata may not be fully␣
↪descriptive for 'sample'.
  warnings.warn(msg.format(self.name()))
C:\Users\duncan\miniconda3\envs\climatebench\lib\site-packages\iris\coords.py:1979:␣
↪UserWarning: Collapsing a non-contiguous coordinate. Metadata may not be fully␣
↪descriptive for 'job'.
  warnings.warn(msg.format(self.name()))
C:\Users\duncan\miniconda3\envs\climatebench\lib\site-packages\iris\coords.py:1803:␣
↪UserWarning: Coordinate 'longitude' is not bounded, guessing contiguous bounds.
  warnings.warn(
C:\Users\duncan\miniconda3\envs\climatebench\lib\site-packages\iris\coords.py:1803:␣
↪UserWarning: Coordinate 'latitude' is not bounded, guessing contiguous bounds.
  warnings.warn(
```

[9]:
```
Text(0.5, 1.0, 'Difference')
```



[10]:
```
m, sd = model.batch_stats(get_random_params(3, int(1e5)), batch_size=1000)
```

```
  0%|          | 0/100000 [00:00<?, ?sample/s]
```

```
<tqdm.auto.tqdm object at 0x0000019580079D60>
```

[ ]:

5.3 Random Forest Example: Cloud-resolving model sensitivity

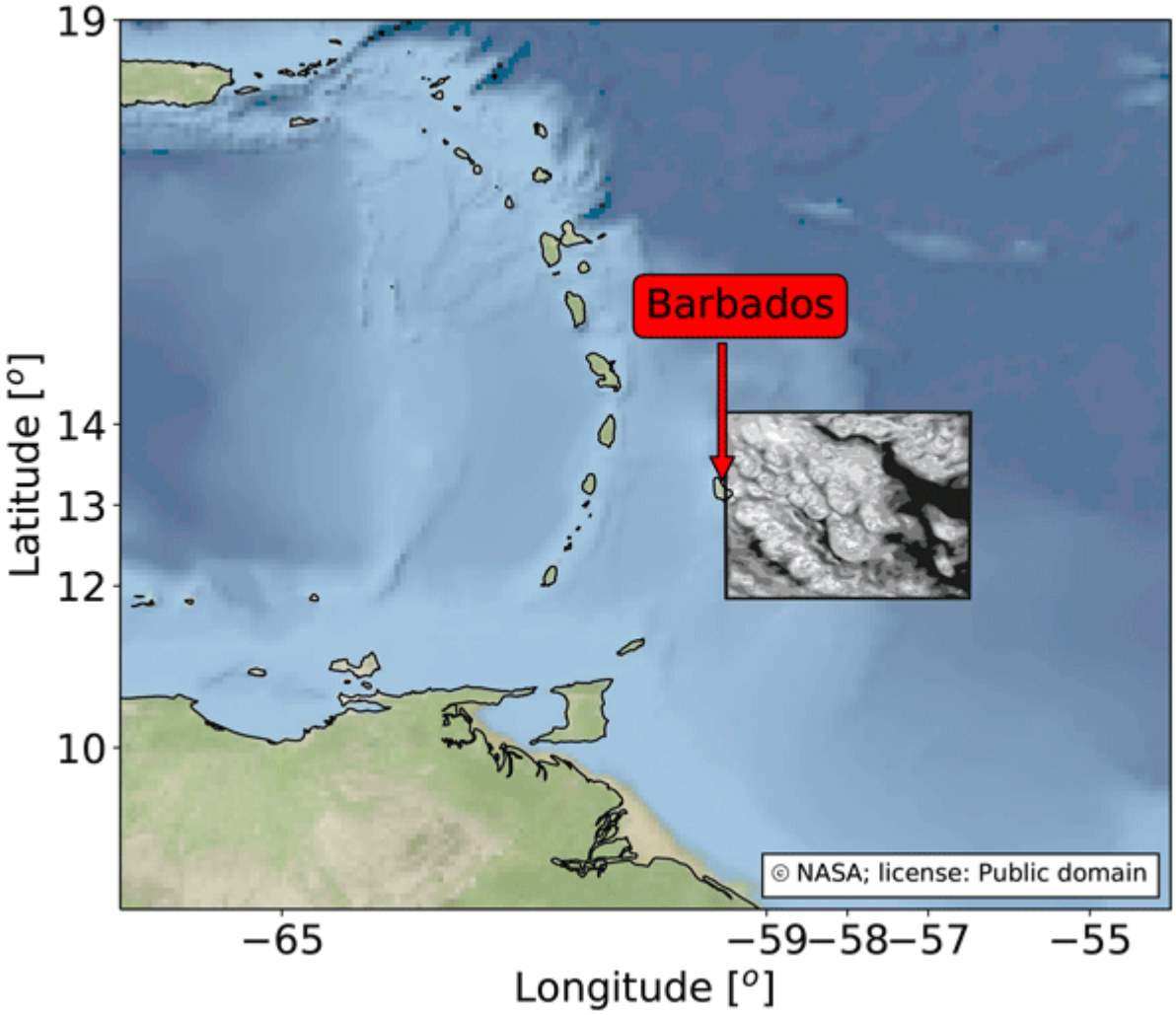
In this example, we will use an ensemble of large-domain simulations of realistic shallow cloud fields to explore the sensitivity of shallow precipitation to local changes in the environment.

The simulation data we use for training the emulator is taken from a recent study Dagan and Stier (2020), where they performed ensemble daily simulations for one month-long period during December 2013 over the ocean to the East of Barbados, such that they samepled variability associated with shallow convection. Each day of the month consisted of two runs, both forced by realistic boundary conditions taken from reanalysis, but with different cloud droplet number concentrations (CDNC) to represent clean and polluted conditions. The altered CDNC was found to have little impact on the precipitation rate in the simulations, and so we simply treat the CDNC change as a perturbation to the initial conditions, and combine the two CDNC runs from each day together to increase the amount of data available for training the emulator. At hourly resolution, this provides us with 1488 data points.

However, given that the amount of precipitation is strongly tied to the local cloud regime, not fully controlling for cloud regime can introduce spurious correlations when training the emulator. As such we also filter out all hours which are not associated with shallow convective clouds. To do this, we consider domain-mean vertical profiles of total cloud water content (liquid + ice), q_t, and filter out all hours where the vertical sum of q_t below 600hPa exceeds 10^{-6} kg/kg. This

condition allows us to filter out hours associated with the onset and development of deep convection in the domain, as well as masking out hours with high cirrus layers or hours dominated by transient mesoscale convective activity which is advected in by the boundary conditions. After this, we are left with 850 hourly data point which meet our criteria and can be used to train the emulator.

In this example we emulate hourly `precip` using domain-mean features: `liquid water path` (LWP), `geopotential height at 700hPa` (z:math:*{700}*), `Estimated Inversion Strength` (EIS), `sea-surface temperature` (SST) and the `vertical pressure velocity at 700hPa` (w700).



References:

Dagan, G. and Stier, P.: Ensemble daily simulations for elucidating cloud–aerosol interactions under a large spread of realistic environmental conditions, Atmos. Chem. Phys., 20, 6291–6303, https://doi.org/10.5194/acp-20-6291-2020, 2020.

[1]:
```
import numpy as np
import pandas as pd
import iris

from utils import get_crm_data
from esem.utils import plot_results, prettify_plot, add_121_line, leave_one_out
```

(continues on next page)

(continued from previous page)

```
from esem import rf_model

from matplotlib import pyplot as plt
plt.style.use('default')
%matplotlib inline
```

Concatenate 20cdnc and 200cdnc runs into one dataframe

[2]:
```
df_crm = get_crm_data()
df_crm
```

[2]:
```
       precip    pres_msl       LWP      WS10     lhfl_s     shfl_s  \
0    0.004593  101407.410  0.035898  6.639860 -167.53857   5.745860
1    0.006900  101356.266  0.044468  6.822748 -176.93939   4.438721
2    0.008916  101316.420  0.051559  6.798490 -182.61697   3.649221
3    0.008932  101270.490  0.057509  6.756970 -188.87599   3.033055
4    0.016204  101256.270  0.064226  6.763690 -194.85498   2.826119
..        ...         ...       ...       ...        ...        ...
845  0.063121  101309.750  0.064794  8.253145 -191.23718  12.219704
846  0.064601  101303.110  0.063914  8.326073 -192.57118  11.947702
847  0.046773  101332.234  0.059974  8.404624 -193.80084  12.372276
848  0.056623  101394.280  0.062895  8.385845 -192.18195  13.336615
849  0.064975  101438.690  0.069100  8.429897 -192.28928  13.679647

           LTS      w500      w600      w700   wmax850   wstd850      zg500  \
0    13.180252 -0.014463 -0.012311 -0.010275 -0.000024  0.000947  56627.516
1    13.279678 -0.015064 -0.012710 -0.008676  0.000030  0.000382  56572.645
2    13.333527 -0.014811 -0.012014 -0.006025  0.000642  0.000511  56525.613
3    13.328018 -0.013470 -0.012141 -0.004758  0.001519  0.000476  56471.332
4    13.317032 -0.010917 -0.011119 -0.003158  0.003252  0.000958  56443.758
..         ...       ...       ...       ...       ...       ...        ...
845  10.142059 -0.024480 -0.006400 -0.007968 -0.000044  0.001105  56084.273
846  10.162674 -0.019426  0.000300 -0.003904 -0.000034  0.000588  56071.547
847  10.166580 -0.014384  0.004355 -0.000284 -0.000251  0.000650  56079.492
848  10.149658 -0.016936  0.002702  0.000667  0.000013  0.000509  56117.140
849  10.164475 -0.021005  0.000413  0.000406 -0.000102  0.000838  56146.297

         zg600      zg700      rh850      rh700   u_shear       EIS  \
0    42694.220  30541.566  67.243774  60.067740 -4.662799  0.989443
1    42640.473  30488.172  69.299180  58.453730 -4.322696  1.130803
2    42593.594  30442.703  71.522900  56.912193 -3.925541  1.242463
3    42539.062  30390.662  74.115690  55.652990 -3.556973  1.304206
4    42510.805  30364.852  77.510765  54.434470 -3.319007  1.362710
..         ...        ...        ...        ...       ...       ...
845  42214.140  30222.945  83.696740  77.278465 -5.993636 -2.696190
846  42206.734  30214.715  84.196236  77.536760 -5.848422 -2.673406
847  42225.984  30236.312  84.394960  77.754560 -5.663757 -2.643809
848  42272.740  30286.500  84.437530  78.009740 -5.427930 -2.635981
849  42305.730  30322.684  84.389620  78.030650 -5.215088 -2.612224

            SST
0    301.173248
1    301.173248
```

(continues on next page)

(continued from previous page)

```
2    301.173248
3    301.173248
4    301.173248
..          ...
845  300.126465
846  300.126465
847  300.126465
848  300.126465
849  300.126465

[850 rows x 20 columns]
```

Extract the precipitation timeseries as target data

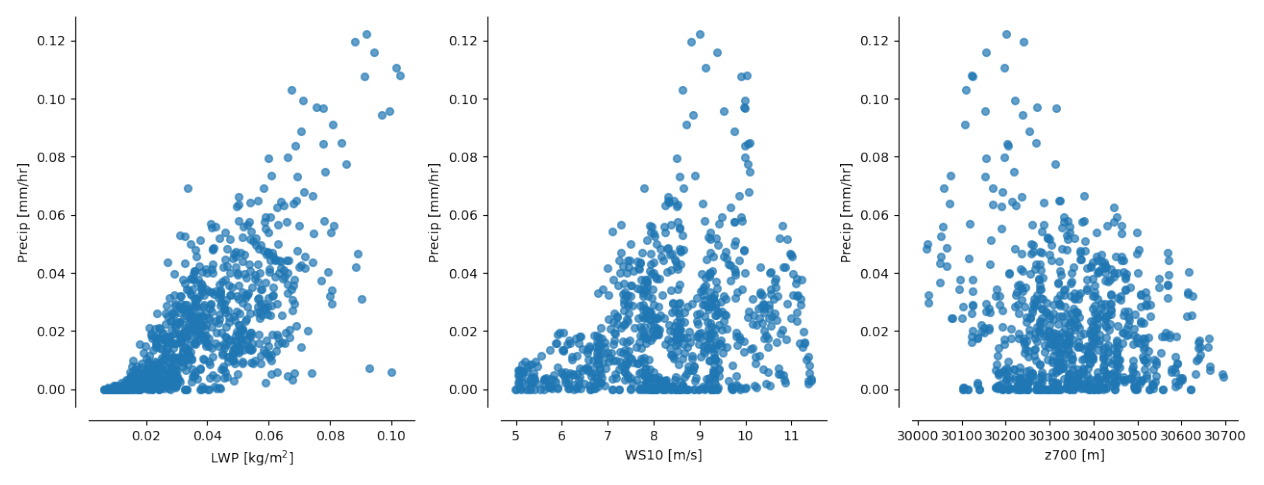
[3]:
```
precip = df_crm['precip'].to_numpy().reshape(-1 ,1)
```

5.3.1 Visualize the precipitation landscape

In the ensemble, shallow precipitation is highly correlated with many different physical features. Most obviously there is a high correlation with liquid water path (`LWP`), 10-metre windspeed (`WS10`) and geopotential height at 700hPa (`z_{700}`).

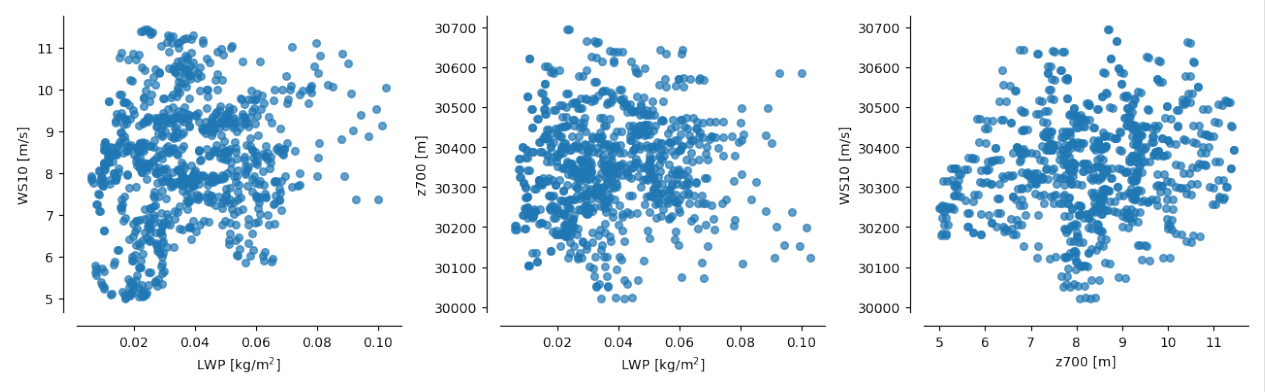
We can use these correlations to create "collapsing spaces" for investigating the relationships between shallow precipitation and the local meteorological environment.

[4]:
```
fig, axs = plt.subplots(ncols=3, figsize=(13,5), dpi=100, gridspec_kw={'width_ratios':[1,
↪1,1]})

axs[0].scatter(df_crm['LWP'], df_crm['precip'], s=30, marker='o', alpha=0.7, label=r'CDNC
↪$_{20}$ and CDNC$_{200}$')
axs[0].set_xlabel('LWP [kg/m$^{2}$]')
axs[0].set_ylabel('Precip [mm/hr]')
prettify_plot(axs[0])

axs[1].scatter(df_crm['WS10'], df_crm['precip'], s=30, marker='o', alpha=0.7, label=r
↪'CDNC$_{20}$ and CDNC$_{200}$')
axs[1].set_xlabel('WS10 [m/s]')
axs[1].set_ylabel('Precip [mm/hr]')
prettify_plot(axs[1])

axs[2].scatter(df_crm['zg700'], df_crm['precip'], s=30, marker='o', alpha=0.7, label=r
↪'CDNC$_{20}$ and CDNC$_{200}$')
axs[2].set_xlabel('z700 [m]')
axs[2].set_ylabel('Precip [mm/hr]')
prettify_plot(axs[2])

fig.tight_layout()
#plt.savefig("Figs/1hr_D13shal_lwp-ws-pr.png", dpi=200)
```



Also, good to note that each of these predictors ``(LWP, WS10, z700)`` are mutually uncorrelated (see plots below)

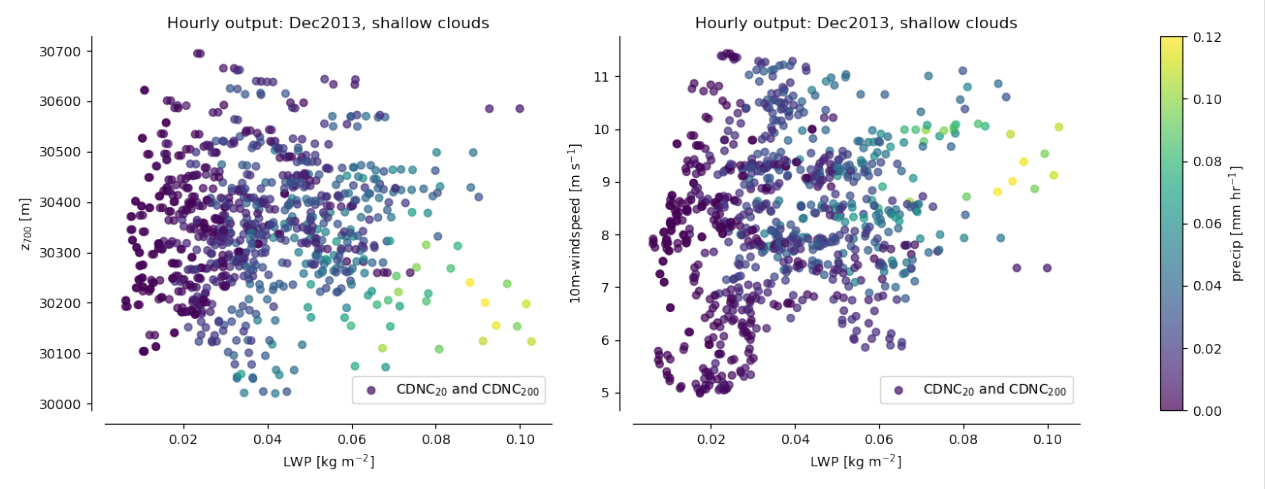
[5]:
```
fig, axs = plt.subplots(ncols=3, figsize=(13,4), dpi=100, gridspec_kw={'width_ratios':[1,
↪1,1]})

axs[0].scatter(df_crm['LWP'], df_crm['WS10'], s=30, marker='o', alpha=0.7, label=r'CDNC$_
↪{20}$ and CDNC$_{200}$')
axs[0].set_xlabel('LWP [kg/m$^{2}$]')
axs[0].set_ylabel('WS10 [m/s]')
prettify_plot(axs[0])

axs[1].scatter(df_crm['LWP'], df_crm['zg700'], s=30, marker='o', alpha=0.7, label=r'CDNC
↪$_{20}$ and CDNC$_{200}$')
axs[1].set_xlabel('LWP [kg/m$^{2}$]')
axs[1].set_ylabel('z700 [m]')
prettify_plot(axs[1])

axs[2].scatter(df_crm['WS10'], df_crm['zg700'], s=30, marker='o', alpha=0.7, label=r'CDNC
↪$_{20}$ and CDNC$_{200}$')
axs[2].set_xlabel('z700 [m]')
axs[2].set_ylabel('WS10 [m/s]')
prettify_plot(axs[2])

fig.tight_layout()
#plt.savefig("Figs/1hr_D13shal_lwp-ws-pr.png", dpi=200)
```



Stratifying precip by pairs of these predictors creates nice "collapsing spaces"

Nice for illustrating how the emulated surface compares to the raw data

[6]:
```
fig, axs = plt.subplots(ncols=3, figsize=(13,5), dpi=100, gridspec_kw={'width_ratios':[1,
↪1,0.05]})

sc1 = axs[0].scatter(df_crm['LWP'], df_crm['zg700'], c=df_crm['precip'],
                vmin=0, vmax=0.12, s=30, marker='o', alpha=0.7, label=r'CDNC$_{20}$ and␣
↪CDNC$_{200}$')

axs[0].set_xlabel(r'LWP [kg m$^{-2}$]')
axs[0].set_ylabel(r'z$_{700}$ [m]')

axs[0].set_title("Hourly output: Dec2013, shallow clouds")
axs[0].legend(loc='lower right')

prettify_plot(axs[0])


sc2 = axs[1].scatter(df_crm['LWP'], df_crm['WS10'], c=df_crm['precip'],
                vmin=0, vmax=0.12, s=30, marker='o', alpha=0.7, label=r'CDNC$_{20}$ and␣
↪CDNC$_{200}$')

axs[1].set_xlabel(r'LWP [kg m$^{-2}$]')
axs[1].set_ylabel(r'10m-windspeed [m s$^{-1}$]')

axs[1].set_title("Hourly output: Dec2013, shallow clouds")
axs[1].legend(loc='lower right')

prettify_plot(axs[1])

fig.colorbar(sc1, cax=axs[2], label=r'precip [mm hr$^{-1}$]')

fig.tight_layout()
#plt.savefig("Figs/1hr_D13shal_lwp-ws-pr.png", dpi=200)
```



5.3.2 Emulation

Our aim is to emulate shallow precipitation as a function of the environmental conditions, and then plot the predictions in `LWP-z700` space to compare with the scatter points above.

To do this we choose a set of predictors which are typical "cloud-controlling factors" such as SST, Estimated Inversion Strength, vertical velocity at 700 hPa, LWP and z700. Other variables could also be chosen and it's worth exploring this just to get a sense for how the model behaves.

After validating the model using Leave-One-Out cross-validation, we then retrain the model using the full dataset, and use this model to predict the precipitation across a wide range of values. Finally, for the purpose of plotting in `LWP-z700` space, we reduce the dimensionality of our final prediction by averaging over all features with aren't `LWP` or `z700`. This gives us a smooth field to compare with the scatter points.

[7]:
```
params = df_crm.loc[:, ['LWP', 'zg700', 'EIS', 'SST', 'w700']]

print("The input params are: \n", params, "\n")
```

```
The input params are:
          LWP      zg700       EIS         SST      w700
0    0.035898  30541.566  0.989443  301.173248 -0.010275
1    0.044468  30488.172  1.130803  301.173248 -0.008676
2    0.051559  30442.703  1.242463  301.173248 -0.006025
3    0.057509  30390.662  1.304206  301.173248 -0.004758
4    0.064226  30364.852  1.362710  301.173248 -0.003158
..        ...        ...       ...         ...       ...
845  0.064794  30222.945 -2.696190  300.126465 -0.007968
846  0.063914  30214.715 -2.673406  300.126465 -0.003904
847  0.059974  30236.312 -2.643809  300.126465 -0.000284
848  0.062895  30286.500 -2.635981  300.126465  0.000667
849  0.069100  30322.684 -2.612224  300.126465  0.000406

[850 rows x 5 columns]
```

LeaveOneOut cross-validation and plotting

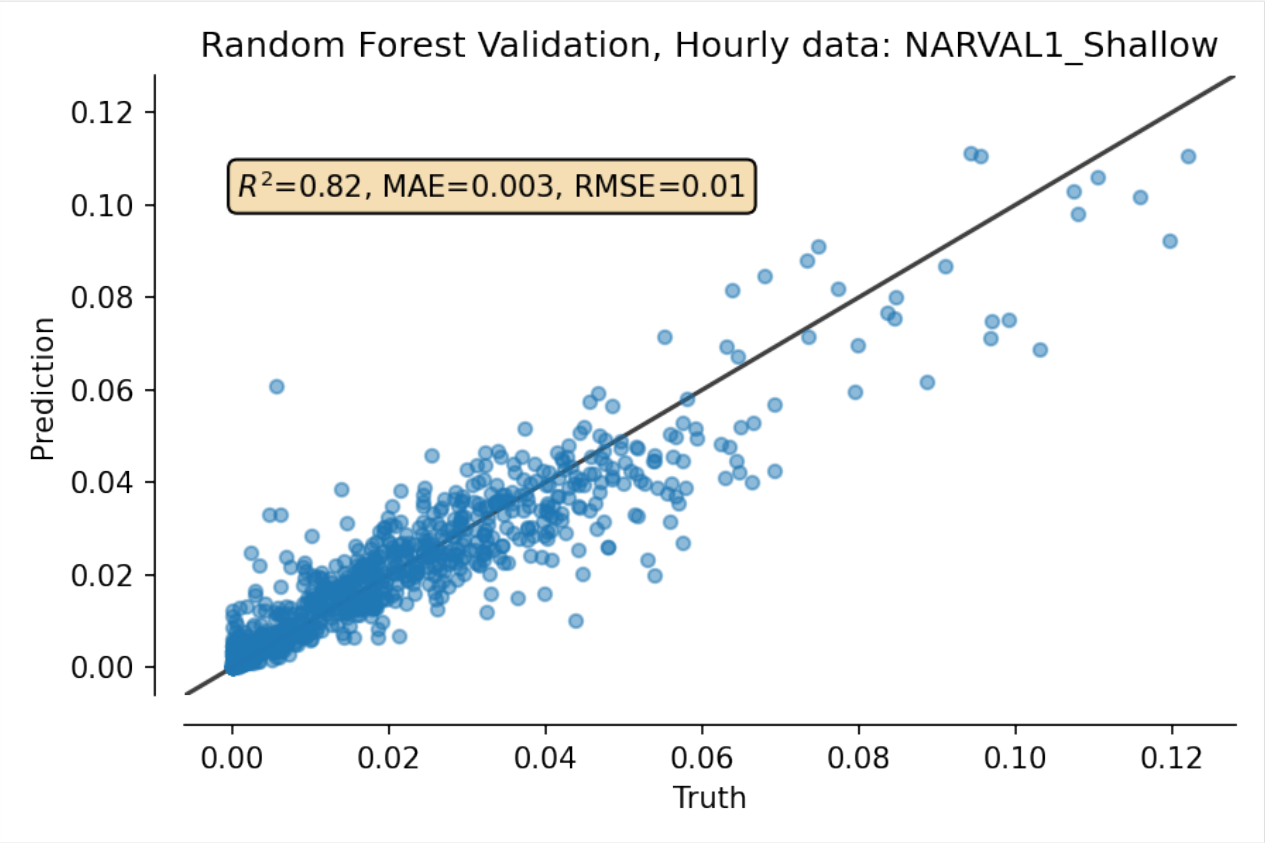
[8]:
```
%%time

# Ignore the mountain of warnings
import warnings
from sklearn.exceptions import DataConversionWarning
warnings.filterwarnings(action='ignore', category=DataConversionWarning)

outp = np.asarray(leave_one_out(Xdata=params, Ydata=precip, model='RandomForest', n_
↪estimators=50, random_state=0))
```

```
Wall time: 2min 12s
```

```
<timed exec>:6: VisibleDeprecationWarning: Creating an ndarray from ragged nested␣
↪sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different␣
↪lengths or shapes) is deprecated. If you meant to do this, you must specify
↪'dtype=object' when creating the ndarray.
```

[9]:
```
truth_RF, pred_RF = outp[:,0], outp[:,1]
```

[10]:
```
from sklearn.metrics import mean_squared_error

""" Validation plot """
fig, ax = plt.subplots(dpi=150)

plot_results(ax, truth_RF, pred_RF, title="Random Forest Validation, Hourly data:␣
↪NARVAL1_Shallow")

fig.tight_layout()
```



Now, retrain model on all data, and extrapolate over whole space

[11]:
```
X_train = params.to_numpy()
Y_train = precip.ravel()
model = rf_model(X_train, Y_train)
```

[12]:
```
model.train()
```

[13]:
```
%%time

# Now, make grid for plotting RF predictions
# more n_points means higher resolution, but takes exponentially longer
n_points = 30

min_vals = params.min()
max_vals = params.max()

# For uniform prediction over full params space
space=np.linspace(min_vals, max_vals, n_points)

# Reshape to (N,D)
reshape_to_ND = np.transpose(space)
Xs_uniform = np.meshgrid(*reshape_to_ND)
test = np.array([_.flatten() for _ in Xs_uniform]).T
```

(continues on next page)

(continued from previous page)

```
# Predict
predictions,_ = model.predict(test)
predictions   = predictions.reshape(Xs_uniform[0].shape)

# Now, take mean over all parameters except [LWP, z700], assumed to be first 2 indices
predictions_reduced = np.mean(predictions, axis=tuple(range(2, predictions.ndim)))
```

```
Wall time: 1min 35s
```

[14]:
```
# Now, make grid for plotting RF predictions

LWP_grid   = np.linspace(min_vals['LWP'], max_vals['LWP'], num=n_points)
zg700_grid = np.linspace(min_vals['zg700'], max_vals['zg700'], num=n_points)

lwp, zg = np.meshgrid(LWP_grid, zg700_grid)
```

5.3.3 Plot!

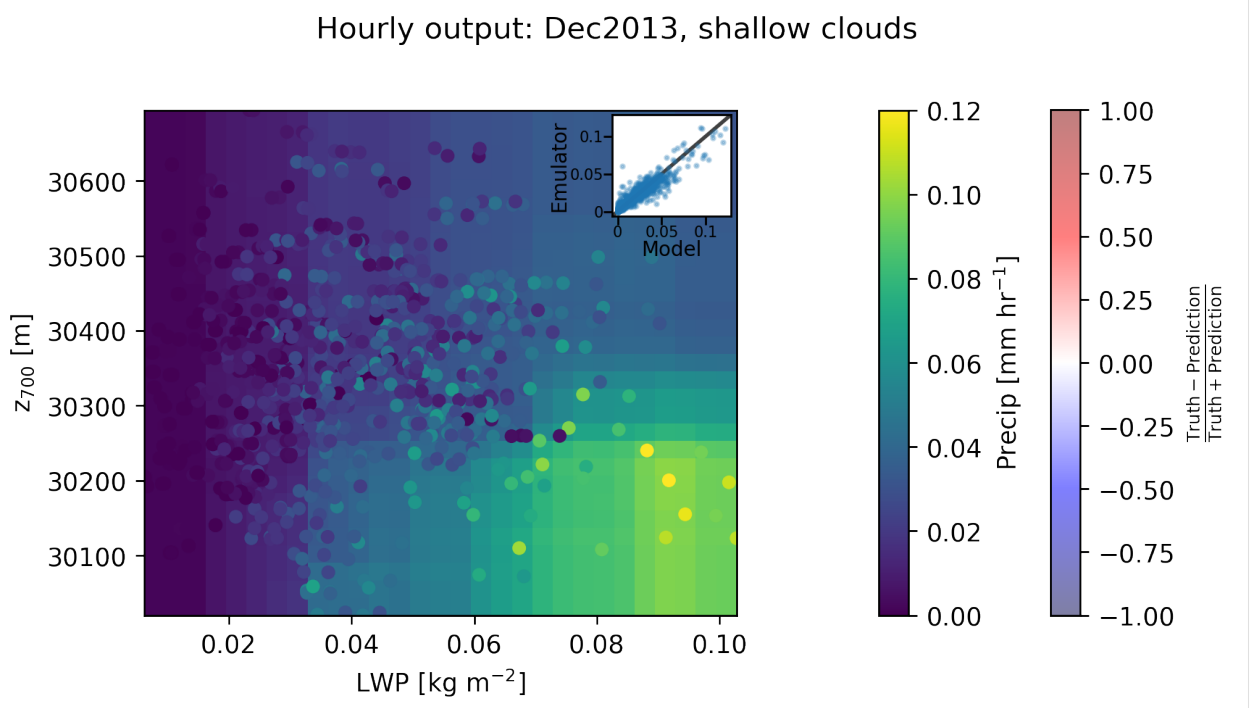
[15]:
```
fig, ax = plt.subplots(ncols=3, figsize=(7,4), dpi=250, gridspec_kw={'width_ratios':[1,␣
↪0.05, 0.05]})
fig.suptitle("Hourly output: Dec2013, shallow clouds")

cp = ax[0].pcolormesh(LWP_grid, zg700_grid,
                      predictions_reduced, vmin=0, vmax=0.12, alpha=1)

fig.colorbar(cp, cax=ax[1], orientation='vertical', shrink=0.05, label=r'Precip [mm hr$^
↪{-1}$]')

"""Overlap errors"""
ax[0].scatter(df_crm['LWP'], df_crm['zg700'], c=df_crm['precip'],
                    vmin=0, vmax=0.12, s=30, marker='o', edgecolors="None")

ers = ax[0].scatter(df_crm['LWP'], df_crm['zg700'], c=(truth_RF-pred_RF)/(truth_RF+pred_
↪RF), facecolors="None",
                    vmin=-1, vmax=1, s=30, marker='o', lw=0.7, alpha=0.5, cmap='seismic')
ers.set_facecolors("None")
fig.colorbar(ers, cax=ax[2], label=r'$\frac{\mathrm{Truth-Prediction}}{\mathrm
↪{Truth+Prediction}}$')

for idx, _ in enumerate(ax[:1]):
    _.set_xlabel(r'LWP [kg m$^{-2}$]')
    _.set_ylabel(r'z$_{700}$ [m]')

    if idx==0:
        _.set_xlim(min_vals['LWP'], max_vals['LWP'])
        _.set_ylim(min_vals['zg700'], max_vals['zg700'])


fig.tight_layout()
```

(continues on next page)

(continued from previous page)

```
fig.subplots_adjust(top=0.85) # Put this AFTER tight_layout() call !

"""Add validation plot as inset to first axis"""
axins = ax[0].inset_axes([0.79, 0.79, 0.2, 0.2], # x0, y0, width, height
                         transform=ax[0].transAxes)
axins.scatter(truth_RF, pred_RF, s=2, alpha=0.3)
add_121_line(axins)
axins.set_xlabel('Model', position=(0.5,0), fontsize=8, labelpad=-0.01)
axins.set_xticks([0, 0.05, 0.1])
axins.set_xticklabels(labels=[0, 0.05, 0.1], fontdict={'fontsize':6})

axins.set_ylabel('Emulator', position=(0,0.5), fontsize=8, labelpad=-0.01)
axins.set_yticks([0, 0.05, 0.1])
axins.set_yticklabels(labels=[0, 0.05, 0.1], fontdict={'fontsize':6})

axins.tick_params(axis='both', which='major', pad=0.01)

#plt.savefig("./Figs/1hr_D13shal_lwp-zg700-pr_w-errorpoints.png", dpi=300)
```

```
C:\Users\duncan\AppData\Local\Temp/ipykernel_21624/1073200818.py:4:␣
↪MatplotlibDeprecationWarning: shading='flat' when X and Y have the same dimensions as␣
↪C is deprecated since 3.3.  Either specify the corners of the quadrilaterals with X␣
↪and Y, or pass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading
↪'].  This will become an error two minor releases later.
  cp = ax[0].pcolormesh(LWP_grid, zg700_grid,
```




```
[ ]:
```

5.4 Calibrating GPs using ABC

[1]:
```
import warnings
warnings.filterwarnings('ignore') # Ignore all the iris warnings...
```

[2]:
```
import pandas as pd
import cis
import iris

from utils import get_aeronet_data, get_bc_ppe_data

from esem.utils import validation_plot, plot_parameter_space, get_random_params,␣
↪ensemble_collocate
from esem import gp_model
from esem.abc_sampler import ABCSampler, constrain

import iris.quickplot as qplt
import matplotlib.pyplot as plt
%matplotlib inline
```

5.4.1 Read in the parameters and observables

[3]:
```
aaod = get_aeronet_data()
print(aaod)
```

```
Ungridded data: Absorption_AOD440nm / (1)
     Shape = (10098,)

     Total number of points = 10098
     Number of non-masked points = 10098
     Long name = Absorption_AOD440nm
     Standard name = None
     Units = 1
     Missing value = -999.0
     Range = (5.1e-05, 0.47236)
     History =
     Coordinates:
       longitude
          Long name =
          Standard name = longitude
          Units = degrees_east
          Missing value = None
          Range = (-155.576755, 141.3407)
          History =
       latitude
          Long name =
          Standard name = latitude
          Units = degrees_north
          Missing value = None
          Range = (-35.495807, 79.990278)
```

(continues on next page)

(continued from previous page)

```
          History =
       time
          Long name =
          Standard name = time
          Units = days since 1600-01-01 00:00:00
          Missing value = None
          Range = (cftime.DatetimeGregorian(2017, 1, 1, 12, 0, 0, 0, has_year_
↪zero=False), cftime.DatetimeGregorian(2018, 1, 2, 12, 0, 0, 0, has_year_zero=False))
          History =
```

[4]:
```
# Read in the PPE parameters, AAOD and DRE
ppe_params, ppe_aaod, ppe_dre = get_bc_ppe_data(dre=True)
```

[5]:
```
# Take the annual mean of the DRE
ppe_dre, = ppe_dre.collapsed('time', iris.analysis.MEAN)
```

```
WARNING:root:Creating guessed bounds as none exist in file
WARNING:root:Creating guessed bounds as none exist in file
WARNING:root:Creating guessed bounds as none exist in file
WARNING:root:Creating guessed bounds as none exist in file
```

5.4.2 Collocate the model on to the observations

[6]:
```
col_ppe_aaod = ensemble_collocate(ppe_aaod, aaod)
```

[7]:
```
n_test = 8

X_test, X_train = ppe_params[:n_test], ppe_params[n_test:]
Y_test, Y_train = col_ppe_aaod[:n_test], col_ppe_aaod[n_test:]
```

[8]:
```
Y_train
```

[8]:
```
<iris 'Cube' of Absorption optical thickness - total 550nm / (1) (job: 31; obs: 10098)>
```

5.4.3 Setup and run the models

Explore different model choices

[9]:
```
from esem.utils import leave_one_out, prediction_within_ci
from scipy import stats
import numpy as np

from esem.data_processors import Log
res_l = leave_one_out(X_train, Y_train, model='GaussianProcess', data_
↪processors=[Log(constant=0.1)], kernel=['Linear', 'Exponential', 'Bias'])
r2_values_l = [stats.linregress(x.data.compressed(), y.data[:, ~x.data.mask].
↪flatten())[2]**2 for x,y,_ in res_l]
```

(continues on next page)

(continued from previous page)

```
ci95_values_l = [prediction_within_ci(x.data.flatten(), y.data.flatten(), v.data.
↪flatten())[2].sum()/x.data.count() for x,y,v in res_l]
print("Mean R^2: {:.2f}".format(np.asarray(r2_values_l).mean()))
print("Mean proportion within 95% CI: {:.2f}".format(np.asarray(ci95_values_l).mean()))

res = leave_one_out(X_train, Y_train, model='GaussianProcess', kernel=['Linear', 'Bias'])
r2_values = [stats.linregress(x.data.flatten(), y.data.flatten())[2]**2 for x,y,v in res]
ci95_values = [prediction_within_ci(x.data.flatten(), y.data.flatten(), v.data.
↪flatten())[2].sum()/x.data.count() for x,y,v in res]
print("Mean R^2: {:.2f}".format(np.asarray(r2_values).mean()))
print("Mean proportion within 95% CI: {:.2f}".format(np.asarray(ci95_values).mean()))

# Note that while the Log pre-processing leads to slightly better R^2, the model is␣
↪under-confident and
#  has too large uncertainties which would adversley effec our implausibility metric.
```

```
Mean R^2: 1.00
Mean proportion within 95% CI: 1.00
Mean R^2: 0.99
Mean proportion within 95% CI: 0.94
```

Build final model

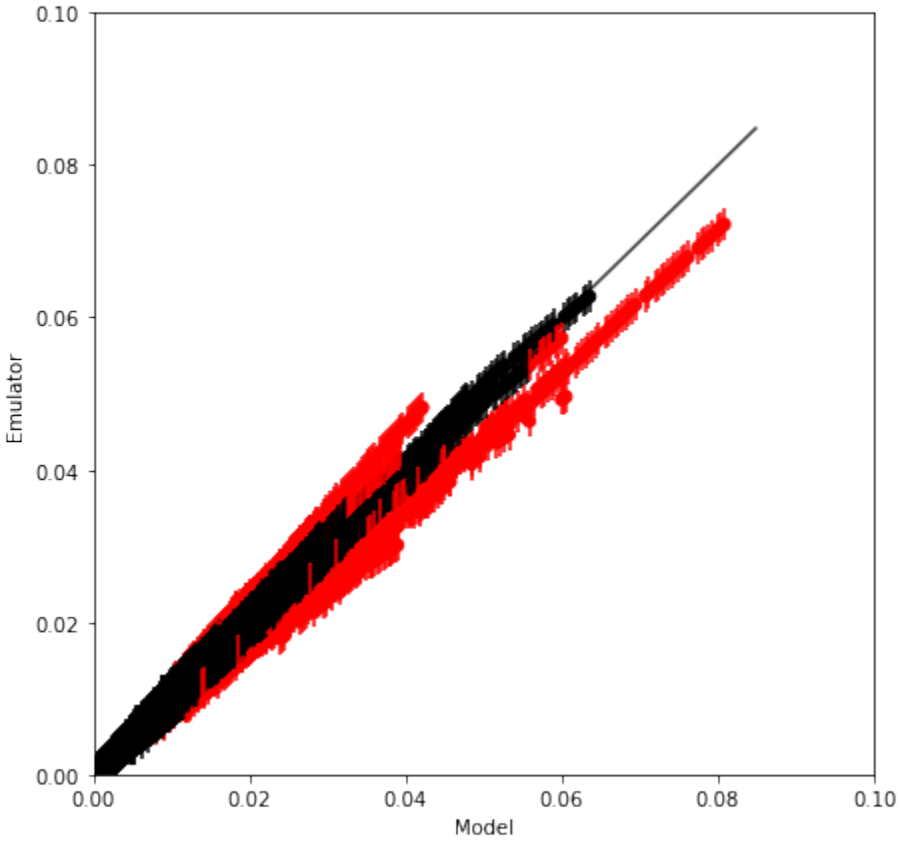
[10]:
```
model = gp_model(X_train, Y_train, kernel=['Linear', 'Bias'])
```

[11]:
```
model.train()
```

[12]:
```
m, v = model.predict(X_test.values)
```

[13]:
```
validation_plot(Y_test.data.flatten(), m.data.flatten(), v.data.flatten(),
                minx=0, maxx=0.1, miny=0., maxy=0.1)
```

```
Proportion of 'Bad' estimates : 5.52%
```



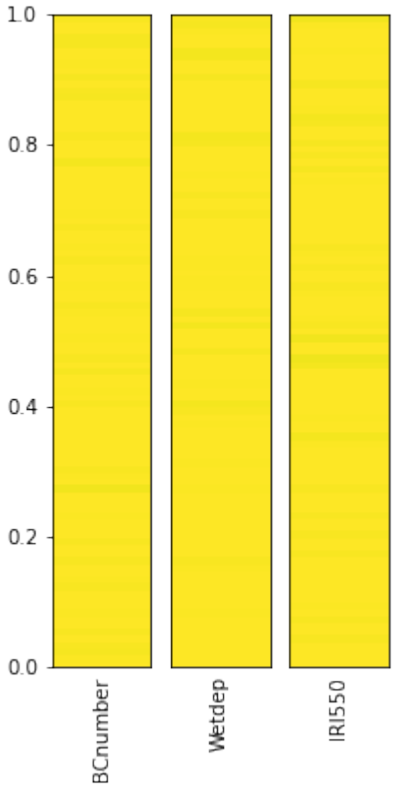
5.4.4 Sample and constrain the models

Emulating 1e6 sample points directly would require 673 Gb of memory so we can either run 1e6 samples for each point, or run the constraint everywhere, but in batches. Here we do the latter, optioanlly on the GPU, using the 'naive' algorithm for calculating the running mean and variance of the various properties.

The rejection sampling happens in a similar manner so that only as much memory as is used for one batch is ever used.

[14]:
```
# In this case
sample_points = pd.DataFrame(data=get_random_params(3, int(1e6)), columns=X_train.
↪columns)
```

[15]:
```
# Note that smoothing the parameter distribution can be slow for large numbers of points
plot_parameter_space(sample_points, fig_size=(3,6), smooth=False)
```



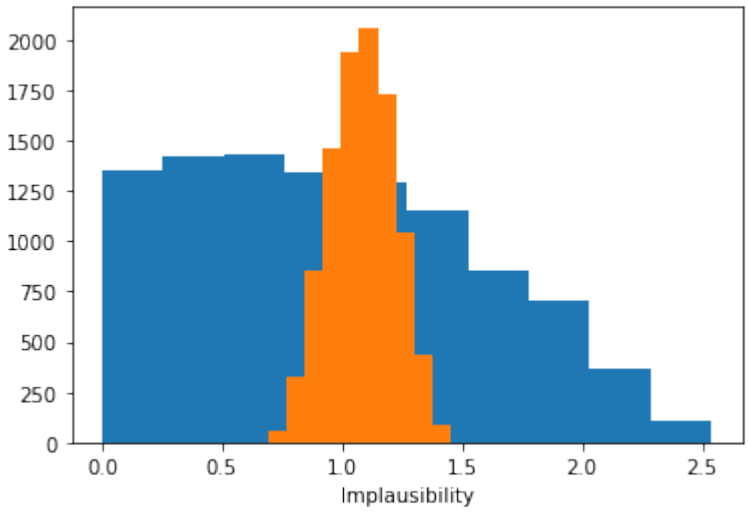
[16]:
```
# Setup the sampler to compare against our AeroNet data
sampler = ABCSampler(model, aaod, obs_uncertainty=0.5, repres_uncertainty=0.5)
```

[17]:
```
# Calculate the implausibilty for each sample against each observation - note this can␣
↪be very large so we only sample a fraction!
implaus = sampler.get_implausibility(sample_points[::100], batch_size=1000)

# The implausibility distributions for different observations can be very different.
_ = plt.hist(implaus.data[:, 1400])
_ = plt.hist(implaus.data[:, 14])
plt.gca().set(xlabel='Implausibility')
```

```
  0%|          | 0/10000 [00:00<?, ?sample/s]
```

```
<tqdm.auto.tqdm object at 0x00000236A10BA5E0>
```

[17]:
```
[Text(0.5, 0, 'Implausibility')]
```

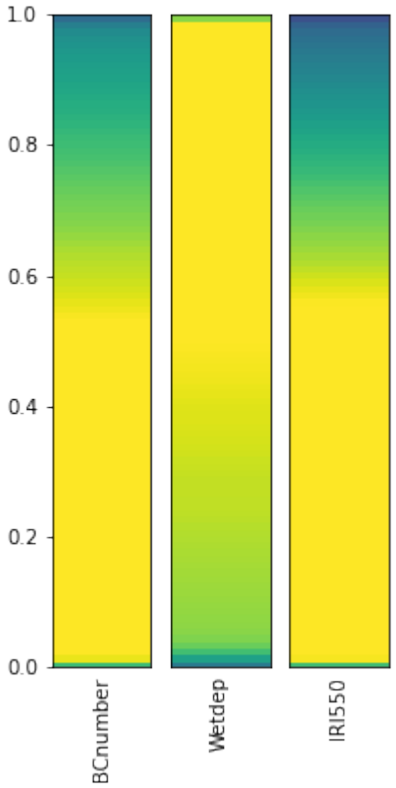


[18]:
```
# Find the valid samples in our full 1million samples by comparing against a given␣
↪tolerance and threshold
valid_samples = sampler.batch_constrain(sample_points, batch_size=10000, tolerance=.1)
print("Remaining points: {}".format(valid_samples.sum()))
```

```
  0%|          | 0/1000000 [00:00<?, ?sample/s]
```

```
<tqdm.auto.tqdm object at 0x000002398006DC70>Remaining points: 729474
```

[19]:
```
# Plot the reduced parameter distribution
constrained_sample = sample_points[valid_samples]
plot_parameter_space(constrained_sample, fig_size=(3,6))
```

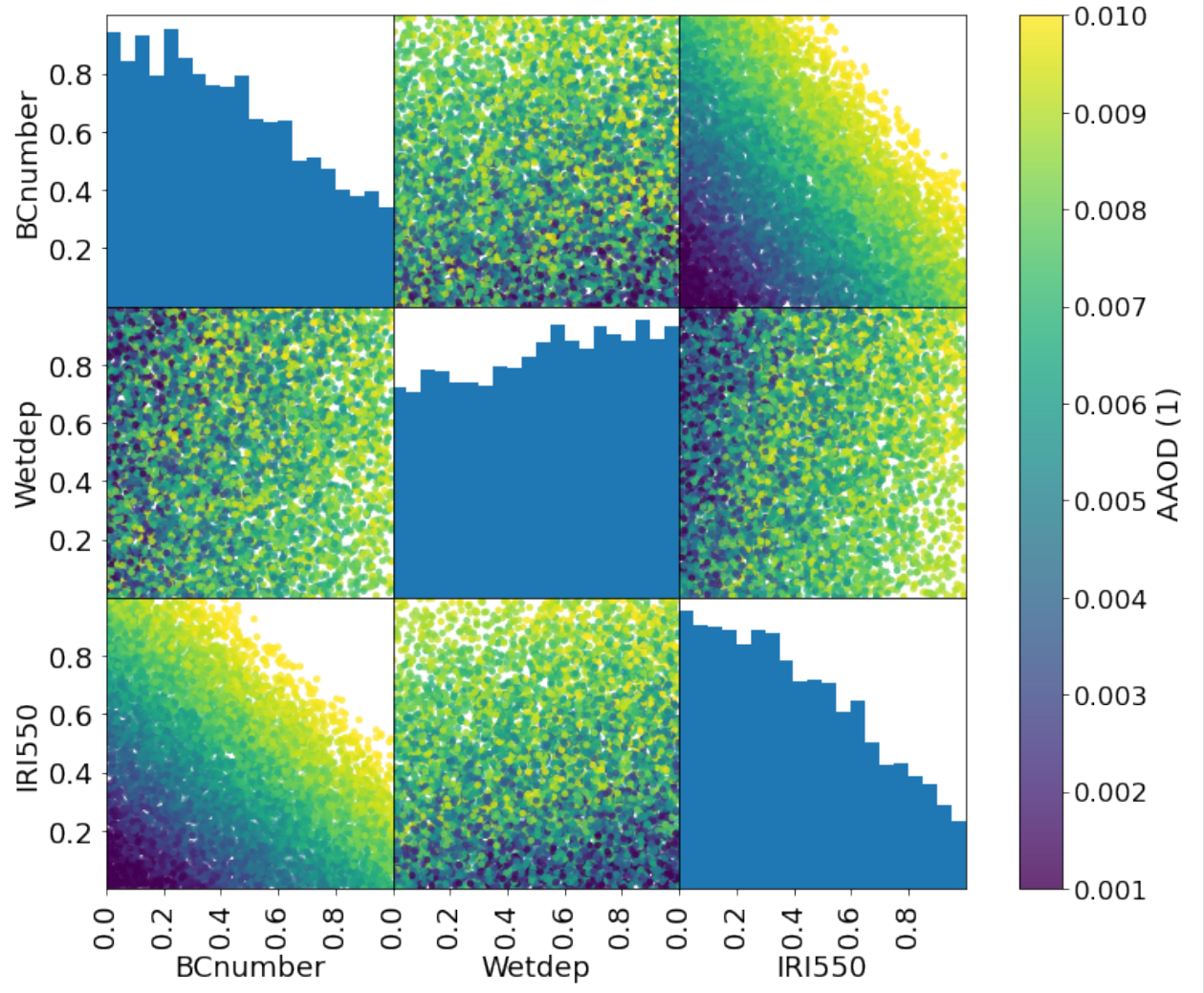


[20]:
```
# We can also easily plot the joint distributions
#  Only plot every one in 100 points as scatter plots with large numbers of points are␣
↪slow...
import matplotlib

# Mimic Seaborn scaling without requiring the whole package
scale = 1.5
matplotlib.rcParams['font.size'] = 12 * scale
matplotlib.rcParams['axes.labelsize'] = 12 * scale
matplotlib.rcParams['axes.titlesize'] = 12 * scale
matplotlib.rcParams['xtick.labelsize'] = 11 * scale
matplotlib.rcParams['ytick.labelsize'] = 11 * scale
matplotlib.rcParams['lines.linewidth'] = 1.5 * scale
matplotlib.rcParams['lines.markersize'] = 6 * scale
#

m, _ = model.predict(constrained_sample[::100].values)
Zs = m.data
# Plot the emulated AAOD value (averaged over observation locations) for each point
grr = pd.plotting.scatter_matrix(constrained_sample[::100], c=Zs.mean(axis=1),␣
↪figsize=(12, 10), marker='o',
                                 hist_kwds={'bins': 20,}, s=20, alpha=.8, vmin=1e-3,␣
↪vmax=1e-2, range_padding=0.,
                                 density_kwds={'range': [[0., 1.], [0., 1.]], 'colormap':
↪'viridis'},
                                 )
```

(continues on next page)

(continued from previous page)

```
# Matplotlib dragons...
grr[0][0].set_yticklabels([0.2, 0.4, 0.6, 0.8], fontsize=12 * scale)
for i in range(2):
    grr[i+1][0].set_yticklabels([0.0, 0.2, 0.4, 0.6, 0.8], fontsize=12 * scale)
for i in range(3):
    grr[2][i].set_xticks([0.0, 0.2, 0.4, 0.6, 0.8])
    grr[2][i].set_xticklabels([0.0, 0.2, 0.4, 0.6, 0.8], fontsize=12 * scale)

plt.colorbar(grr[0][1].collections[0], ax=grr, use_gridspec=True, label='AAOD (1)')

plt.savefig('BCPPE_constrained_params_paper.png', transparent=True)
```



5.4.5 Explore the uncertainty in Direct Radiative Effect of Aerosol in constrianed sample-space

[21]:
```
dre_test, dre_train = ppe_dre[:n_test], ppe_dre[n_test:]

ari_model = gp_model(X_train, dre_train, name="ARI", kernel=['Linear', 'Bias'])
ari_model.train()
```

[22]:
```
# Calculate the mean and std-dev DRE over each set of sample points

unconstrained_mean_ari, unconstrained_sd_ari = ari_model.batch_stats(sample_points,␣
↪batch_size=10000)
constrained_mean_ari, constrained_sd_ari = ari_model.batch_stats(constrained_sample,␣
↪batch_size=10000)
```

```
  0%|          | 0/1000000 [00:00<?, ?sample/s]
```

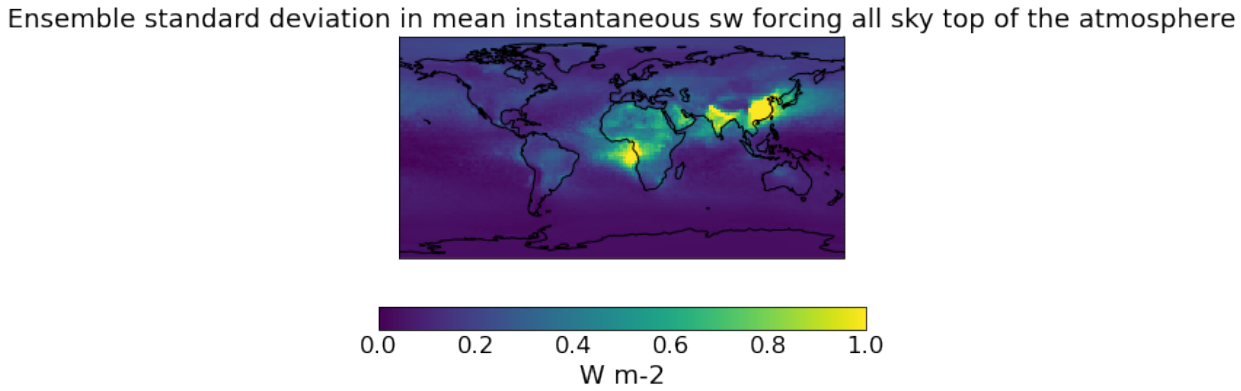
```
<tqdm.auto.tqdm object at 0x00000239A2B2D310>
```

```
  0%|          | 0/729474 [00:00<?, ?sample/s]
```

```
<tqdm.auto.tqdm object at 0x00000239ACFEFDF0>
```

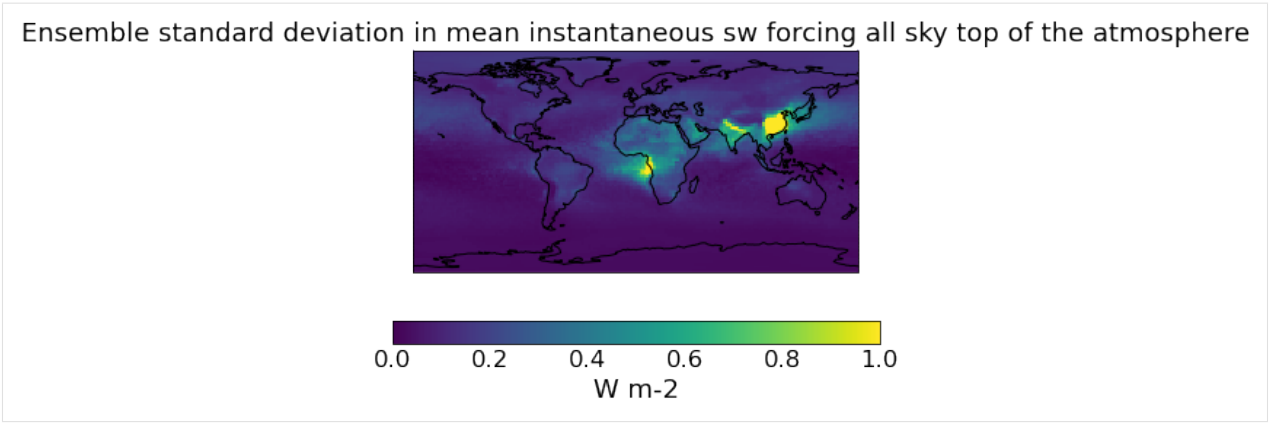
[23]:
```
# The original (unconstrained DRE)
qplt.pcolormesh(unconstrained_sd_ari, vmin=0., vmax=1)
plt.gca().coastlines()
```

[23]:
```
<cartopy.mpl.feature_artist.FeatureArtist at 0x239ad149610>
```



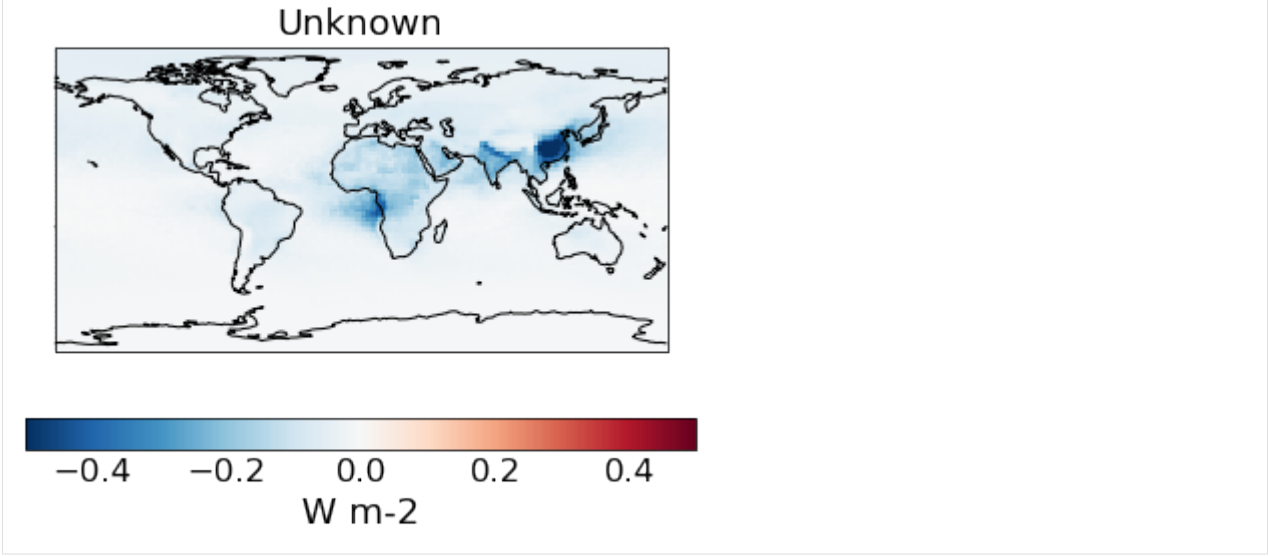
[24]:
```
# The constrained DRE
qplt.pcolormesh(constrained_sd_ari, vmin=0., vmax=1)
plt.gca().coastlines()
```

[24]:
```
<cartopy.mpl.feature_artist.FeatureArtist at 0x239af7e3700>
```



[25]:
```
# The change in spread after the constraint is applied
qplt.pcolormesh((constrained_sd_ari-unconstrained_sd_ari), cmap='RdBu_r', vmin=-5e-1,␣
↪vmax=5e-1)
plt.gca().coastlines()
```

[25]:
```
<cartopy.mpl.feature_artist.FeatureArtist at 0x239af86c310>
```



[ ]:
```
```

5.5 Calibrating GPs using MCMC

[1]:
```
import os
# Ignore my broken HDF5 install...
os.putenv("HDF5_DISABLE_VERSION_CHECK", '1')
```

[2]:
```
import pandas as pd
import numpy as np
import iris
```

(continues on next page)

(continued from previous page)

```
from utils import get_aeronet_data, get_bc_ppe_data
from esem import gp_model
from esem.sampler import MCMCSampler

import os

import matplotlib.pyplot as plt
%matplotlib inline

# GPU = "1"

# os.environ["CUDA_VISIBLE_DEVICES"] = GPU
```

5.5.1 Read in the parameters and observables

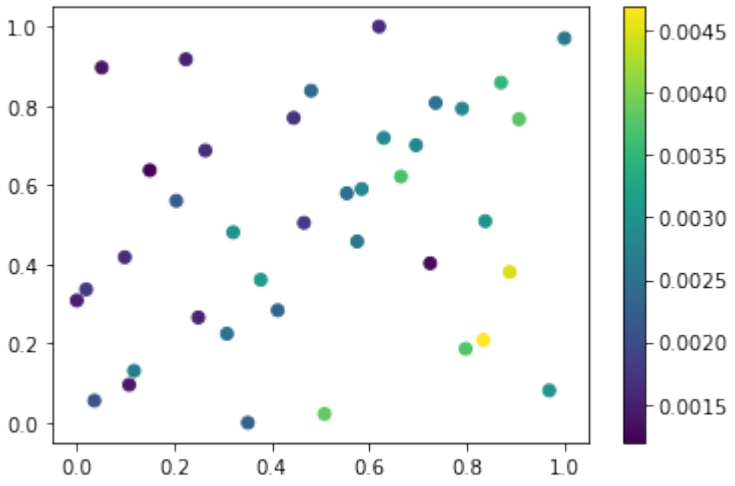
[3]:
```
ppe_params, ppe_aaod = get_bc_ppe_data()
```

[4]:
```
# Calculate the global, annual mean AAOD (CIS will automatically apply the weights)
mean_aaod, = ppe_aaod.collapsed(['latitude', 'longitude', 'time'], 'mean')
```

```
WARNING:root:Creating guessed bounds as none exist in file
WARNING:root:Creating guessed bounds as none exist in file
WARNING:root:Creating guessed bounds as none exist in file
C:\Users\duncan\miniconda3\envs\climatebench\lib\site-packages\iris\analysis\cartography.
↪py:394: UserWarning: Using DEFAULT_SPHERICAL_EARTH_RADIUS.
  warnings.warn("Using DEFAULT_SPHERICAL_EARTH_RADIUS.")
```

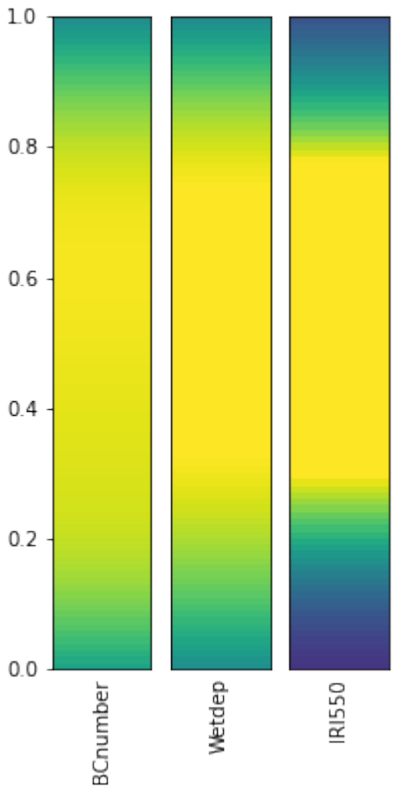
[5]:
```
plt.scatter(ppe_params.BCnumber, ppe_params.Wetdep, c=mean_aaod.data)
plt.colorbar()
```

[5]:
```
<matplotlib.colorbar.Colorbar at 0x1807b701fd0>
```



[6]:
```
from esem.utils import plot_parameter_space

plot_parameter_space(ppe_params, fig_size=(3,6))
```

```
c:\users\duncan\pycharmprojects\gcem\esem\utils.py:119: MatplotlibDeprecationWarning:␣
↪shading='flat' when X and Y have the same dimensions as C is deprecated since 3.3. ␣
↪Either specify the corners of the quadrilaterals with X and Y, or pass shading='auto',
↪'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will become an error␣
↪two minor releases later.
  ax.pcolor(X, Y, vals[:, np.newaxis], vmin=0, vmax=1)
```



[7]:
```
n_test = 8

X_test, X_train = ppe_params[:n_test], ppe_params[n_test:]
Y_test, Y_train = mean_aaod[:n_test], mean_aaod[n_test:]
```

5.5.2 Setup and run the models

[8]:
```
model = gp_model(X_train, Y_train)
```

```
WARNING: Using default kernel - be sure you understand the assumptions this implies.␣
↪Consult e.g.  http://www.cs.toronto.edu/~duvenaud/cookbook/ for an excellent␣
↪description of different kernel choices.
```

[9]:
```
model.train()
```

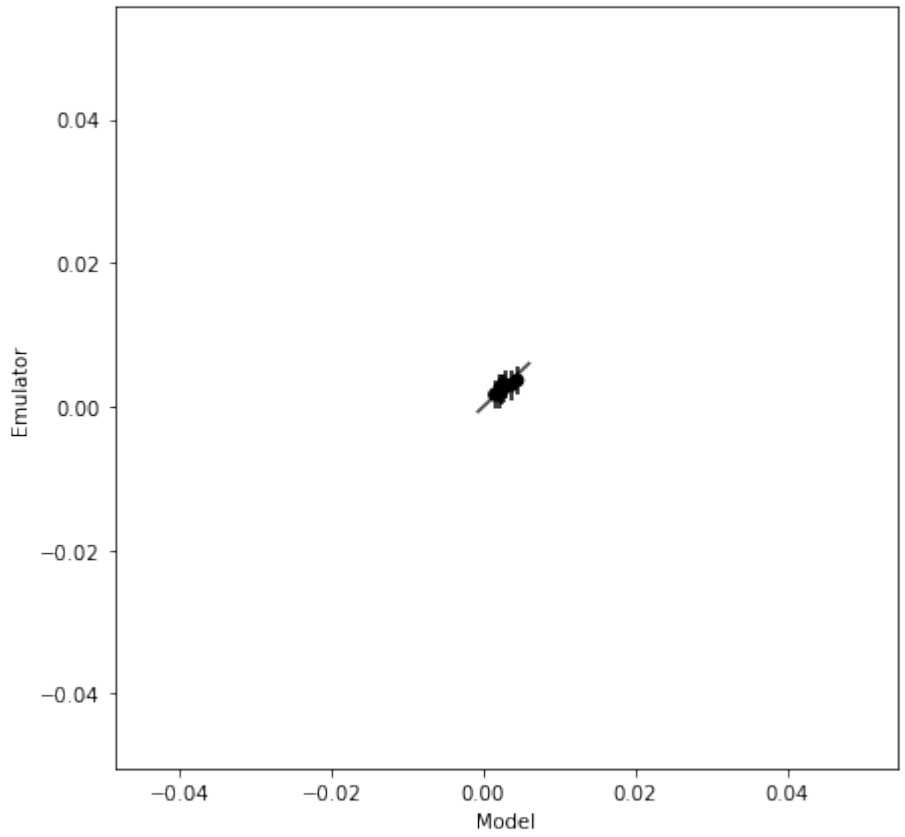
[10]:
```
m, v = model.predict(X_test.values)
```

[11]:
```
Y_test.data
```

[11]:
```
masked_array(data=[0.002822510314728863, 0.002615034735649063,
                   0.002186747212596059, 0.0015112621889264352,
                   0.004456776854431466, 0.0025203727123839117,
                   0.0022378865435589133, 0.003730134076583199],
             mask=[False, False, False, False, False, False, False, False],
       fill_value=1e+20)
```

[12]:
```
from esem.utils import validation_plot

validation_plot(Y_test.data.flatten(), m.data.flatten(), v.data.flatten())
```

```
Proportion of 'Bad' estimates : 0.00%
```



[13]:
```
# Set the objective as one of the test datasets
sampler = MCMCSampler(model, Y_test[0])
```

[14]:
```
samples = sampler.sample(n_samples=800, mcmc_kwargs=dict(num_burnin_steps=100) )
```

```
Acceptance rate: 0.9262387969566703
```

[15]:
```
new_samples = pd.DataFrame(data=samples, columns=ppe_params.columns)
m, _ = model.predict(new_samples.values)
```

(continues on next page)

(continued from previous page)

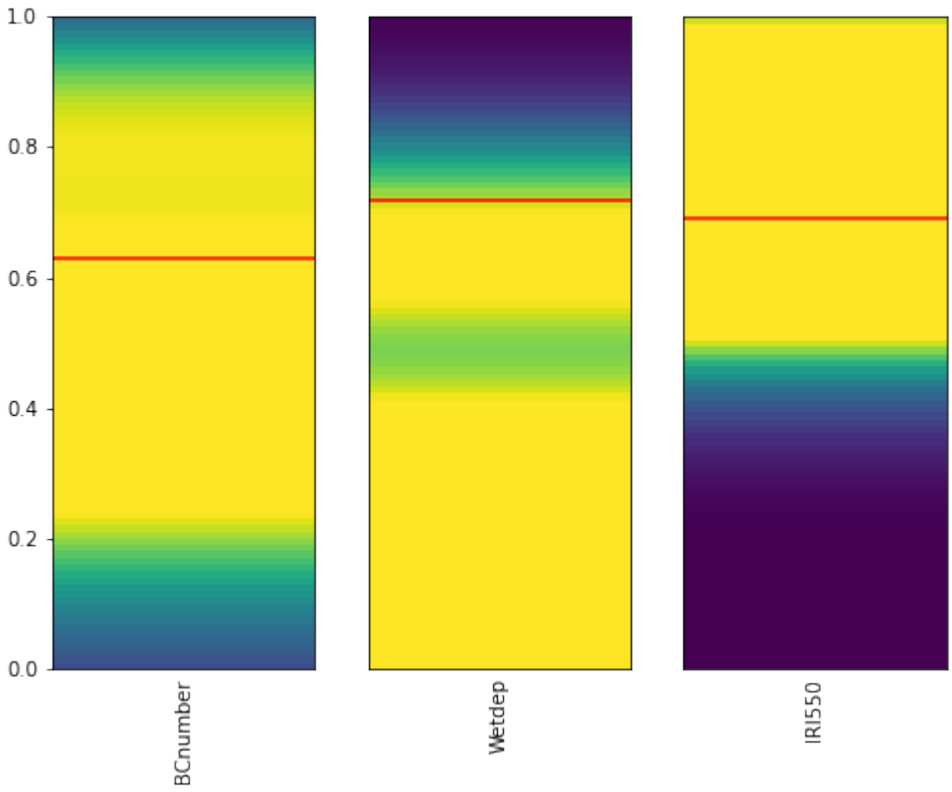
```
Zs = m.data
```

[16]:
```
print("Sample mean: {}".format(Zs.mean()))
print("Sample std dev: {}".format(Zs.std()))
```

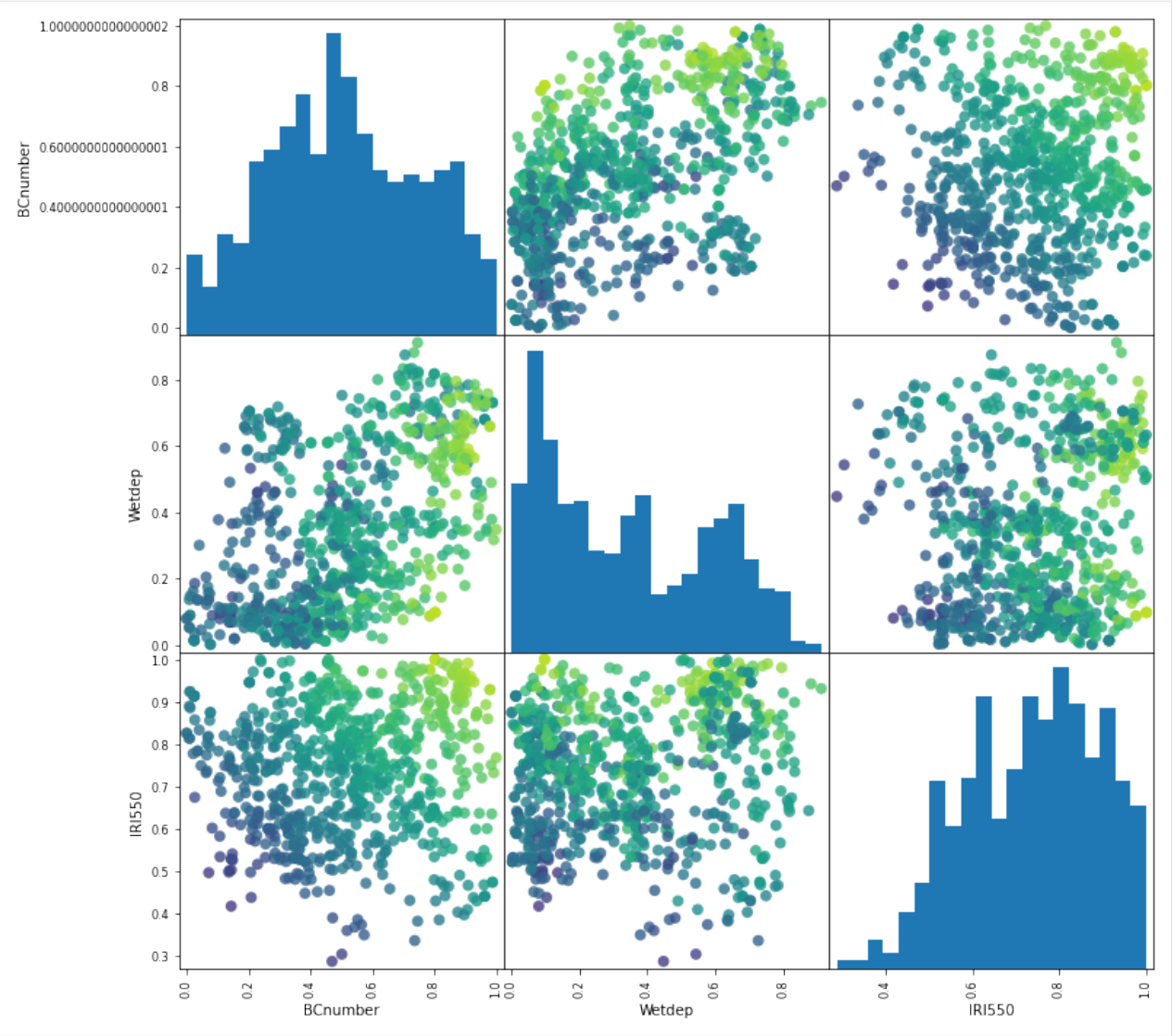
```
Sample mean: 0.003150297212253644
Sample std dev: 0.0006305448129278108
```

[17]:
```
plot_parameter_space(new_samples, target_df=X_test.iloc[0])
```

```
c:\users\duncan\pycharmprojects\gcem\esem\utils.py:119: MatplotlibDeprecationWarning:␣
↪shading='flat' when X and Y have the same dimensions as C is deprecated since 3.3. ␣
↪Either specify the corners of the quadrilaterals with X and Y, or pass shading='auto',
↪'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will become an error␣
↪two minor releases later.
  ax.pcolor(X, Y, vals[:, np.newaxis], vmin=0, vmax=1)
```



[18]:
```
grr = pd.plotting.scatter_matrix(new_samples, c=Zs, figsize=(12, 12), marker='o',
                                 hist_kwds={'bins': 20}, s=60, alpha=.8, vmin=1e-3,␣
↪vmax=5e-3)
```



[19]:
```
from esem.abc_sampler import ABCSampler, constrain
from esem.utils import get_random_params
```

[20]:
```
sampler = ABCSampler(model, Y_test[0])

samples = sampler.sample(n_samples=2000, threshold=0.5)
valid_points = pd.DataFrame(data=samples, columns=ppe_params.columns)
```

```
Acceptance rate: 0.33641715727502103
```

[21]:
```
m, _ = model.predict(valid_points.values)
Zs = m.data
```

[22]:
```
print("Sample mean: {}".format(Zs.mean()))
print("Sample std dev: {}".format(Zs.std()))
```

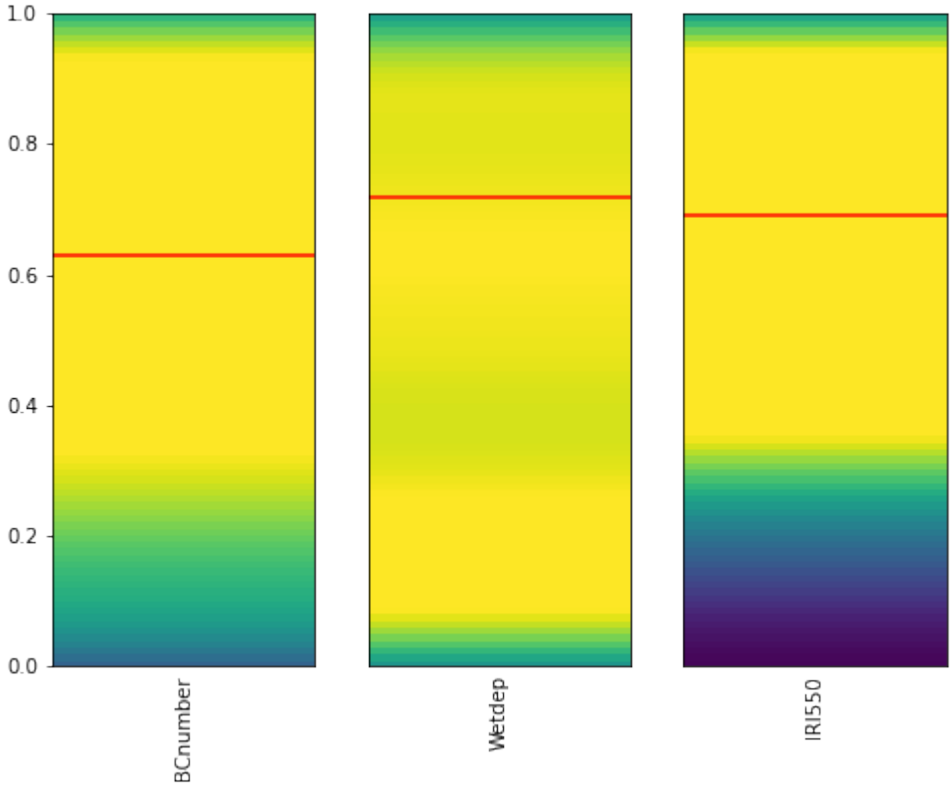
```
Sample mean: 0.002788395703569595
```

(continues on next page)

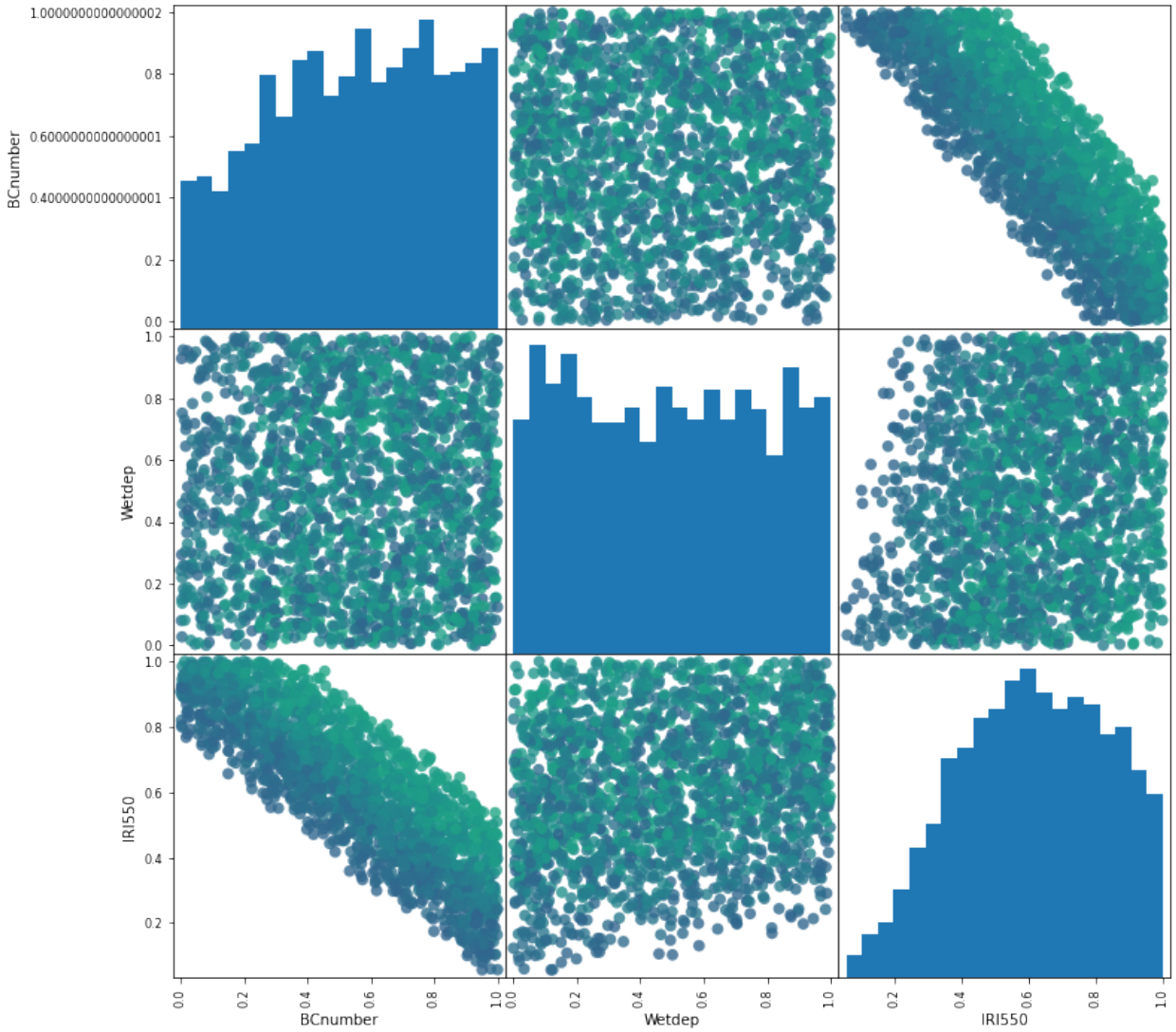
(continued from previous page)

```
Sample std dev: 0.00030132214587246 32
```

[23]:
```
plot_parameter_space(valid_points, target_df=X_test.iloc[0])
```



[24]:
```
grr = pd.plotting.scatter_matrix(valid_points, c=Zs, figsize=(12, 12), marker='o',
                                 hist_kwds={'bins': 20}, s=60, alpha=.8, vmin=1e-3,␣
↪vmax=5e-3)
```



[ ]:

5.6 CMIP6 Emulation

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from esem import gp_model
from esem.utils import validation_plot
%matplotlib inline
```

[2]:
```
df = pd.read_csv('CMIP6_scenarios.csv', index_col=0).dropna()
```

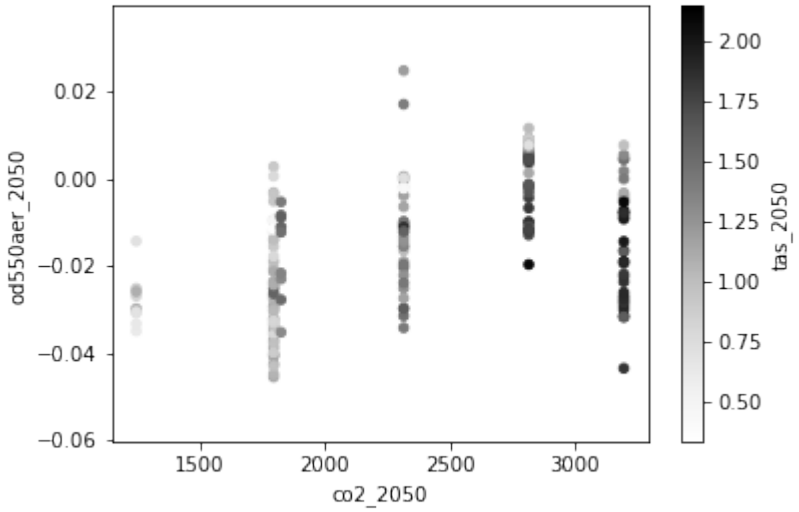
[3]:
```
# These are the models included
df.model.unique()
```

[3]:
```
array(['CanESM5', 'ACCESS-ESM1-5', 'ACCESS-CM2', 'MPI-ESM1-2-HR',
       'MIROC-ES2L', 'HadGEM3-GC31-LL', 'UKESM1-0-LL', 'MPI-ESM1-2-LR',
       'CESM2', 'CESM2-WACCM', 'NorESM2-LM'], dtype=object)
```

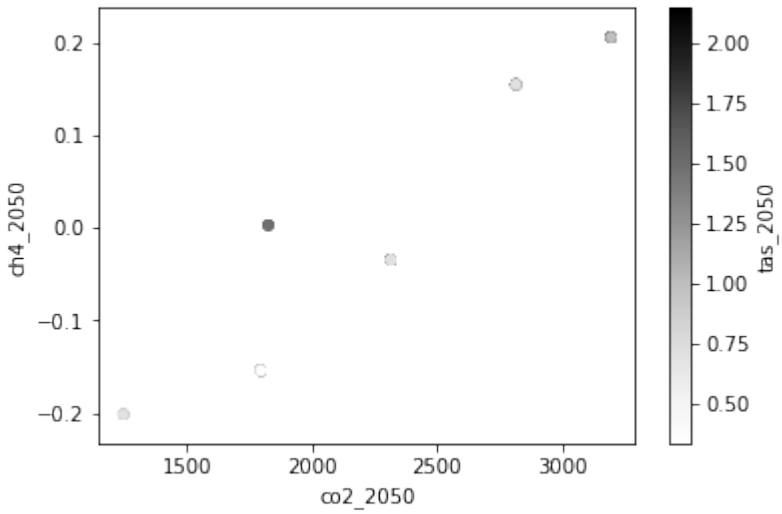
[4]:
```
# And these scenarios
df.scenario.unique()
```

[4]:
```
array(['ssp126', 'ssp119', 'ssp245', 'ssp370', 'ssp585', 'ssp434'],
      dtype=object)
```

[5]:
```
ax = df.plot.scatter(x='co2_2050', y='od550aer_2050', c='tas_2050')
```



[6]:
```
ax = df.plot.scatter(x='co2_2050', y='ch4_2050', c='tas_2050')
```



[7]:
```
# Collapse ensemble members
df = df.groupby(['model', 'scenario']).mean()
df
```

[7]:

model	scenario	tas_2050	od550aer_2050	tas_2100	od550aer_2100
ACCESS-CM2	ssp126	1.000582	-0.025191	1.407198	-0.038552
	ssp245	1.187015	-0.012222	2.489487	-0.027589
	ssp370	1.103402	0.006077	3.791109	0.001188
	ssp585	1.478602	-0.008842	5.016310	-0.016805
ACCESS-ESM1-5	ssp126	0.819904	-0.013947	0.967981	-0.015895
	ssp245	1.078853	-0.004727	2.052867	-0.009796
	ssp370	1.047218	0.003862	3.422980	0.000495
	ssp585	1.412020	-0.002210	4.096812	-0.001356
CESM2	ssp126	0.974787	-0.003323	1.134402	-0.010735
	ssp245	1.119462	0.000371	2.303586	0.000121
	ssp370	1.154434	0.007554	3.501974	0.016237
	ssp585	1.453782	0.002178	4.980513	0.016340
CESM2-WACCM	ssp126	0.770853	0.000561	1.053462	-0.006278
	ssp245	1.118329	0.000520	2.252389	0.006330
	ssp370	1.022903	0.011563	3.526784	0.035894
	ssp585	1.311378	0.005453	5.010599	0.038768
CanESM5	ssp119	0.641977	-0.030857	0.333723	-0.036624
	ssp126	0.989167	-0.029769	0.953098	-0.041451
	ssp245	1.318495	-0.016138	2.454336	-0.033573
	ssp370	1.724648	-0.004288	4.744037	-0.021374
	ssp434	1.304015	-0.023741	1.863815	-0.040477
	ssp585	1.877571	-0.025367	5.984793	-0.040852
HadGEM3-GC31-LL	ssp126	0.768553	-0.026508	1.314000	-0.037692
	ssp245	1.262740	-0.014397	2.651430	-0.024341
	ssp585	1.606117	-0.007902	5.547793	-0.015396
MIROC-ES2L	ssp119	0.703733	-0.014334	0.557255	-0.020002
	ssp126	0.785037	-0.017767	0.572429	-0.028761
	ssp245	0.789146	-0.010066	1.589075	-0.019236
	ssp370	1.131194	0.001314	2.623115	0.006749
	ssp585	1.117971	-0.003265	3.748485	-0.003236
MPI-ESM1-2-HR	ssp126	0.370221	-0.009666	0.351424	-0.014088
	ssp245	0.735819	-0.001871	1.352010	-0.008150
	ssp370	0.819827	0.008950	2.616101	0.003430
	ssp585	0.946618	0.004603	3.153594	-0.004407
MPI-ESM1-2-LR	ssp126	0.518146	-0.009654	0.358184	-0.014067
	ssp370	0.877299	0.008940	2.610222	0.003437
	ssp585	1.029724	0.004613	3.278956	-0.004377
NorESM2-LM	ssp126	0.331449	-0.011588	0.376486	-0.017219
	ssp245	0.533042	-0.001380	1.295670	-0.005419
	ssp370	0.704050	0.007691	2.552758	0.022834
	ssp585	0.983765	0.007667	2.957163	0.009652
UKESM1-0-LL	ssp119	0.918189	-0.027682	0.970074	-0.035182
	ssp126	1.284645	-0.024166	1.563342	-0.036149
	ssp245	1.604553	-0.010934	3.040434	-0.022142
	ssp370	1.749074	0.004618	5.046036	0.007180
	ssp434	1.511831	-0.009184	2.371976	-0.027848
	ssp585	2.028151	-0.007564	6.038747	-0.006429

(continues on next page)

(continued from previous page)

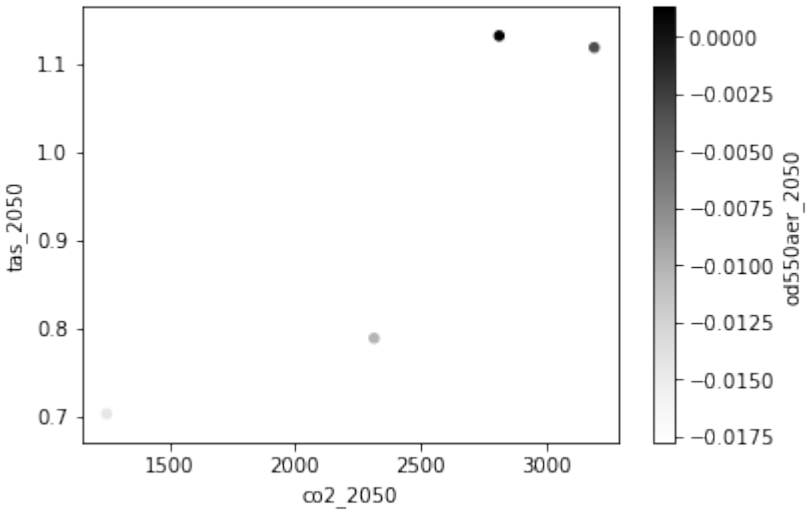
```
                          co2_2050      co2_2100  so2_2050  so2_2100  \
model           scenario
ACCESS-CM2      ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp245    2314.385253   3932.717046 -0.035997 -0.059775
                ssp370    2813.146604   6912.965613  0.004142 -0.020179
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536
ACCESS-ESM1-5   ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp245    2314.385253   3932.717046 -0.035997 -0.059775
                ssp370    2813.146604   6912.965613  0.004142 -0.020179
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536
CESM2           ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp245    2314.385253   3932.717046 -0.035997 -0.059775
                ssp370    2813.146604   6912.965613  0.004142 -0.020179
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536
CESM2-WACCM     ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp245    2314.385253   3932.717046 -0.035997 -0.059775
                ssp370    2813.146604   6912.965613  0.004142 -0.020179
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536
CanESM5         ssp119    1247.788346    905.867767 -0.069425 -0.080790
                ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp245    2314.385253   3932.717046 -0.035997 -0.059775
                ssp370    2813.146604   6912.965613  0.004142 -0.020179
                ssp434    1825.355018   1612.733274 -0.055372 -0.077382
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536
HadGEM3-GC31-LL ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp245    2314.385253   3932.717046 -0.035997 -0.059775
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536
MIROC-ES2L      ssp119    1247.788346    905.867767 -0.069425 -0.080790
                ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp245    2314.385253   3932.717046 -0.035997 -0.059775
                ssp370    2813.146604   6912.965613  0.004142 -0.020179
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536
MPI-ESM1-2-HR   ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp245    2314.385253   3932.717046 -0.035997 -0.059775
                ssp370    2813.146604   6912.965613  0.004142 -0.020179
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536
MPI-ESM1-2-LR   ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp370    2813.146604   6912.965613  0.004142 -0.020179
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536
NorESM2-LM      ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp245    2314.385253   3932.717046 -0.035997 -0.059775
                ssp370    2813.146604   6912.965613  0.004142 -0.020179
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536
UKESM1-0-LL     ssp119    1247.788346    905.867767 -0.069425 -0.080790
                ssp126    1795.710867   1848.864201 -0.064020 -0.082066
                ssp245    2314.385253   3932.717046 -0.035997 -0.059775
                ssp370    2813.146604   6912.965613  0.004142 -0.020179
                ssp434    1825.355018   1612.733274 -0.055372 -0.077382
                ssp585    3192.373467  10283.292188 -0.028185 -0.059536

                          ch4_2050  ch4_2100
```

(continues on next page)

(continued from previous page)

```
model           scenario
ACCESS-CM2      ssp126   -0.153999 -0.241390
                ssp245   -0.034673 -0.092845
                ssp370    0.154481  0.368859
                ssp585    0.205365  0.104140
ACCESS-ESM1-5   ssp126   -0.153999 -0.241390
                ssp245   -0.034673 -0.092845
                ssp370    0.154481  0.368859
                ssp585    0.205365  0.104140
CESM2           ssp126   -0.153999 -0.241390
                ssp245   -0.034673 -0.092845
                ssp370    0.154481  0.368859
                ssp585    0.205365  0.104140
CESM2-WACCM     ssp126   -0.153999 -0.241390
                ssp245   -0.034673 -0.092845
                ssp370    0.154481  0.368859
                ssp585    0.205365  0.104140
CanESM5         ssp119   -0.201275 -0.257405
                ssp126   -0.153999 -0.241390
                ssp245   -0.034673 -0.092845
                ssp370    0.154481  0.368859
                ssp434    0.002907 -0.032744
                ssp585    0.205365  0.104140
HadGEM3-GC31-LL ssp126   -0.153999 -0.241390
                ssp245   -0.034673 -0.092845
                ssp585    0.205365  0.104140
MIROC-ES2L      ssp119   -0.201275 -0.257405
                ssp126   -0.153999 -0.241390
                ssp245   -0.034673 -0.092845
                ssp370    0.154481  0.368859
                ssp585    0.205365  0.104140
MPI-ESM1-2-HR   ssp126   -0.153999 -0.241390
                ssp245   -0.034673 -0.092845
                ssp370    0.154481  0.368859
                ssp585    0.205365  0.104140
MPI-ESM1-2-LR   ssp126   -0.153999 -0.241390
                ssp370    0.154481  0.368859
                ssp585    0.205365  0.104140
NorESM2-LM      ssp126   -0.153999 -0.241390
                ssp245   -0.034673 -0.092845
                ssp370    0.154481  0.368859
                ssp585    0.205365  0.104140
UKESM1-0-LL     ssp119   -0.201275 -0.257405
                ssp126   -0.153999 -0.241390
                ssp245   -0.034673 -0.092845
                ssp370    0.154481  0.368859
                ssp434    0.002907 -0.032744
                ssp585    0.205365  0.104140
```

[8]:
```
ax = df.query("model == 'MIROC-ES2L'").plot.scatter(x='co2_2050', y='tas_2050', c=
↪'od550aer_2050')
```



[9]:
```
from utils import normalize
# Merge the year columns in to a long df
df=pd.wide_to_long(df.reset_index(), ["tas", "od550aer", "co2", "ch4", "so2"], i=['model
↪', 'scenario'], j="year", suffix='_(\d+)')
# Choose only the 2050 data since the aerosol signal is pretty non-existent by 2100
df = df[df.index.isin(["_2050"], level=2)]
```

[10]:
```
df.describe()
```

[10]:

	tas	od550aer	co2	ch4	so2
count	47.000000	47.000000	47.000000	47.000000	47.000000
mean	1.085538	-0.006851	2415.708964	0.024789	-0.035145
std	0.381735	0.011775	613.880074	0.152382	0.025320
min	0.331449	-0.030857	1247.788346	-0.201275	-0.069425
25%	0.804487	-0.014140	1795.710867	-0.153999	-0.064020
50%	1.047218	-0.004727	2314.385253	-0.034673	-0.035997
75%	1.307697	0.003020	2813.146604	0.154481	-0.028185
max	2.028151	0.011563	3192.373467	0.205365	0.004142

[11]:
```
# Do a 20/80 split of the data for test and training
msk = np.random.rand(len(df)) < 0.8
train, test = df[msk], df[~msk]
```

5.6.1 Try a few different models

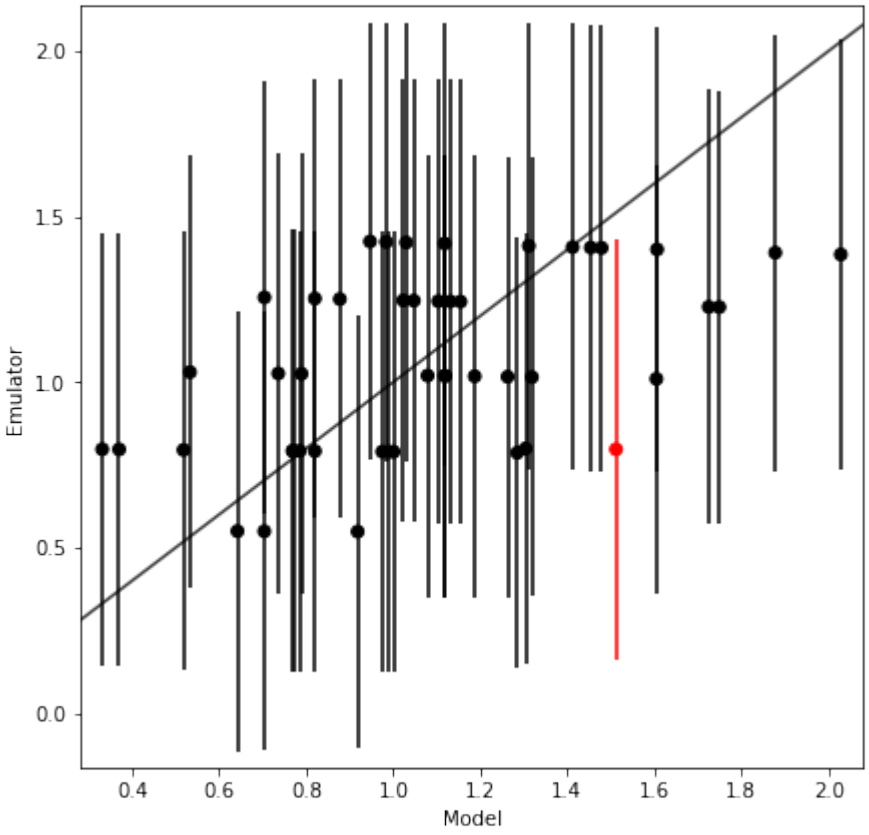
[12]:
```
from esem.utils import leave_one_out, prediction_within_ci
from scipy import stats

# Try just modelling the temperature based on cumulative CO2
res = leave_one_out(df[['co2']], df[['tas']].values, model='GaussianProcess', kernel=[
↪'Linear'])

r2_values = stats.linregress(*np.squeeze(np.asarray(res, dtype=float)).T[0:2])[2]**2
```

(continues on next page)

(continued from previous page)

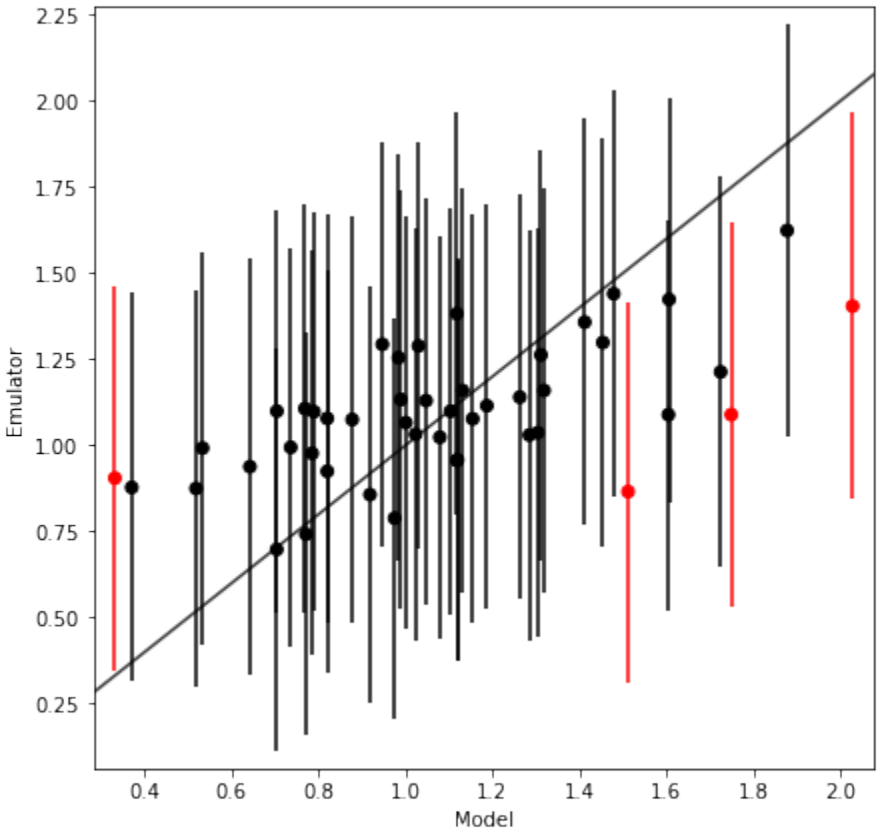
```
print("R^2: {:.2f}".format(r2_values))
validation_plot(*np.squeeze(np.asarray(res, dtype=float)).T)
```

```
R^2: 0.25
Proportion of 'Bad' estimates : 2.13%
```



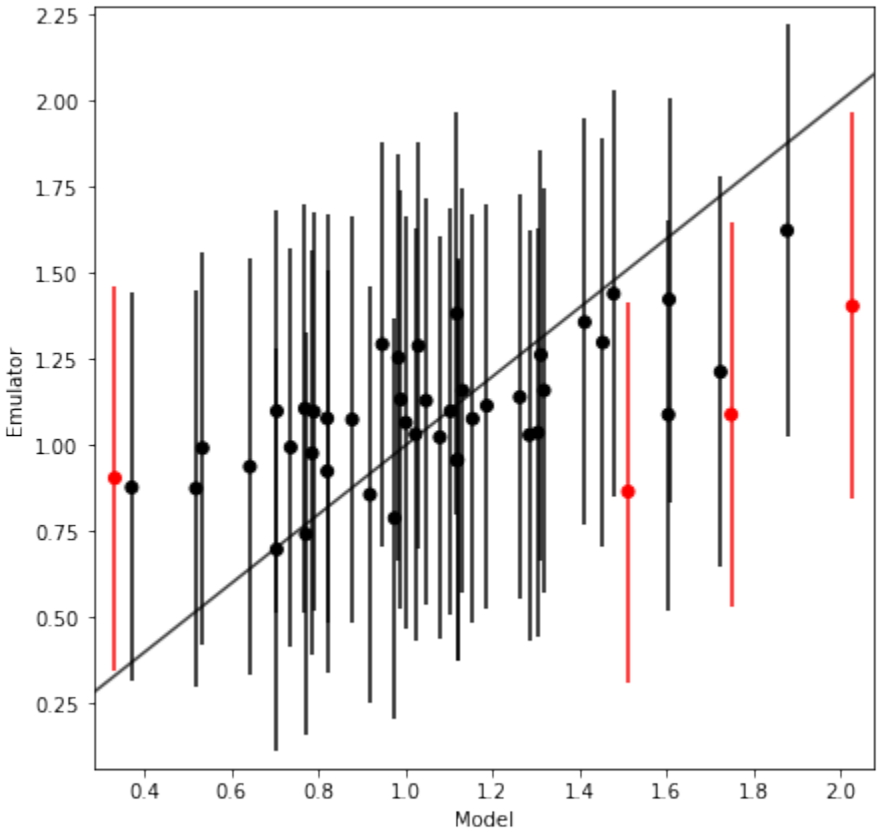
[13]:
```
# This model still doesn't do brilliantly, but it's better than just CO2
res = leave_one_out(df[['co2', 'od550aer']], df[['tas']].values, model='GaussianProcess',
↪ kernel=['Linear'])

r2_values = stats.linregress(*np.squeeze(np.asarray(res, dtype=float)).T[0:2])[2]**2
print("R^2: {:.2f}".format(r2_values))
validation_plot(*np.squeeze(np.asarray(res, dtype=float)).T)
```

```
R^2: 0.40
Proportion of 'Bad' estimates : 8.51%
```



[14]:
```
# Adding Methane doesn't seem to improve the picture
res = leave_one_out(df[['co2', 'od550aer', 'ch4']], df[['tas']].values, model=
↪'GaussianProcess', kernel=['Linear', 'Bias'])

r2_values = stats.linregress(*np.squeeze(np.asarray(res, dtype=float)).T[0:2])[2]**2
print("R^2: {:.2f}".format(r2_values))
validation_plot(*np.squeeze(np.asarray(res, dtype=float)).T)
```

```
R^2: 0.40
Proportion of 'Bad' estimates : 8.51%
```



5.6.2 Plot the best

[15]:
```
m = gp_model(df[['co2', 'od550aer']], df[['tas']].values, kernel=['Linear'])
m.train()
```

[16]:
```
# Sample a large AOD/CO2 space using the emulator
xx, yy = np.meshgrid(np.linspace(0, 4000, 25), np.linspace(-.05, 0.05, 20))
X_new = np.stack([xx.flat, yy.flat], axis=1)
Y_new, Y_new_sigma = m.predict(X_new)
```

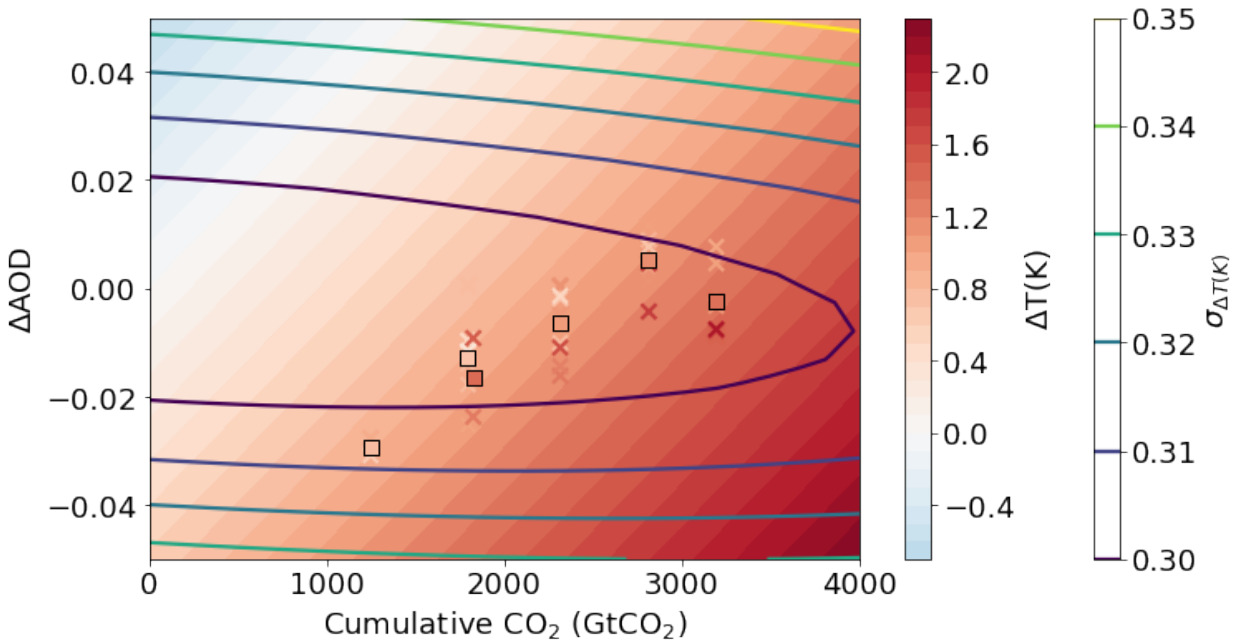
[17]:
```
# Calculate the scnario mean values for comparison
scn_mean = train.groupby(['scenario']).mean()
```

[18]:
```
import matplotlib

scale = 1.5
matplotlib.rcParams['font.size'] = 12 * scale
matplotlib.rcParams['lines.linewidth'] = 1.5 * scale
matplotlib.rcParams['lines.markersize'] = 6 * scale

plt.figure(figsize=(12, 6))
```

(continues on next page)

(continued from previous page)

```
norm = matplotlib.colors.Normalize(vmin=-2.5,vmax=2.5)
p = plt.contourf(xx, yy, Y_new.reshape(xx.shape), norm=norm, levels=30, cmap='RdBu_r')

plt.scatter(train.co2, train.od550aer, c=train.tas, norm=norm, edgecolors='k', cmap=
↪'RdBu_r', marker='x')
plt.scatter(scn_mean.co2, scn_mean.od550aer, c=scn_mean.tas, norm=norm, edgecolors='k',␣
↪cmap='RdBu_r', marker='s')

c = plt.contour(xx, yy, np.sqrt(Y_new_sigma.reshape(xx.shape)), cmap='viridis', levels=6)

plt.setp(plt.gca(), xlabel='Cumulative CO$_2$ (GtCO$_2$)', ylabel='$\Delta$AOD')

plt.colorbar(c, label='$\sigma_{\Delta T(K)}$')
plt.colorbar(p, label='$\Delta$T(K)')

# Cumulative CO2, delta T and delta AOD all relative to a 2015-2020 average. Each point␣
↪represents a single model integration for different scenarios in the CMIP6 archive.
plt.savefig('CMIP6_emulator_paper_v1.1.png', transparent=True)
```



5.6.3 Sample emissions for a particular temperature target

[19]:
```
from esem.sampler import MCMCSampler

# The MCMC algorithm works much better with a normalised parameter range, so recreate␣
↪the model
m = gp_model(pd.concat([df[['co2']]/4000, (df[['od550aer']]+0.05)/0.1], axis=1), df[['tas
↪']].values, kernel=['Linear'])
m.train()
```

(continues on next page)

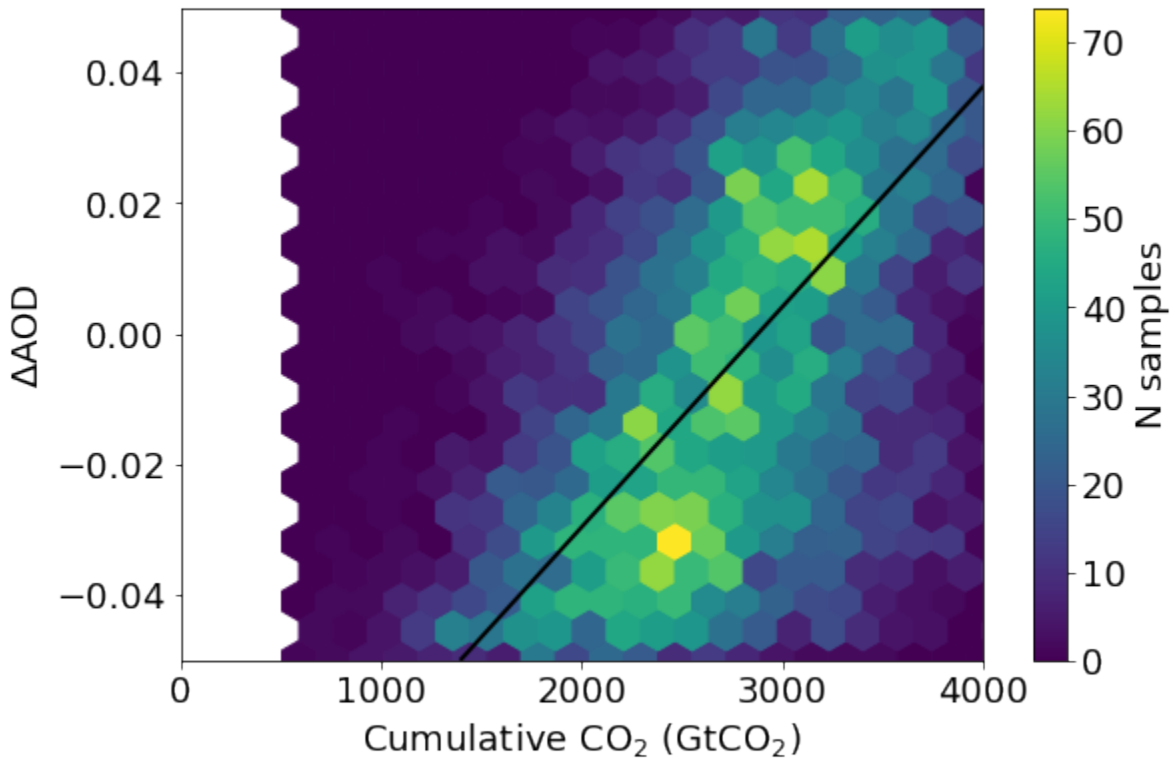
(continued from previous page)

```
# Target 1.2 degrees above present day (roughly 2 degrees above pre-industrial)
sampler = MCMCSampler(m, np.asarray([1.2], dtype=np.float64))
samples = sampler.sample(n_samples=8000, mcmc_kwargs=dict(num_burnin_steps=1000) )
```

```
Acceptance rate: 0.9614173964786951
```

[20]:
```
# Get the emulated temperatures for these samples
new_samples = pd.DataFrame(data=samples, columns=['co2', 'od550aer'])
Z, _ = m.predict(new_samples.values)
```

[21]:
```
fig = plt.figure(figsize=(9, 6))

cl = plt.contour(xx, yy, Y_new.reshape(xx.shape), levels = [1.2],
             colors=('k',),linestyles=('-',),linewidths=(2,))

cl=plt.hexbin(new_samples.co2*4000, new_samples.od550aer*0.1-0.05, gridsize=20)

plt.setp(plt.gca(), xlabel='Cumulative CO$_2$ (GtCO$_2$)', ylabel='$\Delta$AOD')

plt.colorbar(cl, label='N samples')
plt.setp(plt.gca(), ylim=[-0.05, 0.05], xlim=[0, 4000])

plt.savefig('CMIP6_emulator_sampled.png', transparent=True)
```



[ ]:

5.7 Create paper emulation figure

[1]:
```
import os
## Ignore my broken HDF5 install...
os.putenv("HDF5_DISABLE_VERSION_CHECK", '1')
```

[2]:
```
import iris

from utils import get_bc_ppe_data

from esem import cnn_model, gp_model
from esem.utils import get_random_params

import iris.quickplot as qplt
import iris.analysis.maths as imath
import matplotlib.pyplot as plt
%matplotlib inline
```

5.7.1 Read in the parameters and data

[3]:
```
ppe_params, ppe_aaod = get_bc_ppe_data()
```

```
/Users/watson-parris/miniconda3/envs/gcem/lib/python3.8/site-packages/iris/__init__.py:
↪249: IrisDeprecation: setting the 'Future' property 'netcdf_promote' is deprecated and␣
↪will be removed in a future release. Please remove code that sets this property.
  warn_deprecated(msg.format(name))
/Users/watson-parris/miniconda3/envs/gcem/lib/python3.8/site-packages/iris/__init__.py:
↪249: IrisDeprecation: setting the 'Future' property 'netcdf_promote' is deprecated and␣
↪will be removed in a future release. Please remove code that sets this property.
  warn_deprecated(msg.format(name))
```

[4]:
```
## Ensure thWetdepnumbertime dimension is last - this is treated as the color 'channel'
## ppe_aaod.transpose((0,2,3,1))
ppe_aaod = ppe_aaod.collapsed('time')[0]
```

```
WARNING:root:Creating guessed bounds as none exist in file
WARNING:root:Creating guessed bounds as none exist in file
WARNING:root:Creating guessed bounds as none exist in file
```

[5]:
```
n_test = 5

X_test, X_train = ppe_params[:n_test], ppe_params[n_test:]
Y_test, Y_train = ppe_aaod[:n_test], ppe_aaod[n_test:]
```

[6]:
```
Y_train
```

[6]:
```
<iris 'Cube' of Absorption optical thickness - total 550nm / (1) (job: 34; latitude: 96;␣
↪longitude: 192)>
```

5.7.2 Setup and run the models

[7]:
```
nn_model = cnn_model(X_train, Y_train)
```

[8]:
```
nn_model.model.model.summary()
```

```
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 3)]               0
_________________________________________________________________
dense (Dense)                (None, 221184)            884736
_________________________________________________________________
reshape (Reshape)            (None, 96, 192, 12)       0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 96, 192, 12)       2172
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 96, 192, 1)        181
=================================================================
Total params: 887,089
Trainable params: 887,089
Non-trainable params: 0
_________________________________________________________________
```

[9]:
```
nn_model.train()
```

```
Epoch 1/100
4/4 [==============================] - 1s 284ms/step - loss: 0.8766 - val_loss: 0.4800
Epoch 2/100
4/4 [==============================] - 0s 115ms/step - loss: 1.0869 - val_loss: 0.4778
Epoch 3/100
4/4 [==============================] - 0s 100ms/step - loss: 1.1900 - val_loss: 0.4759
Epoch 4/100
4/4 [==============================] - 0s 106ms/step - loss: 1.0656 - val_loss: 0.4741
Epoch 5/100
4/4 [==============================] - 0s 99ms/step - loss: 1.2058 - val_loss: 0.4699
Epoch 6/100
4/4 [==============================] - 0s 112ms/step - loss: 1.2211 - val_loss: 0.4672
Epoch 7/100
4/4 [==============================] - 0s 105ms/step - loss: 1.0138 - val_loss: 0.4624
Epoch 8/100
4/4 [==============================] - 0s 101ms/step - loss: 0.9261 - val_loss: 0.4645
Epoch 9/100
4/4 [==============================] - 0s 112ms/step - loss: 0.9354 - val_loss: 0.4569
Epoch 10/100
4/4 [==============================] - 0s 100ms/step - loss: 1.1767 - val_loss: 0.4494
Epoch 11/100
4/4 [==============================] - 0s 101ms/step - loss: 1.1077 - val_loss: 0.4310
Epoch 12/100
4/4 [==============================] - 0s 101ms/step - loss: 0.8575 - val_loss: 0.4343
Epoch 13/100
4/4 [==============================] - 0s 101ms/step - loss: 0.8162 - val_loss: 0.4476
```

(continues on next page)

(continued from previous page)

```
Epoch 14/100
4/4 [==============================] - 0s 102ms/step - loss: 0.8983 - val_loss: 0.4181
Epoch 15/100
4/4 [==============================] - 1s 104ms/step - loss: 0.8968 - val_loss: 0.4003
Epoch 16/100
4/4 [==============================] - 0s 100ms/step - loss: 0.8450 - val_loss: 0.3989
Epoch 17/100
4/4 [==============================] - 0s 103ms/step - loss: 0.7290 - val_loss: 0.4107
Epoch 18/100
4/4 [==============================] - 0s 100ms/step - loss: 0.8785 - val_loss: 0.3785
Epoch 19/100
4/4 [==============================] - 0s 106ms/step - loss: 0.7236 - val_loss: 0.3918
Epoch 20/100
4/4 [==============================] - 0s 103ms/step - loss: 0.8572 - val_loss: 0.3857
Epoch 21/100
4/4 [==============================] - 0s 102ms/step - loss: 0.8973 - val_loss: 0.3543
Epoch 22/100
4/4 [==============================] - 0s 101ms/step - loss: 0.9261 - val_loss: 0.3282
Epoch 23/100
4/4 [==============================] - 0s 102ms/step - loss: 0.5686 - val_loss: 0.3123
Epoch 24/100
4/4 [==============================] - 0s 100ms/step - loss: 0.9107 - val_loss: 0.3041
Epoch 25/100
4/4 [==============================] - 0s 101ms/step - loss: 0.7340 - val_loss: 0.2931
Epoch 26/100
4/4 [==============================] - 0s 100ms/step - loss: 0.7675 - val_loss: 0.2825
Epoch 27/100
4/4 [==============================] - 0s 105ms/step - loss: 0.5598 - val_loss: 0.2795
Epoch 28/100
4/4 [==============================] - 0s 101ms/step - loss: 0.6441 - val_loss: 0.2640
Epoch 29/100
4/4 [==============================] - 0s 100ms/step - loss: 0.8158 - val_loss: 0.2579
Epoch 30/100
4/4 [==============================] - 0s 102ms/step - loss: 0.7065 - val_loss: 0.2447
Epoch 31/100
4/4 [==============================] - 0s 105ms/step - loss: 0.6948 - val_loss: 0.2351
Epoch 32/100
4/4 [==============================] - 0s 125ms/step - loss: 0.7187 - val_loss: 0.2226
Epoch 33/100
4/4 [==============================] - 0s 122ms/step - loss: 0.5605 - val_loss: 0.2154
Epoch 34/100
4/4 [==============================] - 0s 104ms/step - loss: 0.5716 - val_loss: 0.2117
Epoch 35/100
4/4 [==============================] - 0s 107ms/step - loss: 0.6035 - val_loss: 0.1991
Epoch 36/100
4/4 [==============================] - 0s 107ms/step - loss: 0.6084 - val_loss: 0.1892
Epoch 37/100
4/4 [==============================] - 0s 103ms/step - loss: 0.6416 - val_loss: 0.1786
Epoch 38/100
4/4 [==============================] - 0s 100ms/step - loss: 0.4608 - val_loss: 0.1749
Epoch 39/100
4/4 [==============================] - 0s 100ms/step - loss: 0.3582 - val_loss: 0.1674
```

(continues on next page)

(continued from previous page)

```
Epoch 40/100
4/4 [==============================] - 0s 113ms/step - loss: 0.4616 - val_loss: 0.1607
Epoch 41/100
4/4 [==============================] - 0s 117ms/step - loss: 0.5972 - val_loss: 0.1558
Epoch 42/100
4/4 [==============================] - 0s 121ms/step - loss: 0.3961 - val_loss: 0.1471
Epoch 43/100
4/4 [==============================] - 0s 111ms/step - loss: 0.5399 - val_loss: 0.1404
Epoch 44/100
4/4 [==============================] - 0s 100ms/step - loss: 0.4118 - val_loss: 0.1385
Epoch 45/100
4/4 [==============================] - 0s 126ms/step - loss: 0.3920 - val_loss: 0.1316
Epoch 46/100
4/4 [==============================] - 0s 121ms/step - loss: 0.4126 - val_loss: 0.1237
Epoch 47/100
4/4 [==============================] - 1s 247ms/step - loss: 0.3554 - val_loss: 0.1202
Epoch 48/100
4/4 [==============================] - 0s 129ms/step - loss: 0.3429 - val_loss: 0.1145
Epoch 49/100
4/4 [==============================] - 1s 128ms/step - loss: 0.3157 - val_loss: 0.1075
Epoch 50/100
4/4 [==============================] - 1s 117ms/step - loss: 0.4217 - val_loss: 0.1051
Epoch 51/100
4/4 [==============================] - 0s 112ms/step - loss: 0.4561 - val_loss: 0.1078
Epoch 52/100
4/4 [==============================] - 0s 121ms/step - loss: 0.4475 - val_loss: 0.0932
Epoch 53/100
4/4 [==============================] - 0s 123ms/step - loss: 0.2861 - val_loss: 0.0896
Epoch 54/100
4/4 [==============================] - 0s 115ms/step - loss: 0.3469 - val_loss: 0.0886
Epoch 55/100
4/4 [==============================] - 0s 126ms/step - loss: 0.3332 - val_loss: 0.0920
Epoch 56/100
4/4 [==============================] - 0s 117ms/step - loss: 0.3290 - val_loss: 0.0793
Epoch 57/100
4/4 [==============================] - 1s 136ms/step - loss: 0.3292 - val_loss: 0.0792
Epoch 58/100
4/4 [==============================] - 0s 104ms/step - loss: 0.2757 - val_loss: 0.0760
Epoch 59/100
4/4 [==============================] - 0s 110ms/step - loss: 0.2637 - val_loss: 0.0720
Epoch 60/100
4/4 [==============================] - 0s 123ms/step - loss: 0.4001 - val_loss: 0.0729
Epoch 61/100
4/4 [==============================] - 0s 110ms/step - loss: 0.4507 - val_loss: 0.0666
Epoch 62/100
4/4 [==============================] - 1s 131ms/step - loss: 0.3879 - val_loss: 0.0695
Epoch 63/100
4/4 [==============================] - 0s 106ms/step - loss: 0.2597 - val_loss: 0.0610
Epoch 64/100
4/4 [==============================] - 0s 111ms/step - loss: 0.3288 - val_loss: 0.0725
Epoch 65/100
4/4 [==============================] - 0s 102ms/step - loss: 0.2841 - val_loss: 0.0575
```

(continues on next page)

(continued from previous page)

```
Epoch 66/100
4/4 [==============================] - 0s 101ms/step - loss: 0.2730 - val_loss: 0.0544
Epoch 67/100
4/4 [==============================] - 0s 101ms/step - loss: 0.2694 - val_loss: 0.0517
Epoch 68/100
4/4 [==============================] - 0s 110ms/step - loss: 0.3846 - val_loss: 0.0573
Epoch 69/100
4/4 [==============================] - 0s 100ms/step - loss: 0.3429 - val_loss: 0.0528
Epoch 70/100
4/4 [==============================] - 0s 106ms/step - loss: 0.2360 - val_loss: 0.0478
Epoch 71/100
4/4 [==============================] - 0s 100ms/step - loss: 0.3651 - val_loss: 0.0486
Epoch 72/100
4/4 [==============================] - 0s 107ms/step - loss: 0.2351 - val_loss: 0.0469
Epoch 73/100
4/4 [==============================] - 0s 101ms/step - loss: 0.2993 - val_loss: 0.0476
Epoch 74/100
4/4 [==============================] - 0s 105ms/step - loss: 0.2739 - val_loss: 0.0523
Epoch 75/100
4/4 [==============================] - 0s 103ms/step - loss: 0.3909 - val_loss: 0.0438
Epoch 76/100
4/4 [==============================] - 0s 109ms/step - loss: 0.2526 - val_loss: 0.0423
Epoch 77/100
4/4 [==============================] - 0s 103ms/step - loss: 0.2566 - val_loss: 0.0485
Epoch 78/100
4/4 [==============================] - 0s 101ms/step - loss: 0.2969 - val_loss: 0.0411
Epoch 79/100
4/4 [==============================] - 0s 99ms/step - loss: 0.2602 - val_loss: 0.0413
Epoch 80/100
4/4 [==============================] - 0s 126ms/step - loss: 0.1863 - val_loss: 0.0382
Epoch 81/100
4/4 [==============================] - 0s 101ms/step - loss: 0.3251 - val_loss: 0.0394
Epoch 82/100
4/4 [==============================] - 0s 109ms/step - loss: 0.3038 - val_loss: 0.0376
Epoch 83/100
4/4 [==============================] - 0s 105ms/step - loss: 0.3729 - val_loss: 0.0405
Epoch 84/100
4/4 [==============================] - 0s 107ms/step - loss: 0.2542 - val_loss: 0.0389
Epoch 85/100
4/4 [==============================] - 0s 109ms/step - loss: 0.2379 - val_loss: 0.0403
Epoch 86/100
4/4 [==============================] - 0s 117ms/step - loss: 0.2620 - val_loss: 0.0350
Epoch 87/100
4/4 [==============================] - 0s 108ms/step - loss: 0.2459 - val_loss: 0.0356
Epoch 88/100
4/4 [==============================] - 0s 107ms/step - loss: 0.1902 - val_loss: 0.0368
Epoch 89/100
4/4 [==============================] - 1s 134ms/step - loss: 0.3153 - val_loss: 0.0362
Epoch 90/100
4/4 [==============================] - 0s 116ms/step - loss: 0.3270 - val_loss: 0.0371
Epoch 91/100
4/4 [==============================] - 1s 123ms/step - loss: 0.3058 - val_loss: 0.0393
```

(continues on next page)

(continued from previous page)

```
Epoch 92/100
4/4 [==============================] - 0s 115ms/step - loss: 0.3535 - val_loss: 0.0359
Epoch 93/100
4/4 [==============================] - 0s 102ms/step - loss: 0.3056 - val_loss: 0.0379
Epoch 94/100
4/4 [==============================] - 0s 106ms/step - loss: 0.3331 - val_loss: 0.0354
Epoch 95/100
4/4 [==============================] - 0s 99ms/step - loss: 0.2074 - val_loss: 0.0364
Epoch 96/100
4/4 [==============================] - 0s 99ms/step - loss: 0.1920 - val_loss: 0.0364
Epoch 97/100
4/4 [==============================] - 0s 101ms/step - loss: 0.3333 - val_loss: 0.0329
Epoch 98/100
4/4 [==============================] - 0s 99ms/step - loss: 0.2105 - val_loss: 0.0340
Epoch 99/100
4/4 [==============================] - 0s 105ms/step - loss: 0.3683 - val_loss: 0.0355
Epoch 100/100
4/4 [==============================] - 0s 100ms/step - loss: 0.3073 - val_loss: 0.0361
```

[10]:
```
## Linear model:  0.3566 - val_loss: 0.0867
```

[11]:
```
nn_prediction, _ = nn_model.predict(X_test.values)
```

[12]:
```
gp_model_ = gp_model(X_train, Y_train, kernel=['Bias', 'Linear'])
gp_model_.train()
```

[13]:
```
gp_prediction, _ = gp_model_.predict(X_test.values)
```

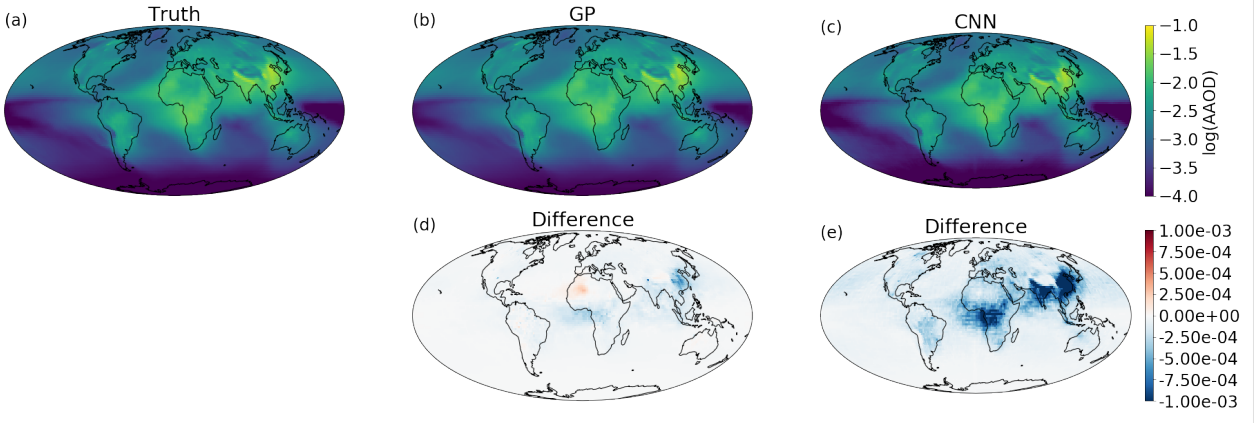
[14]:
```
import matplotlib
import cartopy.crs as ccrs
import iris.plot as iplt


plt.figure(figsize=(30, 10))
matplotlib.rcParams['font.size'] = 24

plt.subplot(2,3,1, projection=ccrs.Mollweide())
plt.annotate("(a)", (0.,1.), xycoords='axes fraction')
iplt.pcolormesh(imath.log10(Y_test[0]), vmin=-4, vmax=-1)
plt.gca().set_title('Truth')
plt.gca().coastlines()

plt.subplot(2,3,2, projection=ccrs.Mollweide())
plt.annotate("(b)", (0.,1.), xycoords='axes fraction')
iplt.pcolormesh(imath.log10(gp_prediction[0]), vmin=-4, vmax=-1)
plt.gca().set_title('GP')
plt.gca().coastlines()

plt.subplot(2,3,3, projection=ccrs.Mollweide())
plt.annotate("(c)", (0.,1.), xycoords='axes fraction')
im=iplt.pcolormesh(imath.log10(nn_prediction[0]), vmin=-4, vmax=-1)
```

(continues on next page)

(continued from previous page)

```
plt.gca().set_title('CNN')
plt.colorbar(im, fraction=0.046, pad=0.04, label='log(AAOD)')
plt.gca().coastlines()

plt.subplot(2,3,5, projection=ccrs.Mollweide())
plt.annotate("(d)", (0.,1.), xycoords='axes fraction')
iplt.pcolormesh((gp_prediction.collapsed(['sample'], iris.analysis.MEAN)-Y_test.
↪collapsed(['job'], iris.analysis.MEAN)), cmap='RdBu_r', vmin=-0.001, vmax=0.001)
plt.gca().coastlines()
plt.gca().set_title('Difference')

plt.subplot(2,3,6, projection=ccrs.Mollweide())
plt.annotate("(e)", (0.,1.), xycoords='axes fraction')
im=iplt.pcolormesh((nn_prediction.collapsed(['sample'], iris.analysis.MEAN)-Y_test.
↪collapsed(['job'], iris.analysis.MEAN)), cmap='RdBu_r', vmin=-1e-3, vmax=1e-3)
cb = plt.colorbar(im, fraction=0.046, pad=0.04)
cb.ax.set_yticklabels(["{:.2e}".format(i) for i in cb.get_ticks()]) ## set ticks of your␣
↪format
plt.gca().coastlines()
plt.gca().set_title('Difference')

plt.savefig('BCPPE_emulator_paper.png', transparent=True)
```

```
/Users/watson-parris/miniconda3/envs/gcem/lib/python3.8/site-packages/iris/coords.py:
↪1410: UserWarning: Collapsing a non-contiguous coordinate. Metadata may not be fully␣
↪descriptive for 'sample'.
  warnings.warn(msg.format(self.name()))
/Users/watson-parris/miniconda3/envs/gcem/lib/python3.8/site-packages/iris/coords.py:
↪1410: UserWarning: Collapsing a non-contiguous coordinate. Metadata may not be fully␣
↪descriptive for 'sample'.
  warnings.warn(msg.format(self.name()))
```



[15]:

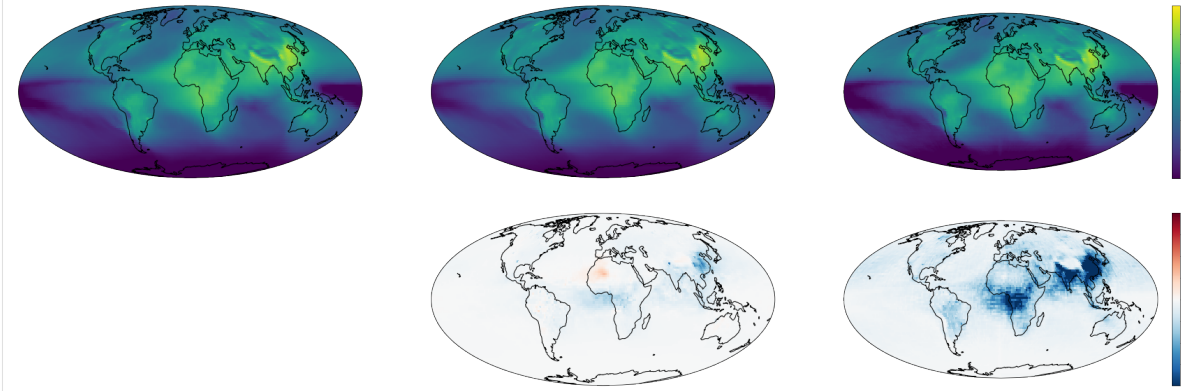
```
COLOR = 'white'
matplotlib.rcParams['text.color'] = COLOR
matplotlib.rcParams['axes.labelcolor'] = COLOR
matplotlib.rcParams['xtick.color'] = COLOR
matplotlib.rcParams['ytick.color'] = COLOR
matplotlib.rcParams['font.size'] = 20
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(30, 10))

plt.subplot(2,3,1, projection=ccrs.Mollweide())
plt.annotate("(a)", (0.,1.), xycoords='axes fraction')
iplt.pcolormesh(imath.log10(Y_test[0]), vmin=-4, vmax=-1)
plt.gca().set_title('Truth')
plt.gca().coastlines()

plt.subplot(2,3,2, projection=ccrs.Mollweide())
plt.annotate("(b)", (0.,1.), xycoords='axes fraction')
iplt.pcolormesh(imath.log10(gp_prediction[0]), vmin=-4, vmax=-1)
plt.gca().set_title('GP')
plt.gca().coastlines()

plt.subplot(2,3,3, projection=ccrs.Mollweide())
plt.annotate("(c)", (0.,1.), xycoords='axes fraction')
im=iplt.pcolormesh(imath.log10(nn_prediction[0]), vmin=-4, vmax=-1)
plt.gca().set_title('CNN')
plt.colorbar(im, fraction=0.046, pad=0.04, label='log(AAOD)')
plt.gca().coastlines()

plt.subplot(2,3,5, projection=ccrs.Mollweide())
plt.annotate("(d)", (0.,1.), xycoords='axes fraction')
iplt.pcolormesh((gp_prediction.collapsed(['sample'], iris.analysis.MEAN)-Y_test.
↪collapsed(['job'], iris.analysis.MEAN)), cmap='RdBu_r', vmin=-0.001, vmax=0.001)
plt.gca().coastlines()
plt.gca().set_title('Difference')

plt.subplot(2,3,6, projection=ccrs.Mollweide())
plt.annotate("(e)", (0.,1.), xycoords='axes fraction')
im=iplt.pcolormesh((nn_prediction.collapsed(['sample'], iris.analysis.MEAN)-Y_test.
↪collapsed(['job'], iris.analysis.MEAN)), cmap='RdBu_r', vmin=-1e-3, vmax=1e-3)
cb=plt.colorbar(im, fraction=0.046, pad=0.04)
cb.ax.set_yticklabels(["{:.2e}".format(i) for i in cb.get_ticks()]) ## set ticks of your␣
↪format
plt.gca().coastlines()
plt.gca().set_title('Difference')

plt.savefig('BCPPE_emulator_talk.png', transparent=True)
```

```
/Users/watson-parris/miniconda3/envs/gcem/lib/python3.8/site-packages/iris/coords.py:
↪1410: UserWarning: Collapsing a non-contiguous coordinate. Metadata may not be fully␣
↪descriptive for 'sample'.
  warnings.warn(msg.format(self.name()))
/Users/watson-parris/miniconda3/envs/gcem/lib/python3.8/site-packages/iris/coords.py:
↪1410: UserWarning: Collapsing a non-contiguous coordinate. Metadata may not be fully␣
↪descriptive for 'sample'.
  warnings.warn(msg.format(self.name()))
```



[ ]:

CHAPTER

SIX

API REFERENCE

This page provides an auto-generated summary of xarray's API. For more details and examples, refer to the relevant chapters in the main part of the documentation.

6.1 Top-level functions

This provides the main interface for ESEm and should be the starting point for most users.

gp_model	Create a Gaussian process (GP) based emulator with provided *training_params* (X) and *training_data* (Y) which assumes independent inputs (and outputs).
cnn_model	Create a simple two layer Convolutional Neural Network Emulator using Keras.
rf_model	Create a simple Random Forest Emulator using sklearn.

6.1.1 esem.gp_model

esem.**gp_model**(*training_params*, *training_data*, *data_processors=None*, *kernel=None*, *kernel_op='add'*, *active_dims=None*, *noise_variance=1.0*, *name=''*, *gpu=0*, ***kwargs*)

Create a Gaussian process (GP) based emulator with provided *training_params* (X) and *training_data* (Y) which assumes independent inputs (and outputs).

The *kernel* is a key parameter in GP emulation and care should be taken in choosing it.

Parameters

- **training_params** (`DataFrame`) – The training parameters
- **training_data** (`xarray.DataArray` or `iris.Cube` or *array_like*) – The training data - the leading dimension should represent training samples
- **data_processors** (`list` of *esem.data_processors.DataProcessor*) – A list of 'DataProcessor` to apply to the data transparently before training. Model output will be untransformed before being returned from the Emulator.
- **kernel** (`gpflow.kernels.Kernel` or `list` of `str` or `None`) – The GP kernel to use. A GPFlow kernel can be specified directly, or a list of kernel names can be provided which will be initialised using the default values (of the correct shape) and combined using *kernel_op*. Alternatively no kernel can be specified and a default will be used.
- **kernel_op** (`{'add', 'mul'}`) – The operation to perform in order to combine the specified *kernel`s. Only used if `kernel* is a list of strings.

- **noise_variance** (`float`) – The noise variance to initialise the GP regression model
- **active_dims** (`list` of `int` or `slice` or `None`) – The dimensions to train the GP over (by default all of the dimensions are used)
- **name** (`str`) – An optional name for the emulator
- **gpu** (`int`) – The GPU to use (only applicable for multi-GPU) machines
- **kwargs** (`dict`) – Dict of optional keyword arguments for *gpflow.models.GPR*, e.g., *mean_function*

Returns `Emulator` – An esem emulator object which can be trained and sampled from

6.1.2 esem.cnn_model

`esem.cnn_model`(*training_params*, *training_data*, *data_processors=None*, *filters=12*, *learning_rate=0.001*, *decay=0.01*, *kernel_size=(3, 5)*, *loss='mean_squared_error'*, *activation='tanh'*, *optimizer='RMSprop'*, *name=''*, *gpu=0*)

Create a simple two layer Convolutional Neural Network Emulator using Keras.

Note that X should include both the train and validation data

Parameters

- **training_params** (`pd.DataFrame`) – The training parameters
- **training_data** (`xarray.DataArray` or `iris.cube.Cube` or *array_like*) – The training data - the leading dimension should represent training samples
- **data_processors** (`list` of *esem.data_processors.DataProcessor*) – A list of *DataProcessor* to apply to the data transparently before training. Model output will be untransformed before being returned from the Emulator.
- **filters** (`int`) – The dimensionality of the first convolutional layer output space
- **learning_rate** (`float`) – The learning rate to use with the chosen optimizer
- **decay** (`float`) – Any decay to apply to the learning rate
- **kernel_size** (`tuple` of `int`) – The convolutional kernel size
- **loss** (`str`) – The loss function to train against (see https://keras.io/api/losses/)
- **activation** (`str`) – The activation function to use in the final CNN layer (see https://keras.io/api/layers/activations/)
- **optimizer** (`{'RMSprop', 'Adam'}`) – The optimizer to train the model with
- **name** (`str`) – An optional name for the emulator
- **gpu** (`int`) – The GPU to use (only applicable for multi-GPU) machines

Returns `Emulator` – An esem emulator object which can be trained and sampled from

Notes

The Keras model is compiled but not trained until *train* is called on the returned *Emulator* object.

6.1.3 esem.rf_model

esem.**rf_model**(*training_params*, *training_data*, *data_processors=None*, *name=''*, *gpu=0*, **args*, ***kwargs*)
Create a simple Random Forest Emulator using sklearn.

Note that because a Random Forest is just a recursive binary partition over the training data, there is no need to normalize/standardize the inputs.

i.e. At least in theory, Random Forests are invariant to monotonic transformations of the independent variables

Parameters

- **training_params** (`pd.DataFrame`) – The training parameters
- **training_data** (`xarray.DataArray` or `iris.cube.Cube` or *array_like*) – The training data - the leading dimension should represent training samples
- **data_processors** (`list` of *esem.data_processors.DataProcessor*) – A list of *DataProcessor* to apply to the data transparently before training. Model output will be untransformed before being returned from the Emulator.
- **name** (`str`) – An optional name for the emulator
- **gpu** (`int`) – The GPU to use (only applicable for multi-GPU) machines
- **args** (`list`) – List of optional arguments for *sklearn.ensemble.RandomForestRegressor*
- **kwargs** (`dict`) – Dict of optional keyword arguments for *sklearn.ensemble.RandomForestRegressor*

Returns `Emulator` – An esem emulator object which can be trained and sampled from

6.2 Emulator

Emulator	A class wrapping a statistical emulator
Emulator.train	Train on the training data
Emulator.predict	Make a prediction using a trained emulator
Emulator._predict	The (internal) predict interface used by e.g., a sampler.
Emulator.batch_stats	Return mean and standard deviation in model predictions over samples, without storing the intermediate predicions in memory to enable evaluating large models over more samples than could fit in memory

6.2.1 esem.emulator.Emulator

class esem.emulator.**Emulator**(*model*, *training_params*, *training_data*, *name=''*, *gpu=0*)

A class wrapping a statistical emulator

training_data

A wrapped representation of the training data

Type *esem.wrappers.DataWrapper*

model

The underlying model which performs the emulation

Type ModelAdaptor

name

A human-readable name for the model

Type str

__init__(*model*, *training_params*, *training_data*, *name=''*, *gpu=0*)

Parameters

- **model** (ModelAdaptor) – The (compiled but not trained) model to be wrapped
- **training_params** (pd.DataFrame or *array-like*) – The training parameters (X)
- **training_data** (*esem.wrappers.DataWrapper* or xarray.DataArray or iris.Cube or *array-like*) – The training data - the leading dimension should represent training samples (Y)
- **name** (str) – Human readable name for the model
- **gpu** (int) – The machine GPU to assign this model to

Methods

__init__(model, training_params, training_data) **Parameters** • **model** (ModelAdaptor) – The (compiled but not trained) model to be wrapped	
batch_stats(sample_points[, batch_size])	Return mean and standard deviation in model predictions over samples, without storing the intermediate predicions in memory to enable evaluating large models over more samples than could fit in memory
predict(x, *args, **kwargs)	Make a prediction using a trained emulator
train([verbose])	Train on the training data

6.2.2 esem.emulator.Emulator.train

Emulator.**train**(*verbose=False*, ***kwargs*)

Train on the training data

Parameters **verbose** (bool) – Print verbose training output to screen

6.2.3 esem.emulator.Emulator.predict

Emulator.**predict**(*x*, **args*, ***kwargs*)

Make a prediction using a trained emulator

Parameters

- **x** (pd.DataFrame or *array-like*) – The points at which to make predictions from the model
- **args** – The specific arguments needed for prediction with this model
- **kwargs** – Any keyword arguments that might need to be passed through to the model

Returns Emulated prediction and variance with the same type as `self.training_data`

6.2.4 esem.emulator.Emulator._predict

Emulator.**_predict**(*x*, **args*, ***kwargs*)

The (internal) predict interface used by e.g., a sampler. It is still in tf but has been post-processed to allow comparison with obs.

Parameters

- **x** (*array-like*) – The points at which to make predictions from the model
- **args** – The specific arguments needed for prediction with this model
- **kwargs** – Any keyword arguments that might need to be passed through to the model

Returns Emulated prediction and variance as either np.ndarray or tf.Tensor

6.2.5 esem.emulator.Emulator.batch_stats

Emulator.**batch_stats**(*sample_points*, *batch_size=1*)

Return mean and standard deviation in model predictions over samples, without storing the intermediate predicions in memory to enable evaluating large models over more samples than could fit in memory

Parameters

- **sample_points** (pd.DataFrame or *array-like*) – The parameter values at which to sample the emulator
- **batch_size** (int) – The number of samples to calculate in each batch. This can be optimised to fill the available (GPU) memory

Returns The batch mean and standard deviation with the same type as `self.training_data`

6.3 Sampler

This class defines the sampling interface currently used by the ABC and MCMC sampling implementations.

Sampler	A class that efficiently samples a Model object for posterior inference
Sampler.sample	This is the call that does the actual inference.

6.3.1 esem.sampler.Sampler

class esem.sampler.**Sampler**(*model*, *obs*, *obs_uncertainty=0.0*, *interann_uncertainty=0.0*, *repres_uncertainty=0.0*, *struct_uncertainty=0.0*, *abs_obs_uncertainty=0.0*, *abs_interann_uncertainty=0.0*, *abs_repres_uncertainty=0.0*, *abs_struct_uncertainty=0.0*)

A class that efficiently samples a Model object for posterior inference

__init__(*model*, *obs*, *obs_uncertainty=0.0*, *interann_uncertainty=0.0*, *repres_uncertainty=0.0*, *struct_uncertainty=0.0*, *abs_obs_uncertainty=0.0*, *abs_interann_uncertainty=0.0*, *abs_repres_uncertainty=0.0*, *abs_struct_uncertainty=0.0*)

Parameters

- **model** (*esem.emulator.Emulator*)
- **obs** (iris.cube.Cube or *array-like*) – The objective
- **obs_uncertainty** (float) – Fractional, relative (1 sigma) uncertainty in observations
- **repres_uncertainty** (float) – Fractional, relative (1 sigma) uncertainty due to the spatial and temporal representitiveness of the observations
- **interann_uncertainty** (float) – Fractional, relative (1 sigma) uncertainty introduced when using a model run for a year other than that the observations were measured in.
- **struct_uncertainty** (float) – Fractional, relative (1 sigma) uncertainty in the model itself.
- **abs_obs_uncertainty** (float) – Fractional, absolute (1 sigma) uncertainty in observations
- **abs_repres_uncertainty** (float) – Fractional, absolute (1 sigma) uncertainty due to the spatial and temporal representitiveness of the observations
- **abs_interann_uncertainty** (float) – Fractional, absolute (1 sigma) uncertainty introduced when using a model run for a year other than that the observations were measured in.
- **abs_struct_uncertainty** (float) – Fractional, absolute (1 sigma) uncertainty in the model itself.

Methods

__init__(model, obs[, obs_uncertainty, ...])	**Parameters** • **model** (*esem.emulator.Emulator*)
sample([prior_x, n_samples])	This is the call that does the actual inference.

6.3.2 esem.sampler.Sampler.sample

Sampler.**sample**(*prior_x=None*, *n_samples=1*)

This is the call that does the actual inference.

It should call model.sample over the prior, compare with the objective, and then output samples from the posterior distribution

Parameters

- **prior_x** (tensorflow_probability.distribution) – The distribution to sample parameters from. By default it will uniformly sample the unit N-D hypercube
- **n_samples** (int) – The number of samples to draw

Returns np.array – Array of samples

6.3.3 MCMCSampler

MCMCSampler	Sample from the posterior using the TensorFlow Markov-Chain Monte-Carlo (MCMC) sampling tools.
MCMCSampler.sample	This is the call that does the actual inference.

esem.sampler.MCMCSampler

class esem.sampler.**MCMCSampler**(*model*, *obs*, ***kwargs*)

Sample from the posterior using the TensorFlow Markov-Chain Monte-Carlo (MCMC) sampling tools. It uses a HamiltonianMonteCarlo kernel.

Notes

Note that NaN observations will create ill-defined likelihoods.

__init__(*model*, *obs*, ***kwargs*)

Parameters

- **model** (*esem.emulator.Emulator*)
- **obs** (iris.cube.Cube or *array-like*) – The objective
- **obs_uncertainty** (float) – Fractional, relative (1 sigma) uncertainty in observations

- **repres_uncertainty** (`float`) – Fractional, relative (1 sigma) uncertainty due to the spatial and temporal representitiveness of the observations
- **interann_uncertainty** (`float`) – Fractional, relative (1 sigma) uncertainty introduced when using a model run for a year other than that the observations were measured in.
- **struct_uncertainty** (`float`) – Fractional, relative (1 sigma) uncertainty in the model itself.
- **abs_obs_uncertainty** (`float`) – Fractional, absolute (1 sigma) uncertainty in observations
- **abs_repres_uncertainty** (`float`) – Fractional, absolute (1 sigma) uncertainty due to the spatial and temporal representitiveness of the observations
- **abs_interann_uncertainty** (`float`) – Fractional, absolute (1 sigma) uncertainty introduced when using a model run for a year other than that the observations were measured in.
- **abs_struct_uncertainty** (`float`) – Fractional, absolute (1 sigma) uncertainty in the model itself.

Methods

__init__(model, obs, **kwargs)	**Parameters** • **model** (*esem.emulator.Emulator*)
sample([prior_x, n_samples, kernel_kwargs, ...])	This is the call that does the actual inference.

esem.sampler.MCMCSampler.sample

MCMCSampler.**sample**(*prior_x=None*, *n_samples=1*, *kernel_kwargs=None*, *mcmc_kwargs=None*)

This is the call that does the actual inference.

It should call model.sample over the prior, compare with the objective, and then output a posterior distribution

Parameters

- **prior_x** (`tensorflow_probability.distribution`) – The distribution to sample parameters from. By default it will uniformly sample the unit N-D hypercube
- **n_samples** (`int`) – The number of samples to draw
- **kernel_kwargs** (`dict`) – kwargs for the MCMC kernel
- **mcmc_kwargs** (`dict`) – kwargs for the MCMC sampler

Returns `np.array` – Array of samples

6.3.4 ABCSampler

ABCSampler	Sample from the posterior using Approximate Bayesian Computation (ABC).
ABCSampler.sample	Sample the emulator over *prior_x* and compare with the observations, returning *n_samples* of the posterior distribution (those points for which the model is compatible with the observations).
ABCSampler.get_implausibility	Calculate the implausibility of the provided sample points, optionally in batches.
ABCSampler.batch_constrain	Constrain the supplied sample points based on the tolerance threshold, optionally in bathes.

esem.abc_sampler.ABCSampler

class esem.abc_sampler.**ABCSampler**(*model*, *obs*, *obs_uncertainty=0.0*, *interann_uncertainty=0.0*, *repres_uncertainty=0.0*, *struct_uncertainty=0.0*, *abs_obs_uncertainty=0.0*, *abs_interann_uncertainty=0.0*, *abs_repres_uncertainty=0.0*, *abs_struct_uncertainty=0.0*)

Sample from the posterior using Approximate Bayesian Computation (ABC). This is a style of rejection sampling.

Notes

Note that emulator samples compared to NaN observations are always treated as 'plausible'.

__init__(*model*, *obs*, *obs_uncertainty=0.0*, *interann_uncertainty=0.0*, *repres_uncertainty=0.0*, *struct_uncertainty=0.0*, *abs_obs_uncertainty=0.0*, *abs_interann_uncertainty=0.0*, *abs_repres_uncertainty=0.0*, *abs_struct_uncertainty=0.0*)

Parameters

- **model** (*esem.emulator.Emulator*)
- **obs** (`iris.cube.Cube` or *array-like*) – The objective
- **obs_uncertainty** (`float`) – Fractional, relative (1 sigma) uncertainty in observations
- **repres_uncertainty** (`float`) – Fractional, relative (1 sigma) uncertainty due to the spatial and temporal representitiveness of the observations
- **interann_uncertainty** (`float`) – Fractional, relative (1 sigma) uncertainty introduced when using a model run for a year other than that the observations were measured in.
- **struct_uncertainty** (`float`) – Fractional, relative (1 sigma) uncertainty in the model itself.
- **abs_obs_uncertainty** (`float`) – Fractional, absolute (1 sigma) uncertainty in observations
- **abs_repres_uncertainty** (`float`) – Fractional, absolute (1 sigma) uncertainty due to the spatial and temporal representitiveness of the observations
- **abs_interann_uncertainty** (`float`) – Fractional, absolute (1 sigma) uncertainty introduced when using a model run for a year other than that the observations were measured in.
- **abs_struct_uncertainty** (`float`) – Fractional, absolute (1 sigma) uncertainty in the model itself.

Methods

__init__(model, obs[, obs_uncertainty, ...])	**Parameters** • **model** (*esem.emulator.Emulator*)
batch_constrain(sample_points[, tolerance, ...])	Constrain the supplied sample points based on the tolerance threshold, optionally in bathes.
get_implausibility(sample_points[, batch_size])	Calculate the implausibility of the provided sample points, optionally in batches.
sample([prior_x, n_samples, tolerance, ...])	Sample the emulator over *prior_x* and compare with the observations, returning *n_samples* of the posterior distribution (those points for which the model is compatible with the observations).

esem.abc_sampler.ABCSampler.sample

ABCSampler.**sample**(*prior_x=None*, *n_samples=1*, *tolerance=0.0*, *threshold=3.0*)

Sample the emulator over *prior_x* and compare with the observations, returning *n_samples* of the posterior distribution (those points for which the model is compatible with the observations).

Parameters

- **prior_x** (`tensorflow_probability.distribution` or `None`) – The distribution to sample parameters from. By default it will uniformly sample the unit N-D hypercube
- **n_samples** (`int`) – The number of samples to draw
- **tolerance** (`float`) – The fraction of samples which are allowed to be over the threshold
- **threshold** (`float`) – The number of standard deviations a sample is allowed to be away from the obs

Returns `ndarray[n_samples]` – Array of samples conforming to the specified tolerance and threshold

esem.abc_sampler.ABCSampler.get_implausibility

ABCSampler.**get_implausibility**(*sample_points*, *batch_size=1*)

Calculate the implausibility of the provided sample points, optionally in batches.

Note this calculates an array of shape (n_sample_points, n_obs) and so can easily exceed available memory if not used carefully.

Parameters

- **sample_points** (`ndarray` or `DataFrame`) – The sample points to calculate the implausibility for
- **batch_size** (`int`) – The size of the batches in which to calculate the implausibility (useful for large samples)

Returns `Cube` – A cube of the impluasibility of each sample against each observation

esem.abc_sampler.ABCSampler.batch_constrain

ABCSampler.**batch_constrain**(*sample_points*, *tolerance=0.0*, *threshold=3.0*, *batch_size=1*)

Constrain the supplied sample points based on the tolerance threshold, optionally in bathes.

Return a boolean array indicating if each sample meets the implausibility criteria:

I < T

Return True (for a sample) if the number of implausibility measures greater than the threshold is less than or equal to the tolerance

Parameters

- **sample_points** (ndarray) – An array of sample points which are to be emulated and compared with the observations
- **tolerance** (float) – The fraction of samples which are allowed to be over the threshold
- **threshold** (float) – The number of standard deviations a sample is allowed to be away from the obs
- **batch_size** (int) – The size of the batches in which to perform the constraining (useful for large samples)

Returns ndarray – A boolean array which is true where the (emulated) samples are compatible with the observations and false otherwise

6.4 Wrappers

ProcessWrapper	This class handles applying any data pre- and post-processing by any provided DataProcessor
DataWrapper	Provide a unified interface for numpy arrays, Iris Cube's and xarray DataArrays.
DataWrapper.name	
DataWrapper.data	
DataWrapper.dtype	
DataWrapper.wrap	Wrap back in a cube if one was provided
CubeWrapper	
DataArrayWrapper	

6.4.1 esem.wrappers.ProcessWrapper

class esem.wrappers.**ProcessWrapper**(*data*, *data_processors=None*)

This class handles applying any data pre- and post-processing by any provided DataProcessor

__init__(*data*, *data_processors=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(data[, data_processors])	Initialize self.
post_process(mean, variance)	Any necessary reshaping or un-weightings are performed here
pre_process(data)	Any necessary rescaling or weightings are performed here

Attributes

data

6.4.2 esem.wrappers.DataWrapper

class esem.wrappers.**DataWrapper**(*data*, *data_processors=None*)

Provide a unified interface for numpy arrays, Iris Cube's and xarray DataArrays. Emulation outputs will be provided based on the provided input type, preserving appropriate metadata.

__init__(*data*, *data_processors=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(data[, data_processors])	Initialize self.
name()	
wrap(data[, name_prefix])	Wrap back in a cube if one was provided

Attributes

data
dtype

6.4.3 esem.wrappers.DataWrapper.name

DataWrapper.**name**()

6.4.4 esem.wrappers.DataWrapper.data

property DataWrapper.**data**

6.4.5 esem.wrappers.DataWrapper.dtype

property DataWrapper.**dtype**

6.4.6 esem.wrappers.DataWrapper.wrap

DataWrapper.**wrap**(*data*, *name_prefix='Emulated '*)

Wrap back in a cube if one was provided

Parameters

- **data** (*np.array*) – Model output to wrap
- **name_prefix** (*str*) –

Returns

6.4.7 esem.wrappers.CubeWrapper

class esem.wrappers.**CubeWrapper**(*cube*, *data_processors=None*)

__init__(*cube*, *data_processors=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(cube[, data_processors])	Initialize self.
name()	
wrap(data[, name_prefix])	Wrap back in a cube if one was provided

Attributes

data
dtype

6.4.8 esem.wrappers.DataArrayWrapper

class esem.wrappers.**DataArrayWrapper**(*dataarray*, *data_processors=None*)

__init__(*dataarray*, *data_processors=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(dataarray[, data_processors])	Initialize self.
name()	
wrap(data[, name_prefix])	Wrap back in a xr.DataArray if one was provided

Attributes

data
dtype

6.5 ModelAdaptor

ModelAdaptor	Provides a unified interface for all emulation engines within ESEm.
SKLearnModel	A wrapper around scikit-learn models.
KerasModel	A wrapper around Keras models
GPFlowModel	A wrapper around GPFlow regression models

6.5.1 esem.model_adaptor.ModelAdaptor

class esem.model_adaptor.**ModelAdaptor**(*model*)

Provides a unified interface for all emulation engines within ESEm. Concrete classes must implement both `train()` and `predict()` methods.

See the API documentation for a list of concrete classes implementing this interface.

__init__(*model*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(model)	Initialize self.
predict(*args, **kwargs)	This is either the tf model which can be called directly, or a generator over the model.predict (in tf, so it's quick).
train(training_params, training_data[, verbose])	Train on the training data :return:

6.5.2 esem.model_adaptor.SKLearnModel

class esem.model_adaptor.**SKLearnModel**(*model*)

A wrapper around scikit-learn models.

__init__(*model*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(model)	Initialize self.
predict(*args, **kwargs)	This is either the tf model which can be called directly, or a generator over the model.predict (in tf, so it's quick).
train(training_params, training_data[, verbose])	Train the RF model.

6.5.3 esem.model_adaptor.KerasModel

class esem.model_adaptor.**KerasModel**(*model*)

A wrapper around Keras models

__init__(*model*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(model)	Initialize self.
predict(*args, **kwargs)	This is either the tf model which can be called directly, or a generator over the model.predict (in tf, so it's quick).
train(training_params, training_data[, ...])	Train the Keras model.

6.5.4 esem.model_adaptor.GPFlowModel

class esem.model_adaptor.**GPFlowModel**(*model*)

A wrapper around GPFlow regression models

__init__(*model*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(model)	Initialize self.
predict(*args, **kwargs)	This is either the tf model which can be called directly, or a generator over the model.predict (in tf, so it's quick).
train(training_params, training_data[, ...])	Train on the training data :return:

6.6 DataProcessor

DataProcessor	A utility class for transparently processing (transforming) numpy arrays and un-processing TensorFlow Tensors to aid in emulation.
Log	Return log(x + c) where c can be specified.
Whiten	Scale the data to have zero mean and unit variance
Normalise	Linearly scale the data to lie between [0, 1]
Flatten	Flatten all dimensions except the leading one
Reshape	Ensure the training data is the right shape for the ConvNet
Recast	Cast the data to a given type

6.6.1 esem.data_processors.DataProcessor

class esem.data_processors.**DataProcessor**

A utility class for transparently processing (transforming) numpy arrays and un-processing TensorFlow Tensors to aid in emulation.

See the API documentation for a list of concrete classes implementing this interface.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
process(data)	
unprocess(mean, variance)	

6.6.2 esem.data_processors.Log

class esem.data_processors.**Log**(*constant=0.0*)

Return log(x + c) where c can be specified.

__init__(*constant=0.0*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([constant])	Initialize self.
process(data)	
unprocess(mean, variance)	

6.6.3 esem.data_processors.Whiten

class esem.data_processors.**Whiten**

Scale the data to have zero mean and unit variance

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
process(data)	
unprocess(mean, variance)	

6.6.4 esem.data_processors.Normalise

class esem.data_processors.**Normalise**

Linearly scale the data to lie between [0, 1]

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
process(data)	
unprocess(mean, variance)	

6.6.5 esem.data_processors.Flatten

class esem.data_processors.**Flatten**

Flatten all dimensions except the leading one

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
process(data)	
unprocess(mean, variance)	

6.6.6 esem.data_processors.Reshape

class esem.data_processors.**Reshape**

Ensure the training data is the right shape for the ConvNet

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
process(data)	
unprocess(mean, variance)	

6.6.7 esem.data_processors.Recast

class esem.data_processors.**Recast**(*new_type*)

Cast the data to a given type

__init__(*new_type*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(new_type)	Initialize self.
process(data)	
unprocess(mean, variance)	

6.7 Utilities

A collection of associated utilities which might be of use when performing typical ESEm workflows.

plot_results	Validation plot for LeaveOneOut
validation_plot	
plot_parameter_space	
get_uniform_params	Slightly convoluted method for getting a flat set of points evenly
get_random_params	Get points randomly sampling a (unit) N-dimensional space
ensemble_collocate	Efficiently collocate (interpolate) many ensemble members on to a set of (un-gridded) observations
leave_one_out	Function to perform LeaveOneOut cross-validation with different models.
get_param_mask	Determine the most relevant parameters in the input space using a regularised linear model and either the Aikake or Baysian Information Criterion.

6.7.1 esem.utils.plot_results

esem.utils.**plot_results**(*ax*, *truth*, *pred*, *title*)
Validation plot for LeaveOneOut

6.7.2 esem.utils.validation_plot

esem.utils.**validation_plot**(*test_mean*, *pred_mean*, *pred_var*, *figsize=(7, 7)*, *minx=None*, *miny=None*, *maxx=None*, *maxy=None*, *ax=None*)

6.7.3 esem.utils.plot_parameter_space

esem.utils.**plot_parameter_space**(*df*, *nbins=100*, *target_df=None*, *smooth=True*, *xmins=None*, *xmaxs=None*, *fig_size=(8, 6)*)

6.7.4 esem.utils.get_uniform_params

esem.utils.**get_uniform_params**(*n_params*, *n_samples=5*)

Slightly convoluted method for getting a flat set of points evenly sampling a (unit) N-dimensional space

Parameters

- **n_params** (int) – The number of parameters (dimensions) to sample from
- **n_samples** (int) – The number of uniformly spaced samples (in each dimension)

Returns ndarray – n_samples**n_params parameters uniformly sampled

6.7.5 esem.utils.get_random_params

esem.utils.**get_random_params**(*n_params*, *n_samples=5*)

Get points randomly sampling a (unit) N-dimensional space

Parameters

- **n_params** (int) – The number of parameters (dimensions) to sample from
- **n_samples** (int) – The number of parameters to (radnomly) sample

6.7.6 esem.utils.ensemble_collocate

esem.utils.**ensemble_collocate**(*ensemble*, *observations*, *member_dimension='job'*)

Efficiently collocate (interpolate) many ensemble members on to a set of (un-gridded) observations

Note: This function requires both Iris and CIS to be installed

Parameters

- **ensemble** (`GriddedData`) – The ensemble of (model) samples to interpolate on to the observations
- **observations** (`UngriddedData`) – The observations on to which the observations will be sampled
- **member_dimension** (`str`) – The name of the dimension which represents the ensemble members in *ensemble*

Returns **col_ensemble** (`iris.cube.Cube`) – The ensemble values interpolated on to the observation locations, with the ensemble members along the leading dimension.

6.7.7 esem.utils.leave_one_out

`esem.utils.`**`leave_one_out`**(*Xdata*, *Ydata*, *model='RandomForest'*, ***model_kwargs*)

Function to perform LeaveOneOut cross-validation with different models.

Parameters

- **Xdata** (*array-like* of `shape (n_samples, n_features)`) – Parameter values
- **Ydata** (*array-like* of `shape (n_samples,)`) – Target values.
- **model** (`{'RandomForest', 'GaussianProcess', 'NeuralNet'}`, *default* `'RandomForest'`)
- **model_kwargs** (`dict`) – More arguments to pass to the model.

Returns **output** (`list` of `n_samples (truth, prediction, variance) tuples`) – which can then be passed to *validation_plot()*

6.7.8 esem.utils.get_param_mask

`esem.utils.`**`get_param_mask`**(*X*, *y*, *criterion='bic'*, ***kwargs*)

Determine the most relevant parameters in the input space using a regularised linear model and either the Aikake or Baysian Information Criterion.

Parameters

- **X** (*array-like* of `shape (n_samples, n_features)`) – Parameter values
- **y** (*array-like* of `shape (n_samples,)`) – target values.
- **criterion** (`{'bic', 'aic'}`, *default* `'bic'`) – The information criteria to apply for parameter selection. Either Aikake or Baysian Information Criterion.
- **kwargs** (`dict`) – Further arguments for sklearn.feature_selection.SelectFromModel

Returns **mask** (`ndarray`) – A boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention.

CHAPTER

SEVEN

ESEM DESIGN

Here we provide a brief description of the main architectural decisions behind the design for ESEm in order to hopefully make it easier for contributors and users alike to understand the various components and how they fit together.

7.1 Emulation

We try to provide a seamless interface for users whether they provide iris Cube's, xarray DataArray's or numpy ndarrays. This is done using the *esem.wrappers.DataWrapper* and associated subclasses, which keep a copy of the provided object but only exposes the underlying numpy array to the emulation engines. When the data is requested from this wrapper using the *esem.wrappers.DataWrapper.wrap()* method then it will return a copy of the input object (Cube or DataArray) with the data replaced by the emulated data.

This layer will also ensure the underlying (numpy) data is wrapped in a *esem.wrappers.ProcessWrapper*. This class transparently applies any requested *esem.data_processors.DataProcessor* in sequence.

The user can then create an *esem.emulator.Emulator* object by providing a concrete *esem.model_adaptor.ModelAdaptor* such as a *esem.model_adaptor.KerasModel*. There are two layers of abstraction here: The first to deal with different interfaces to different emulation libraries; and the second to apply the pre- and post-processing and allow a single *esem.emulator.Emulator.batch_stats()* method. The *esem.emulator.Emulator._predict()* provides an important internal interface to the underlying model which reverts any data-processing but leaves the emulator output as a TensorFlow Tensor to allow optimal sampling.

The top-level functions *esem.gp_model()*, *esem.cnn_model()* and *esem.rf_model()* provide an simple interface for constructing these emulators and should be sufficient for most users.

7.2 Calibration

We try and keep this interface very simple; a *esem.sampler.Sampler* should be initialised with an *esem.emulator.Emulator* object to sample from, some observations and associated uncertainties. The only method it has to provide is *esem.sampler.Sampler.sample()* which should provide sample θ from the posterior.

Wherever possible these samplers should take advantage of the fact that the *esem.emulator.Emulator._predict()* method returns TensorFlow tensors and always prefer to use them directly rather than using *esem.emulator.Emulator.predict()* or calling *.numpy()* on them. This allows the sampling to happen on GPUs where available and can substantially speed-up sampling.

The *esem.abc_sampler.ABCSampler* extends this interface to include both *esem.abc_sampler.ABCSampler.get_implausibility()* and *esem.abc_sampler.ABCSampler.batch_constrain()* methods. The first allows inspection of the effect of different observations on the constraint and the second allows a streamlined approach for rejecting samples in batch, taking advantage of the large amounts of memory available on modern GPUs.

CHAPTER

EIGHT

GLOSSARY

array_like Any object that can be treated as a numpy array, i.e. can be indexed to retrieve numerical values. Typically not a tensorflow Tensor.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

D

E

F

G

K

L

M

N

P

R

S

T

V

W