elm Documentation

Release 0.1.0

elm contributors

Jul 20, 2019

Getting Started

1 Why Elm? 3

2 Use Cases 5

3 Installation 9

4 Quick Start 11

5 Tutorials 15

6 Examples 31

7 ElmStore 33

8 Pipeline 41

9 Customizable Pipeline Steps 45

10 Multi-Model Fitting I: Ensemble Fitting 51

11 Multi-Model Fitting II: Evolutionary Algorithms 55

12 Multi-Model Prediction 59

13 `elm` `yaml` Specs - *Deprecated Temporarily* 65

14 elm-main Entry Point *Deprecated Temporarily* 71

15 API 75

16 Environment Variables 77

17 `py.test` Unit Tests 79

18 Longer Running Tests 81

19 Contributing to `elm` 83

20 Release Procedure 85

This website documents the archived prototype ELM software developed in NASA SBIR Phase I and II from 2016 to January 2018. Current versions of this code are available at EarthML.pyviz.org; the code documented here is primarily of historical interest.

Getting Started

- *Why Elm?*
- *Use Cases*
- *Installation*
- *Quick Start*
- *Tutorials*
- *Examples*

CHAPTER 1

Why Elm?

Ensemble Learning Models (elm) is a set of tools for creating multiple unsupervised and supervised machine learning models and training them in parallel on datasets too large to fit into the RAM of a single machine, with a focus on applications in climate science, GIS, and satellite imagery.

Some reasons for using elm over scikit-learn alone:

- Parallelize ML pipelines across the cores of a single machine or compute cluster
- Use out-of-core ML algorithms to process large datasets which are too large to fit into RAM
- Analyze multidimensional climate data, extending beyond the limitations of two-dimensional arrays and matrices
- Read data from file formats popular to climate science and GIS, such as netCDF, HDF4, HDF5, Shapefiles, GeoJSON, and GeoTIFF

More use-cases can be found here.

1.1 `elm` is a Work in Progress

`elm` is immature and largely for experimental use.

The developers do not promise backwards compatibility with future versions.

1.2 Next steps

- *Use Cases for elm*
- *Install elm*
- Try the example notebooks

CHAPTER 2

Use Cases

`elm` (**Ensemble Learning Models**) is a versatile set of tools for ensemble and evolutionary algorithm approaches to training and selecting machine learning models and large scale prediction from trained models. `elm` has a focus on data structures that are common in satellite and weather data analysis, such as rasters representing bands of satellite data or cubes of weather model output.

Common computational challenges in satellite and weather data machine learning include:

- *Large-Scale Model Training*
- *Model Uncertainty*
- *Hyperparameterization / Model Selection*
- *Data/Metadata Formats*
- *Preprocessing Input Data*
- *Predicting for Many Large Samples and/or Models*

To address these challenges `elm` draws from existing Python packages:

- dask-distributed: `elm` uses dask-distributed for parallelism over ensemble fitting and prediction
- scikit-learn : `elm` can use unsupervised and supervised models, preprocessors, scoring functions, and postprocessors from `scikit-learn` or any estimator that follows the *scikit-learn* initialize / fit / predict estimator interface.
- xarray : `elm` wraps xarray data structures for n-dimensional arrays, such as 3-dimensional weather cubes, and for collections of 2-D rasters, such as a LANDSAT sample

2.1 Large-Scale Model Training

`elm` offers the following strategies for large scale training:

- Use of `partial_fit` for incremental training on series of saemples

- Ensemble modeling, training batches of models in generations in parallel, with model selection after each generation
- Use of a *Pipeline* with a sequence of *transformation steps*
- `partial_fit` for incremental training of transformers used in `Pipeline` steps, such as PCA
- Custom user-given model selection logic in ensemble approaches to training

`elm` can use `dask` to parallelize the activities above.

2.2 Model Uncertainty

Ensemble modeling can be used to account for uncertainty that arises from uncertain model parameters or uncertainty in the fitting process. The ensemble approach in `elm` allows training and prediction from an ensemble where model parameters are varied, including parameters related to preprocessing transformations, such as feature selection or PCA transforms. See the *predict_many* example.

2.3 Hyperparameterization / Model Selection

`elm` offers two different algorithms for multi-model training with model selection:

- *fit_ensemble*: Running one batch of models at a time (a generation), running a user-given model selection function after each generation
- *fit_ea*: Using the NSGA-2 evolutionary algorithm to select best parameters for the best fit.

In either of these algorithms `elm` can use most of the model scoring features of `scikit-learn` or a user-given model scoring callable.

See also:

- *fit_ensemble*
- *fit_ea*
- `elm.model_selection` in *API docs*
- `scikit-learn` scoring classes that work with `elm`

2.4 Data/Metadata Formats

One challenge in satellite and weather data processing is the variety of input data formats, including GeoTiff, NetCDF, HDF4, HDF5, and others. `elm` offers a function `load_array` which can load spatial array data in the following formats:

- GeoTiff: Loads files from a directory of GeoTiffs, assuming each is a single-band raster
- NetCDF: Loads variables from a NetCDF file
- HDF4 / HDF5: Loads subdatasets from HDF4 and HDF5 files

`load_array` creates an `ElmStore` (read more here), a fundamental data structure in `elm` that is essentially an `xarray.Dataset` with metadata standardization over the various file types.

2.5 Preprocessing Input Data

`elm` has a wide range of support for preprocessing activities. One important feature of `elm` is its ability to train and/or predict from more than one sample and for each sample run a series of preprocessing steps that may include:

- Scaling, adding polynomial features, or other preprocessors from `sklearn.preprocessing`
- Feature selection using any class from `sklearn.feature_selection`
- Flattening collections of rasters to a single 2-D matrix for fitting / prediction
- Running user-given sample transformers
- Resampling one raster onto another raster's coordinates
- In-polygon selection
- Feature extraction through transform models like PCA or ICA

See *elm.pipeline.steps* for more information on preprocessing.

2.6 Predicting for Many Large Samples and/or Models

`elm` can use dask-distributed, a dask thread pool, or serial processing for predicting over a group (ensemble) of models and a single sample or series of samples. `elm`'s interface for large scale prediction, described here, is via the *predict_many* method of a `Pipeline` instance.

2.6.1 `elm` Capabilities

- *Ensemble learning*
- *Large scale prediction*
- *Genetic algorithms*
- *Common preprocessing operations for satellite imagery and climate data*

These capabilities are best shown in the

- *Elm introduction*
- *Elm clustering introduction*
- *Other elm examples*
- *Use cases*

CHAPTER 3

Installation

There are two options for installing *elm*:

- *Install with Conda*
- *Install from Source*

3.1 Install with Conda

Conda is a package manager backed by Continuum Analytics, Inc. Notable features include first-class support for Python and R software that depends on C extensions, cross-platform support for Windows/Mac OSX/Linux, and standalone packages that are easy to distribute and deploy.

3.1.1 Stable

The "stable" release is more thoroughly tested, but does not include the latest experimental features:

```
conda create -c elm -c conda-forge -c ioam -c scitools --name earth-env elm earthio
```

The above command creates a conda environment with the latest stable releases of *elm* and *earthio* installed. To begin using, activate the environment:

```
source activate earth-env
```

If you encounter any issues with installation of the latest release from `conda` or source, then raise a github issue in the elm repo here or email psteinberg [at] continuum [dot] io.

3.1.2 Development

The "development" releases are less stable, but include newer features:

```
conda create -c elm/label/dev -c conda-forge -c ioam -c scitools/label/dev --name␣
↪earth-env-dev python=3.5 elm earthio
```

Like for the stable release, activate the environment to begin using *elm*:

```
source activate earth-env-dev
```

3.2 Install from Source

Installing *elm* from source is recommended if you want to develop *elm* features and iterate rapidly over your code changes. To install `elm` from source:

```
git clone https://github.com/ContinuumIO/elm
cd elm
export ELM_EXAMPLE_DATA_PATH=~/elm-data
PYTHON_TEST_VERSION=3.5 EARTHIO_INSTALL_METHOD=conda . build_elm_env.sh
```

Verify the install with:

```
python -c "from earthio import *;from elm import *"
```

You may want to add the following line to your .bashrc (or equivalent shell config) to avoid >1 download of the test data and have the test data discovered by `py.test`:

```
export ELM_EXAMPLE_DATA_PATH=~/elm-data
```

Do the tutorials and examples:

- *K-Means with LANDSAT example*
- *Examples*

CHAPTER 4

Quick Start

The following steps generate a visualization using *elm* on a synthetic dataset. As development on *elm* continues we strive to condense this document into a smaller example. For now, it offers insight into *elm*'s customizability and extensive feature set.

4.1 Step 1 - Choose Model(s)

First import model(s) from scikit-learn and `Pipeline` and `steps` from `elm.pipeline`:

```
from elm.config import client_context
from earthio.filters.make_blobs import random_elm_store
from elm.pipeline import Pipeline, steps
from sklearn.decomposition import PCA
from sklearn.cluster import AffinityPropagation
```

- `random_elm_store` is a function that returns random rasters (`xarray.DataArray` s) in an *ElmStore*, a data structure similar to an xarray.Dataset
- `steps` is a module of *all the transformation steps possible in a Pipeline*

See the *LANDSAT K-Means* and *other examples* to see how to read an *ElmStore* from GeoTiff, `HDF4`, `HDF5`, or `NetCDF`.

4.2 Step 2 - Define a `sampler`

If fitting more than one sample, then define a `sampler` function to pass to *fit_ensemble*. Here we are using a partial of `random_elm_store` (synthetic data). If using a `sampler` , we also need to define `args_list` a list of tuples where each tuple can be unpacked as arguments to `sampler`. The length of `args_list` determines the number of samples potentially used. Here we have 2 empty tuples as `args_list` because our `sampler` needs no arguments and we want 2 samples. Alternatively the arguments `X` , `y` , and `sample_weight` may be given in place of `sampler` and `args_list` .

```
from functools import partial
N_SAMPLES = 2
bands = ['band_{}'.format(idx + 1) for idx in range(10)]
sampler = partial(random_elm_store,
                  bands=bands,
                  width=60,
                  height=60)
args_list = [(),] * N_SAMPLES
```

4.3 Step 2 - Define a Pipeline

The code block below will use `Flatten` to convert each 2-D raster (`DataArray`) to give a single 1-D column in 2-D `DataArray` for machine learning. The output of `Flatten` will be in turn passed to sklearn.decomposition.PCA and the reduced feature set from `PCA` will be passed to the sklearn.cluster.AffinityPropagation clustering model.

```
pipe = Pipeline([('flat', steps.Flatten()),
                 ('pca', steps.Transform(PCA())),
                 ('aff_prop', AffinityPropagation())])
```

4.4 Step 3 - Call fit_ensemble with `dask`

Now we can use *fit_ensemble* to fit to one or more samples and one more instances of the `pipe` *Pipeline* above. Below we are passing the `sampler` and `args_list`, `client`, which will be a `dask-distributed` or `ThreadPool` or None, depending on *environment variables*. `init_ensemble_size` sets the number of *Pipeline* instances and `models_share_sample=False` means to fit all `Pipeline` / sample combinations (`2 X 2 == 4` total members in this case).

```
pipe.fit_ensemble(sampler=sampler,
                  args_list=args_list,
                  init_ensemble_size=2,
                  models_share_sample=False,
                  ngen=1)
```

The code block with *fit_ensemble* above would show the `repr` of the `Pipeline` object as follows:

```
>>> print(pipe)
<elm.pipeline.Pipeline> with steps:
    flat: <elm.steps.Flatten>:

    pca: <elm.steps.Transform>:
        copy: True
        iterated_power: 'auto'
        n_components: None
        partial_fit_batches: None
        random_state: None
        svd_solver: 'auto'
        tol: 0.0
        whiten: False
    aff_prop: AffinityPropagation(affinity='euclidean', convergence_iter=15,␣
↪copy=True,
              damping=0.5, max_iter=200, preference=None, verbose=False)
```

We can confirm that we have 4 *Pipeline* instances in the trained ensemble:

```
>>> len(pipe.ensemble)
4
```

4.5 Step 4 - Call predict_many

predict_many will by default predict from the ensemble that was just trained (4 models in this case). *predict_many* takes `sampler` and `args_list` like *fit_ensemble*. The `args_list` may differ from that given to `fit_ensemble` or be the same. We have 4 trained models in the `.ensemble` attribute of `pipe` and 2 samples specified by `args_list` , so *predict_many* returns a list of 8 prediction :doc:‘ElmStore<elm-store>‘s

```
preds = pipe.predict_many(sampler=sampler, args_list=args_list)
example = preds[0]
```

```
import matplotlib.pyplot as plt
example.predict.plot.pcolormesh()
plt.show()
```

Read More : *LANDSAT K-Means example*

CHAPTER 5

Tutorials

This page walks through a `Jupyter` notebook using `elm` to ensemble fit K-Means and predict from all members of the ensemble.

It demonstrates the common steps of using `elm` :

- Working with `earthio.load_array` to read `NetCDF` , `HDF4` , `HDF5` , and GeoTiff files, and controlling how a sample is composed of bands or separate rasters with `LayerSpec` . See also *Creating an ElmStore from File*
- Defining a `Pipeline` of transformers (e.g. normalization and PCA) and an estimator, where the transformers use classes from `elm.pipeline.steps` and the estimator is a model with a `fit` / `predict` interface. See also *Pipeline*
- Calling *fit_ensemble* to train the *Pipeline* under varying parameters with one or more input samples
- Calling *predict_many* to predict from all trained ensemble members to one or more input samples

5.1 LANDSAT

The LANDSAT classification is notebook from elm examples. This section walks through that notebook, pointing out:

- How to use `earthio` for scientific data files like GeoTiffs
- How to set up an `elm.pipeline.Pipeline` of transformations
- How to use `dask` to `fit` a `Pipeline` in ensemble and predict from many models

NOTE : To follow along, make sure you follow the *Prerequisites for Examples*. The LANDSAT sample used here can be found in the AWS S3 LANDSAT bucket here.

5.2 earthio Walk-Through

First the notebook sets some environment variables related to usage of a `dask-distributed Client` and the path to the GeoTiff example files from **'elm-data'_**:

Each GeoTiff file has 1 raster (band of LANDSAT data):

Most bands are at a resolution of about 30 meters, but band 8 is panchromatic with 15 m resolution.

In [2]:
```
[os.path.basename(tif) for tif in TIFS]
```

Out[2]:
```
['LC80150332013207LGN00_B1.TIF',
 'LC80150332013207LGN00_B10.TIF',
 'LC80150332013207LGN00_B11.TIF',
 'LC80150332013207LGN00_B2.TIF',
 'LC80150332013207LGN00_B3.TIF',
 'LC80150332013207LGN00_B4.TIF',
 'LC80150332013207LGN00_B5.TIF',
 'LC80150332013207LGN00_B6.TIF',
 'LC80150332013207LGN00_B7.TIF',
 'LC80150332013207LGN00_B8.TIF',
 'LC80150332013207LGN00_B9.TIF',
 'LC80150332013207LGN00_BQA.TIF']
```

See more inforation on `ElmStore` in *ElmStore*.

5.3 earthio.LayerSpec

Using a list of `LayerSpec` objects, as shown below, is how one can control which bands, or individual GeoTiff files, become the sample dimensions for learning:

- `buf_xsize`: The size of the output raster horizontal dimension
- `buf_ysize`: The size of the output raster vertical dimension
- `name`: What do call the band in the `ElmStore` returned. For example `band_1` as a name will mean you can use `X.band_1` and find `band_1` as a key in `X.data_vars`.
- `search_key`: Where to look for the band identifiying info, in this case the file `name`
- `search_value` : What string token identifies a band, e.g. `B1.TIF` (see file names printed above)

We are using `buf_xsize` and `buf_ysize` below to downsample.

In [3]:
```
from elm.readers import load_array, load_tif_meta, BandSpec

# Detect the native height and width of one TIF
# they are all the same except for band 8, the panchromatic band
handle, meta = load_tif_meta([t for t in TIFS if 'B1.TIF' in t][0])

# Downsample for example
DOWNSAMPLE = 8

# Make band_specs
band_specs = []
for band in (list(range(1, 8)) + list(range(9, 12))): # skip panchromatic band at 2x resolution
    b = BandSpec(search_key='name',
                 search_value='B{}.TIF'.format(band),
                 name='band_{}'.format(band),
                 buf_xsize=meta['meta']['width'] // DOWNSAMPLE,
                 buf_ysize=meta['meta']['height'] // DOWNSAMPLE)
    band_specs.append(b)
band_specs
```

Check the `repr` of the `LayerSpec` objects to see all possible arguments controlling reading of bands:

```
Out[3]: [BandSpec(search_key='name', search_value='B1.TIF', name='band_1', key_re_flags=None, value_r
        e_flags=None, buf_xsize=978, buf_ysize=996, window=None, meta_to_geotransform=None, stored_co
        ords_order=('y', 'x'), down_sample=None),
         BandSpec(search_key='name', search_value='B2.TIF', name='band_2', key_re_flags=None, value_r
        e_flags=None, buf_xsize=978, buf_ysize=996, window=None, meta_to_geotransform=None, stored_co
        ords_order=('y', 'x'), down_sample=None),
         BandSpec(search_key='name', search_value='B3.TIF', name='band_3', key_re_flags=None, value_r
        e_flags=None, buf_xsize=978, buf_ysize=996, window=None, meta_to_geotransform=None, stored_co
        ords_order=('y', 'x'), down_sample=None),
         BandSpec(search_key='name', search_value='B4.TIF', name='band_4', key_re_flags=None, value_r
        e_flags=None, buf_xsize=978, buf_ysize=996, window=None, meta_to_geotransform=None, stored_co
        ords_order=('y', 'x'), down_sample=None),
         BandSpec(search_key='name', search_value='B5.TIF', name='band_5', key_re_flags=None, value_r
        e_flags=None, buf_xsize=978, buf_ysize=996, window=None, meta_to_geotransform=None, stored_co
        ords_order=('y', 'x'), down_sample=None),
         BandSpec(search_key='name', search_value='B6.TIF', name='band_6', key_re_flags=None, value_r
        e_flags=None, buf_xsize=978, buf_ysize=996, window=None, meta_to_geotransform=None, stored_co
        ords_order=('y', 'x'), down_sample=None),
         BandSpec(search_key='name', search_value='B7.TIF', name='band_7', key_re_flags=None, value_r
        e_flags=None, buf_xsize=978, buf_ysize=996, window=None, meta_to_geotransform=None, stored_co
        ords_order=('y', 'x'), down_sample=None),
         BandSpec(search_key='name', search_value='B9.TIF', name='band_9', key_re_flags=None, value_r
        e_flags=None, buf_xsize=978, buf_ysize=996, window=None, meta_to_geotransform=None, stored_co
        ords_order=('y', 'x'), down_sample=None),
         BandSpec(search_key='name', search_value='B10.TIF', name='band_10', key_re_flags=None, value
        _re_flags=None, buf_xsize=978, buf_ysize=996, window=None, meta_to_geotransform=None, stored_
        coords_order=('y', 'x'), down_sample=None),
         BandSpec(search_key='name', search_value='B11.TIF', name='band_11', key_re_flags=None, value
        _re_flags=None, buf_xsize=978, buf_ysize=996, window=None, meta_to_geotransform=None, stored_
        coords_order=('y', 'x'), down_sample=None)]
```

5.4 `earthio.load_array`

`load_array` aims to find a file reader for a `NetCDF`, `HDF4`, `HDF5`, or GeoTiff source.

The first argument to `load_array` is a directory if reading GeoTiff files and it is assumed that the directory contains GeoTiff files each with a 1-band raster.

For `NetCDF`, `HDF4`, and `HDF5` the first argument is a single filename, and the bands are taken from the `variables` (`NetCDF`) or `subdatasets` (`HDF4` / `HDF5`).

`band_specs` (list of `LayerSpec` objects) is passed in to `load_array` (the list of `LayerSpec` objects from above) to control which bands are read from the directory of GeoTiffs.

```
In [ ]: X = load_array(TIF_DIR, band_specs=band_specs)
```

5.5 Using an `ElmStore` like an (xarray.Dataset)

See also xarray docs on Dataset

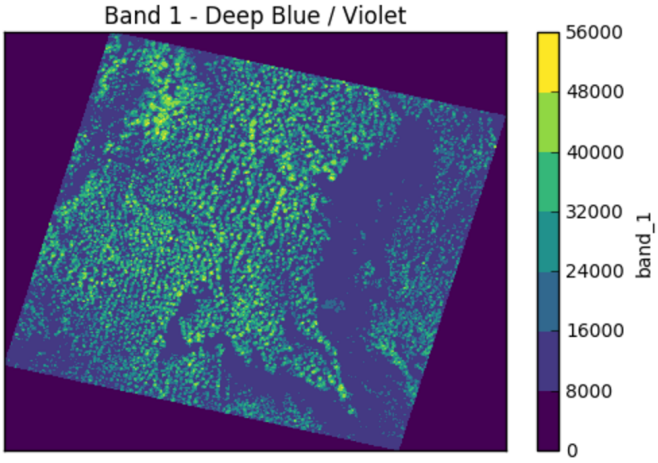
```
In [5]: X.band_1.values.shape
Out[5]: (996, 978)

In [6]: X.band_1.min(), X.band_1.max()
Out[6]: (<xarray.DataArray 'band_1' ()>
         array(0, dtype=uint16), <xarray.DataArray 'band_1' ()>
         array(52932, dtype=uint16))
```

5.6 Visualization with `ElmStore`

The notebook then goes through a number of examples similar to:

- `X.band_1.plot.pcolormesh()` - The code uses names like `band_1`, `band_2`. These are named `DataArray` objects in the `ElmStore` `X` because of the `name` argument to the `LayerSpec` objects above. The `plot.pcolormesh()` comes from the data viz tools with xarray.DataArray .
- The output of `X.band_1.plot.pcolormesh()`

```
X.band_1.plot.contourf()
plt.title('Band 1 - Deep Blue / Violet')
no_axis_labels()
plt.show()
```



5.7 Building a `Pipeline`

Building an `elm.pipeline.Pipeline` of transformations is similar to the idea of a *Pipeline* in scikit-learn.

- All steps but the last step in a *Pipeline* must be instances of classes from the *elm.pipeline.steps* - these are the transformers.
- The final step in a *Pipeline* should be an estimator from *scikit-learn* with a *fit/predict* interface.

The notebook shows how to specify a several-step *Pipeline* of

- Flattening rasters

- Assigning *NaN* where needed
- Dropping *NaN* rows
- Standardizing (Z-scoring) by band means and standard deviations
- Adding polynomial interaction terms of degree two
- Transforming with PCA
- K-Means with *partial_fit* several times per model

Preamble - Imports

This cell show typical import statments for working with a `elm.pipeline.steps` that become part of a `Pipeline`, including importing a transformer and estimator from scikit-learn:

In [12]:
```
from elm.pipeline import steps, Pipeline
import numpy as np
from sklearn.decomposition import PCA
from sklearn.cluster import MiniBatchKMeans
```

5.8 Steps - `Flatten`

This transform-flatten step is essentially `.ravel` on each `DataArray` in `X` to create a single 2-D `DataArray` :

`steps.Flatten` - flatten each raster

Flatten each 2-d raster (currently as `DataArrays`) into a single `DataArray` called `flat` inside an `elm.readers.ElmStore`.

See also `elm.readers.reshape.flatten` (called by `steps.Flatten`)

In [13]:
```
flat = steps.Flatten()
```

5.9 Steps - `ModifySample` - `set_nans`

The next step uses `elm.pipeline.steps.ModifySample` to run a custom callable in a `Pipeline` of transformations. This function sets `NaN` for the no-data perimeters of the rasters:

Set `NaN` in out-of-sample regions

You may have noticed in the visualizations above that the study region has a boundary of no-data that is expressed as zeros.

The cell below shows how to use `steps.ModifySample` to use your own callable in a `Pipeline`.

Note the signature of `set_nans` example below, a callable given to `steps.ModifySample`. The signature must always be

`func(X, y=None, sample_weight=None, **kwargs)`

And the return value type must always be a tuple of:

`(X, y, sample_weight)`

`y` and/or `sample_weight` can be `None` if using unsupervised models.

In [14]:
```
def set_nans(X, y=None, sample_weight=None, **kwargs):
    for band in X.data_vars:
        band_arr = getattr(X, band)
        band_arr.values = band_arr.values.astype(np.float32)
        band_arr.values[band_arr.values == 0] = np.NaN
    return (X, y, sample_weight)
```

In [ ]:
```
set_nans_step = steps.ModifySample(set_nans)
```

5.10 Steps - `DropNaRows` - Drop Null / NaN Rows

The transform-dropnarows is a transformer to remove the `NaN` values from the `DataArray flat` (the flattened (`ravel`) rasters as a single 2-D `DataArray`)

Drop `NaN`s from sample

This will drop `NaN` rows, but *note*: the sample must have gone through `Flatten` first or be similar in data structure to an output of `Flatten` (an `ElmStore` with a single 2-D `DataArray` called `flat`)

In [15]:
```
drop_na = steps.DropNaRows()
```

5.11 Steps - `ModifySample` - Log Transform (or pass through)

This usage of `ModifySample` will allow the `Pipeline` to use log transformation or not (see usage of `set_params` several screenshots later)

Log Transform

Here we use `sklearn.preprocessing.FunctionTransformer` as it is wrapped by `elm.pipeline.steps`. Note - `FunctionTransformer` must be called on a sample that has gone through `Flatten`, otherwise use `ModifySample`. `FunctionTransformer` may be easier in some cases because it operates on a single 2-D matrix rather than separate `DataArrays`

In [16]:
```
def log_or_not_log(X, do_log=True, **kwargs):
    '''Log transform if do_log but only ever the positive semi-definite columns

    This function allows the Pipeline to switch between log / no log transform
    Parameters
        X: numpy 2-d array
        do_log: Do natural log if do_log
        kwargs: ignored
    Returns:
        numpy array modified in place
    '''
    if do_log:
        for idx in range(X.shape[1]):
            if not np.any(X[:, idx] < 0):
                X[X[:, idx] == 0] = 0.01
                X[:, idx] = np.log(X[:, idx])
    return X
log_transform_or_not = steps.FunctionTransformer(func=log_or_not_log, params=('do_log',))
```

5.12 Feature engineering in a `Pipeline`

Define a function that can do normalized differences between bands (raster or `DataArray`), adding the normalized differences to what will be the `X` data in the `Pipeline` of transformations.

Feature engineering - Add band ratios

As an example of feature engineering, we make function to use in the `Pipeline` that adds ratios between different bands. Normalized differences between bands can highlight patterns not apparent in the individual bands. Note the signature mentioned is the same as the preceding `set_nans` function.

In [17]:
```
from elm.readers import ElmStore
import xarray as xr
def add_band_ratios(X, y=None, sample_weight=None, **kwargs):
    ratios = kwargs['ratios']
    bands = X.band_order.copy()
    es = {}
    for idx, (key, (b1, b2)) in enumerate(sorted(ratios.items())):
        band1 = getattr(X, b1)
        band2 = getattr(X, b2)
        normed_diff = (band1 - band2) / (band1 + band2)
        es[key] = normed_diff
        es[key].attrs['canvas'] = band1.canvas
        bands.append(key)
    Xnew = xr.merge([ElmStore(es, add_canvas=False), X])
    Xnew.attrs['band_order'] = bands
    return (Xnew, y, sample_weight)
```

5.13 Feature engineering - `ModifySample` with arguments

And here is how the function above can be used in a `Pipeline` (wrapping with `elm.pipeline.steps.ModifySample`):

We are calculating:

- `NDWI` : *Normalized Difference Water Index* * `(band_4 - band_5) / (band_4 + band_5)`
- `NDVI` : *Normalized Difference Vegetation Index* * `(band_5 - band_4) / (band_5 + band_4)`
- `NDSI` : *Normalized Difference SnowIndex* * `(band_2 - band_6) / (band_2 + band_6)`
- `NBR` : *Normalized Burn Ratio* * `(band_4 - band_7) / (band_7 + band_4)`

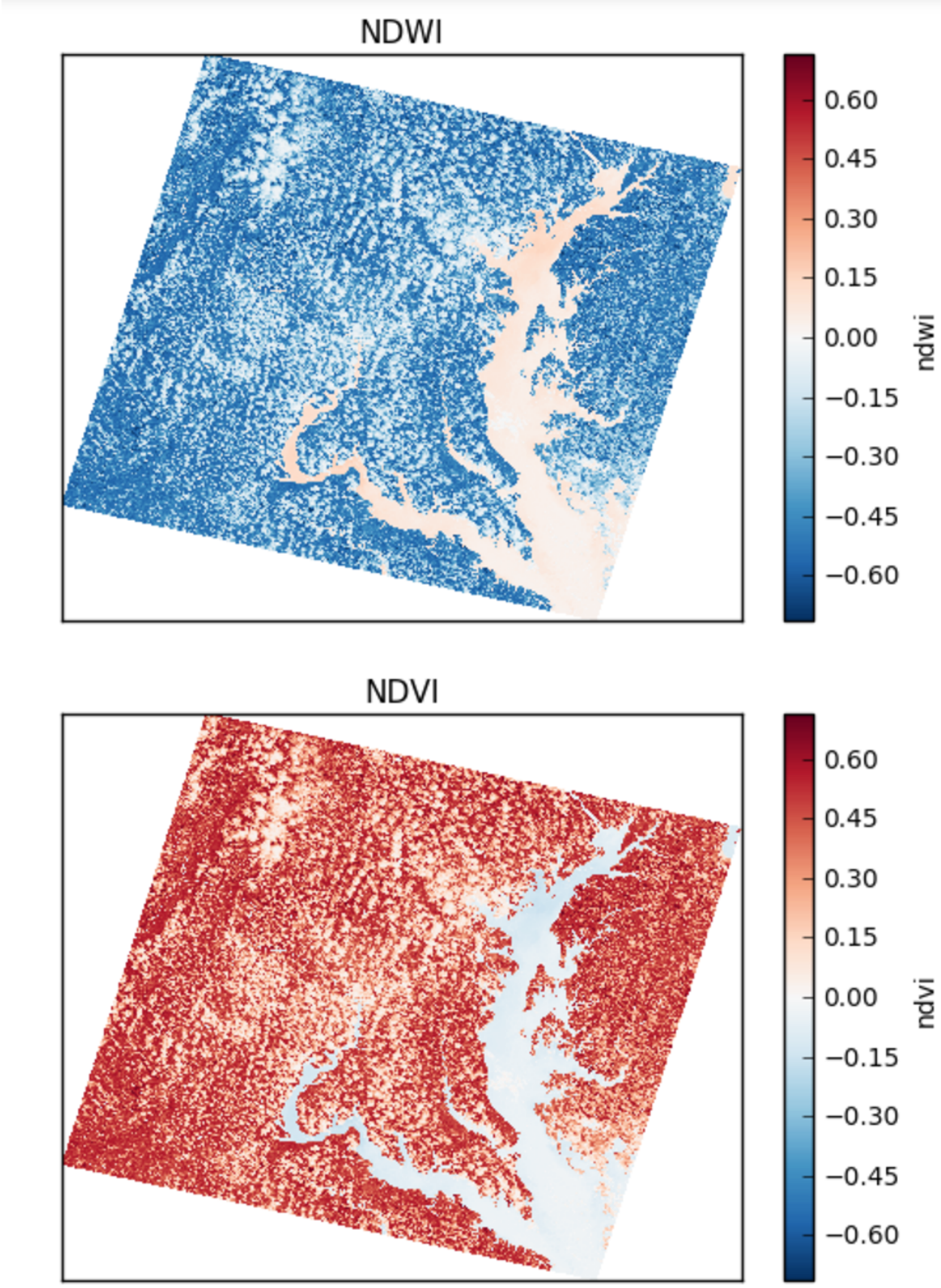
Using `ModifySample` to add bands

Keyword arguments to `ModifySample` are in turn passed to the callable given. Here we pass `ratios` to control which band normalized differences are added to the sample.

In [18]:
```
ratios = {'ndwi': ('band_4', 'band_5'),
          'ndvi': ('band_5', 'band_4'),
          'ndsi': ('band_2', 'band_6'),
          'nbr':  ('band_4', 'band_7'),
         }
normed_diffs_step = steps.ModifySample(add_band_ratios, ratios=ratios)
```

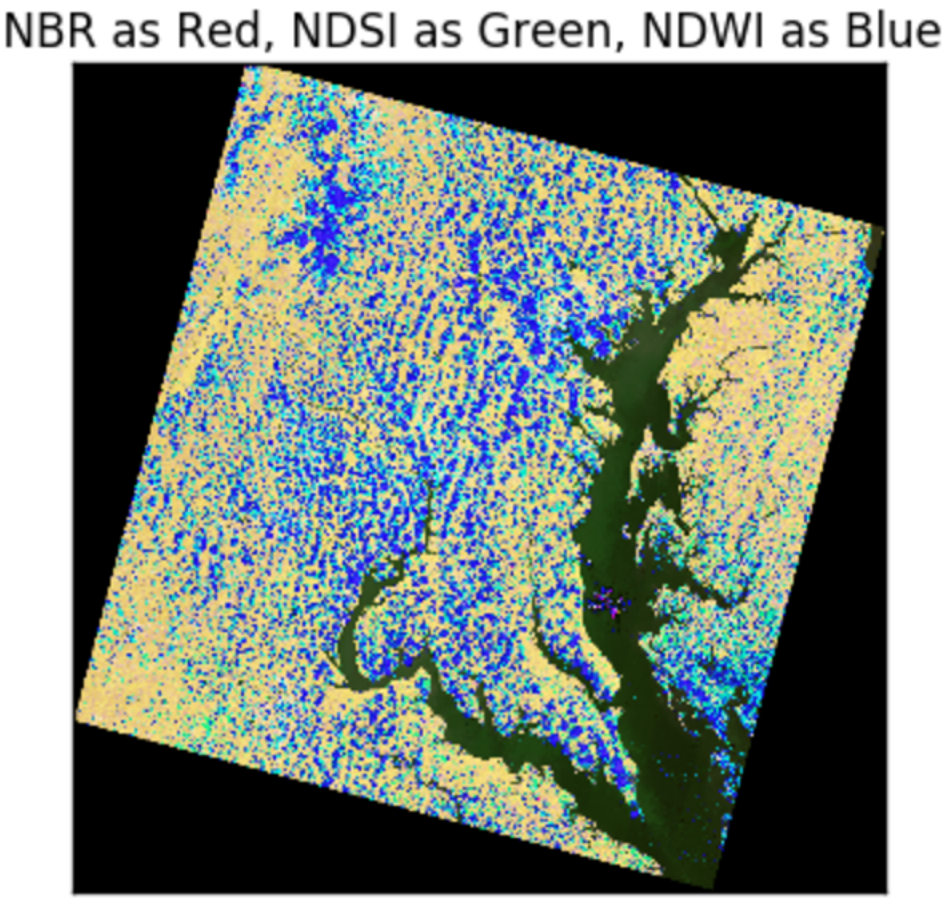
Using pcolormesh on normalized differences of bands

Here are the `NDWI` and `NDVI` plotted with the **'xarray-pcolormesh'_** method of the `predict DataArray`


NDWI
0.60
0.45
0.30
0.15
0.00
−0.15
−0.30
−0.45
−0.60
ndwi
NDVI
0.60
0.45
0.30
0.15
0.00
−0.15
−0.30
−0.45
−0.60
ndvi

False Color with normalized differences of bands

The image below has an RGB (red, green, blue) matrix made up of the `NBR` , `NDSI` , `NDWI` normalized differences:



5.14 Normalization and Adding Polynomial Terms

The following snippets show how to use a class from `sklearn.preprocessing` or `sklearn.feature_selection` with `Pipeline` :

Feature engineering - Standardize with means / standard deviations per band

Any of the scalers from `sklearn.preprocessing` may be used like this, such as `steps.MinMaxScaler`.

```
In [22]: standardize = steps.StandardScaler()
```

Feature engineering - Add polynomial terms

The classes in `sklearn.preprocessing`, such as `PolynomialFeatures`, are all available in `elm.pipeline.steps`. Keyword arguments are passed to the underlying `scikit-learn` method.

```
In [23]: poly = steps.PolynomialFeatures(interaction_only=True)
```

Custom Feature Selection

By defining the function below, we will be able to choose among random combinations of the original data or normalized differences

Feature engineering - Random selector of bands / normalized diffs

In [24]:
```
DEFAULT_BANDS = tuple('band_{}'.format(idx) for idx in range(1, 8))
def choose_bands(X, y=None, sample_weight=None, **kwargs):
    new = {}
    bands = kwargs.get('bands', DEFAULT_BANDS)
    include_normed_diffs = kwargs.get('include_normed_diffs', True)
    for band in bands:
        new[band] = getattr(X, band)
    if include_normed_diffs:
        for diff in normalized_diffs:
            new[diff] = getattr(X, diff)
    ks = list(new)
    np.random.shuffle(ks)
    es = ElmStore({k: new[k] for k in ks[:kwargs.get('num_choices', 10)]}, add_canvas=False)
    print('Chose', es.data_vars)
    return (es, y, sample_weight)
```

In [25]:
```
choose_bands_step = steps.ModifySample(func=choose_bands,
                                       bands=DEFAULT_BANDS,
                                       num_choices=10,
                                       include_normed_diffs=True)
```

5.15 PCA

Use `steps.Transform` to wrap `PCA` or another method from `sklearn.decomposition` for `elm.pipeline.Pipeline`.

PCA

Use `steps.Transform` to wrap a scikit-learn transform model, typically a transformer from `sklearn.decomposition` like `PCA`.

In [26]:
```
pca = steps.Transform(PCA())
```

Read more on sklearn.decomposition models here.

5.16 Use an estimator from `scikit-learn`

Use a model with a `fit` / `predict` interface, such as `KMeans`.

Use an estimator from scikit-learn

The final step in the `Pipeline` may be any estimator from scikit-learn with a `fit` / `predict` interface.

In [27]:
```
kmeans = MiniBatchKMeans(n_clusters=5)
```

Most scikit-learn models described here are supported.

5.17 Create `Pipeline` instance

The following uses all the steps we have created in sequence of tuples and configures scoring for K-Means with the Akaike Information Criterion.

`Pipeline` initialization with steps and scoring

Now we have all of the steps we need for our transformers and final estimator K-Means.

To create a `Pipeline` we need to:

- Put each of steps we have defined into a tuple where the first item in the tuple is a label for the step (labels may be used elsewhere for modifying parameters)
- Give a model `scoring` function and `scoring_kwargs`

The `Pipeline` will automate the series of `fit_transform` calls, analogous to what was done several cells above for adding normalized differences to `Xnew`.

In [28]:
```
from elm.model_selection.kmeans import kmeans_aic, kmeans_model_averaging
pipe = Pipeline([('set_nans', set_nans_step),
                 ('normed_diffs', normed_diffs_step),
                 ('choose', choose_bands_step),
                 ('flat', flat),
                 ('drop_na', drop_na),
                 ('log_or_not', log_transform_or_not),
                 ('standard', standardize),
                 ('poly', poly),
                 ('pca', pca),
                 ('kmeans', kmeans)],
                scoring=kmeans_aic,
                scoring_kwargs=dict(score_weights=[-1]))
```

The next steps deal with controlling *fit_ensemble* (fitting with a group of models of different parameters)

See more info on *Pipeline here*.

5.18 `ensemble_init_func`

This is an example `ensemble_init_func` to pass to *fit_ensemble*, using `pipe.new_with_params(**new_params)` to create a new unfitted `Pipeline` instance with new parameters.

In [36]:
```
from elm.model_selection.kmeans import kmeans_aic, kmeans_model_averaging
def ensemble_init_func(pipe, **kwargs):
    '''Initialize Random Pipeline Instances
       Vary N of components, N of clusters, poly terms

    Parameters:
        pipe: a Pipeline instance
        kwargs: Not used here
    Returns:
        List of Pipeline instances with varying parameters
    '''
    models = []
    for repeat in range(36):
        # Do random choices of parameters with some contraints
        normed_diffs = np.random.choice((True,) * 3 + (False,))
        bands = np.random.choice((DEFAULT_BANDS, DEFAULT_BANDS[1:-1], [], [],))
        if not bands:
            normed_diffs = True
        num_choices = np.random.randint(8, 12)
        n_clusters = np.random.choice(range(7, 18))
        if not bands:
            n_components = np.random.choice((3, 4))
        else:
            n_components = np.random.choice(range(2, num_choices - 1))
        degree = np.random.choice((1, 1, 1, 2))
        do_log = np.random.choice((True, False))
        # Make a parameters dict
        # using the parameters naming shown above in .get_params()
        params = dict(choose__include_normed_diffs=normed_diffs,
                      choose__bands=bands,
                      choose__num_choices=num_choices,
                      kmeans__n_clusters=n_clusters,
                      pca__n_components=n_components,
                      log_or_not__kw_args={'do_log': do_log},
                      poly__degree=degree)
        # Create a new Pipeline instance with new parameters (unfitted)
        new = pipe.new_with_params(**params)
        models.append(new)
    return models  # return a list of Pipeline instances
```

The *fit_ensemble docs* also show an example of an `ensemble_init_func`.

5.19 More `fit_ensemble` control

The following sets the number of generations (`ngen`) and the `model_selection` callable after each generation.

In [37]:
```
ensemble_kwargs = {
    'model_selection': kmeans_model_averaging,
    'model_selection_kwargs': {'evolve_n': 12, 'drop_n': 12},
    'ensemble_init_func': ensemble_init_func, # the function above
    'ngen': 3,
}
```

5.20 Parallelism with `dask-distributed`

fit_ensemble , to fit a group of models in generations with model selection after each generation, and *predict_many* each take a `client` keyword as a dask `Client` (dask). *predict_many* parallelizes over multiple models and samples, though here only one sample is used.

In [ ]:
```
from elm.config import client_context
with client_context() as client:
    print("FIT")
    pipe.fit_ensemble(X=X, client=client, **ensemble_kwargs)
    print("PREDICT")
    preds = pipe.predict_many(X=X, client=client)
    print("OK")
```

In [40]:
```
tag, best = pipe.ensemble[0]
best
```

Out[40]:
```
<elm.pipeline.Pipeline> with steps:
set_nans: <elm.steps.ModifySample>:
        func: <function set_nans at 0x11b74d840>
normed_diffs: <elm.steps.ModifySample>:
        func: <function add_band_ratios at 0x11b74da60>
        ratios: {'nbr': ('band_4', 'band_7'), 'ndvi': ('band_5', 'band_4'), 'ndsi': ('band_
2', 'band_6'), 'ndwi': ('band_4', 'band_5')}
choose: <elm.steps.ModifySample>:
        bands: ('band_1', 'band_2', 'band_3', 'band_4', 'band_5', 'band_6', 'band_7')
        func: <function choose_bands at 0x11b767510>
        include_normed_diffs: False
        num_choices: 11
flat: <elm.steps.Flatten>:

drop_na: <elm.steps.DropNaRows>:

log_or_not: <elm.steps.FunctionTransformer>:
        accept_sparse: False
```

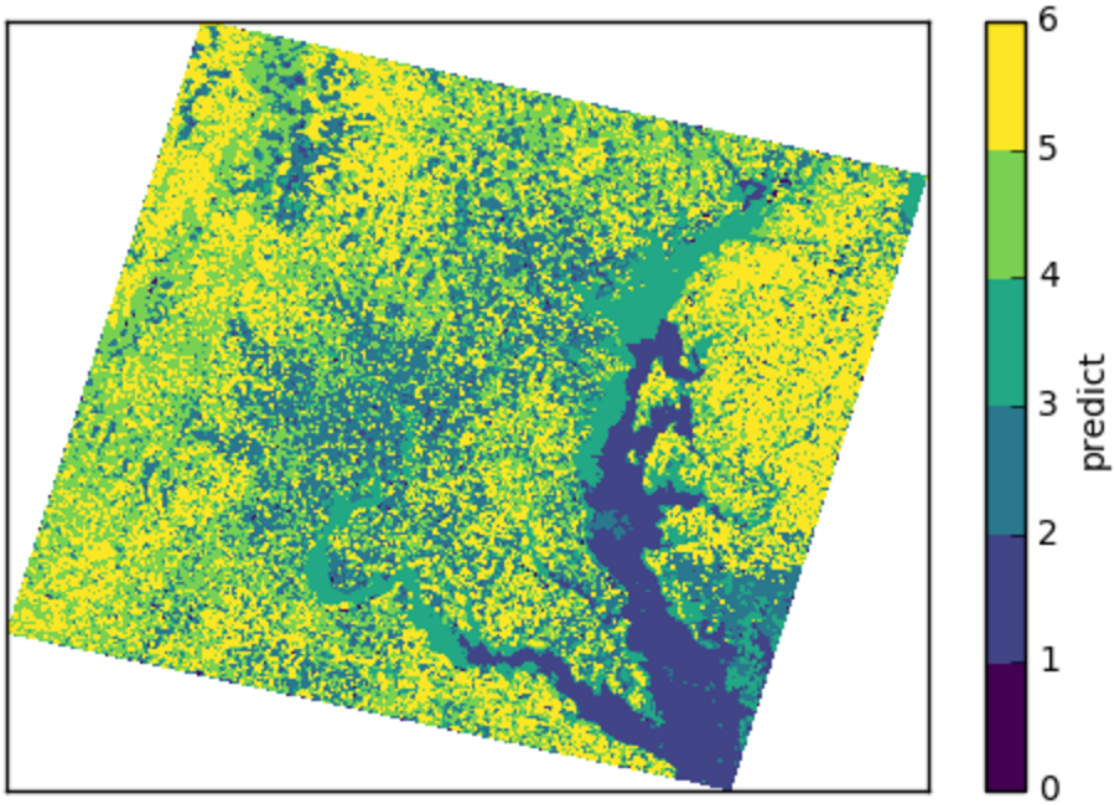
5.21 Using an `ElmStore` from predict_many

predict_many has called transform-inverseflatten to reshape the 1-D numpy array from the `sklearn.cluster.KMeans.predict` method to a 2-D raster with the coordinates of the original data. Note also the `inverse_flatten` is typically able to preserve `NaN` regions of the original data (the `NaN` borders of this image are preserved).

Using the xarray's pcolormesh on the `predict` attribute (`DataArray`) of an `ElmStore` returned by *predict_many* :

In [43]:
```
TOP_N = 5 #  Plot only the top 5 models in terms of AIC score
          # (see the scoring arg to Pipeline above)
import matplotlib.pyplot as plt
for p in preds[:TOP_N]:
    # avoid interpolating the classes
    levels = np.unique(p.predict.values[~np.isnan(p.predict.values)])
    p.predict.plot.contourf(levels=levels)
    no_axis_labels()
    plt.show()
```

The best prediction in terms of `AIC` :


6
5
4
3
2
1
0
predict

CHAPTER 6

Examples

This page provides examples of Python sessions using `elm` code and `yaml` config files that can be run with *elm-main*.

6.1 Prerequisites for Examples

Follow the instructions for installation of `elm` and the `elm-env` conda environment (*Install*):

- https://github.com/ContinuumIO/elm/tree/master/examples
- https://github.com/ContinuumIO/elm-data/

Also, define the environment variable `ELM_EXAMPLE_DATA_PATH` to be your full path to local clone of `elm-data`

6.2 Jupyter (IPython) Notebooks with `elm`

Two notebooks provide worked out examples of using `elm` with time series of spatial weather data from NetCDF files.

The notebook then goes through how to build true and false color images with an *ElmStore* and `matplotlib.pyplot.imshow`

6.3 Notebooks using `elm`

- Clustering of temperature probability distributions in time
- Land cover clustering with K-Means, PCA, and other transformations

6.4 `yaml` config files for `elm-main`

- **'Examples with SGD and K-Means in Elm examples'_**

User Guide

- *ElmStore*
- *Pipeline*
- *Customizable Pipeline Steps*
- *Multi-Model Fitting I: Ensemble Fitting*
- *Multi-Model Fitting II: Evolutionary Algorithms*
- *Multi-Model Prediction*
- *elm yaml Specs - Deprecated Temporarily*
- *elm-main Entry Point Deprecated Temporarily*

CHAPTER 7

ElmStore

`ElmStore`, from `earthio`, is a fundamental data structure in `elm` and is the data structure used to pass arrays and metadata through each of the steps in an *Pipeline* (series of transformations). An `ElmStore` is oriented around multi-band rasters and cubes stored in HDF4 / 5, NetCDF, or GeoTiff formats. `ElmStore` is a light wrapper around `xarray.Dataset`.

This page discusses:

- *Creating an ElmStore from File*
- *Creating an ElmStore - Contructor*
- *Attributes of an ElmStore*
- *Common ElmStore Transformations*

See also *API docs*.

7.1 Creating an `ElmStore` from File

An `ElmStore` can be created from `HDF4` / `HDF5` or `NetCDF` file with `load_array` from `earthio`. The simple case is to load all bands or subdatasets from an HDF or NetCDF file:

```
from earthio import load_array
filename = '3B-HHR-E.MS.MRG.3IMERG.20160708-S153000-E155959.0930.V03E.HDF5.nc'
es = load_array(filename)
```

For GeoTiffs the argument is a directory name rather than a file name and each band is formed from individual GeoTiff files in the directory. The following is an example with LANDSAT GeoTiffs for bands 1 through 7:

```
In [1]: from earthio import LayerSpec, load_array

In [2]: ls
LC80150332013207LGN00_B1.TIF  LC80150332013207LGN00_B5.TIF
LC80150332013207LGN00_B2.TIF  LC80150332013207LGN00_B6.TIF
```

(continues on next page)

(continued from previous page)

```
LC80150332013207LGN00_B3.TIF  LC80150332013207LGN00_B7.TIF
LC80150332013207LGN00_B4.TIF  logfile.txt

In [3]: es = load_array('.')
In [4]: es.data_vars
Out[4]:
Data variables:
    band_0   (y, x) uint16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
    band_1   (y, x) uint16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
    band_2   (y, x) uint16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
    band_3   (y, x) uint16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
    band_4   (y, x) uint16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
    band_5   (y, x) uint16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
    band_6   (y, x
```

The example above for GeoTiffs loaded the correct bands, but labeled them in a way that may be confusing downstream in the analysis. The following section shows how to control which bands are loaded and what they are named.

7.2 Controlling Which Bands Are Loaded

Use the `band_specs` keyword to `load_array` to

- Control which subdatasets, or bands typically, are loaded into the `ElmStore` and/or
- To standardize the names of the bands (`DataArrays`) in the `ElmStore`.

The `band_specs` work slightly differently for each file type:

- HDF4 / HDF5: The `band_specs` determine matching against one of the HDF4 file's subdatasets (see also GDAL subdatasets).
- NetCDF: The `band_specs` determine matching against one of the NetCDF file's `variables` metadata (NetCDF4 python variables interface).
- GeoTiff: When calling `load_array` for GeoTiffs, the argument is a directory (of GeoTiff files) not a single GeoTiff file. The `band_specs` for a GeoTiff file determine matching based on the gdal metadata for each GeoTiff in the directory. GeoTiffs are read using `rasterio` , a wrapper around GDAl.

In simple cases `band_specs` can be a list of strings to match a `NetCDF` variable name, subdataset name, or GeoTiff file name, as shown below:

```
In [4]: from earthio import load_array
In [5]: filename = '3B-HHR-E.MS.MRG.3IMERG.20160708-S153000-E155959.0930.V03E.HDF5.nc'
In [6]: es = load_array(filename, band_specs=['HQobservationTime'])
In [7]: es.data_vars
Out[7]:
Data variables:
    HQobservationTime  (lon, lat) timedelta64[ns] NaT NaT NaT NaT NaT NaT ...
```

With GeoTiffs, giving a list of strings as `band_specs` finds matching GeoTiff files (bands) by testing if each string is `in` a GeoTiff file name of the directory. Here is an example:

```
from earthio import load_array
dir_of_tifs = '.'
load_array(dir_of_tifs, band_specs=["B1.TIF", "B2.TIF","B3.TIF"])
```

band_specs can be given as a list of earthio.LayerSpec objects. The following shows an example of loading 4 bands from an HDF4 file where the band name, such as "Band 1 " is found in the long_name key/value of the subdataset (band) metadata and the band names are standardized to lower case with no spaces.

```
In [1]: from earthio import LayerSpec, load_array

In [2]: band_specs = list(map(lambda x: LayerSpec(**x),
   [{'search_key': 'long_name', 'search_value': "Band 1 ", 'name': 'band_1'},
   {'search_key': 'long_name', 'search_value': "Band 2 ", 'name': 'band_2'},
   {'search_key': 'long_name', 'search_value': "Band 3 ", 'name': 'band_3'},
   {'search_key': 'long_name', 'search_value': "Band 4 ", 'name': 'band_4'}]))

In [3]: filename = 'NPP_DSRF1KD_L2GD.A2015017.h09v05.C1_03001.2015018132754.hdf'

In [4]: es = load_array(filename, band_specs=band_specs)

In [5]: es.data_vars
Out[5]:
Data variables:
    band_1   (y, x) uint16 877 877 767 659 920 935 935 918 957 989 989 789 ...
    band_2   (y, x) uint16 899 899 770 659 954 973 973 935 994 1004 1004 841 ...
    band_3   (y, x) uint16 1023 1023 880 781 1115 1141 1141 1082 1155 1154 ...
    band_4   (y, x) uint16 1258 1258 1100 1009 1374 1423 1423 1341 1408 1405 ...
```

Note the LayerSpec objects could have also used the keyword arguments key_re_flags and value_re_flags with a list of flags passed to *re* for regular expression matching.

7.3 LayerSpec - File Reading Control

Here are a few more things a LayerSpec can do:

- A LayerSpec can control the resolution at which a file is read (and improve loading speed). To control resolution when loading rasters, provide buf_xsize and buf_ysize keyword arguments (integers) to LayerSpec.
- A LayerSpec can provide a window that subsets the file. See this rasterio demo that shows how window is effectively interpreted in load_array.
- A LayerSpec with a meta_to_geotransform callable attribute can be used to construct a geo_transform array from band metadata (e.g. when GDAL fails to detect the geo_transform accurately)
- A LayerSpec can control whether a raster is loaded with *("y", "x")* pixel order (the default behavior that suits most top-left-corner based rasters) or *("x", "y")* pixel order.

See also the definition of LayerSpec in earthio showing all the recognized fields (snippet taken from earthio.util).

```
@attr.s
class LayerSpec(object):
    search_key = attr.ib()
    search_value = attr.ib()
    name = attr.ib()
    key_re_flags = attr.ib(default=None)
    value_re_flags = attr.ib(default=None)
    buf_xsize = attr.ib(default=None)
    buf_ysize = attr.ib(default=None)
    window = attr.ib(default=None)
```

(continues on next page)

(continued from previous page)

```
    meta_to_geotransform = attr.ib(default=None)
    stored_coords_order = attr.ib(default=('y', 'x'))
```

7.4 Creating an `ElmStore` - Contructor

Here is an example of creating an `ElmStore` from `numpy` arrays and `xarray.DataArrays`. In most ways, an `ElmStore` is interchangeable with an `xarray.Dataset` (see also docs on working with a Dataset.

```
from collections import OrderedDict
import numpy as np
import xarray as xr
from earthio import ElmStore

rand_array = lambda: np.random.normal(0, 1, 1000000).reshape(-1,10)

def sampler(**kwargs):
    bands = ['b1', 'b2', 'b3', 'b4']
    es_data = OrderedDict()
    for band in bands:
        arr = rand_array()
        y = np.arange(arr.shape[0])
        x = np.arange(arr.shape[1])
        es_data[band] = xr.DataArray(arr, coords=[('y', y), ('x', x)], dims=('y', 'x
↪'), attrs={})
    return ElmStore(es_data, add_canvas=False)
```

Calling `sampler` above gives:

```
<elm.ElmStore>
Dimensions:  (x: 10, y: 100000)
Coordinates:
  * y        (y) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
  * x        (x) int64 0 1 2 3 4 5 6 7 8 9
Data variables:
    b1       (y, x) float64 1.772 -0.414 1.37 2.107 -1.306 0.9612 -0.0696 ...
    b2       (y, x) float64 0.07442 1.908 0.5816 0.06838 -2.712 0.4544 ...
    b3       (y, x) float64 -2.597 -1.893 0.05608 -0.5394 1.406 -0.6185 ...
    b4       (y, x) float64 1.054 -1.522 -0.03202 -0.02127 0.02914 -0.6757 ...
Attributes:
    _dummy_canvas: True
    band_order: ['b1', 'b2', 'b3', 'b4']
```

`ElmStore` has the initialization keyword argument `add_canvas` that differs from `xarray.Dataset`. If `add_canvas` is True (default), it expected that the band metadata in the `DataArrays` each contain a `geo_transform` key with a value that is a sequence of length 6. See the GDAL data model for more information on geo transforms. In the example above each `DataArray` did not have a `geo_transform` in `attrs` so `add_canvas` was set to `False`. The limitation of not having a `canvas` attribute is inability to use some spatial reindexing transformations (e.g. `elm.pipeline.steps.SelectCanvas` described further below)

7.5 Attributes of an ElmStore

If an ElmStore was initialized with add_canvas (the behavior in load_array), then it is expected each band, or DataArray, will have a geo_transform in its metadata. The geo_transform information, in combination with the array dimensions and shape, create the ElmStore's canvas attribute.

```
In [4]: es.canvas

Out[5]: Canvas(geo_transform=(-180.0, 0.1, 0, -90.0, 0, 0.1), buf_xsize=3600, buf_
↪ysize=1800, dims=('lon', 'lat'), ravel_order='C', zbounds=None, tbounds=None,␣
↪zsize=None, tsize=None, bounds=BoundingBox(left=-180.0, bottom=-90.0, right=179.
↪90000000000003, top=89.9))
```

The canvas is used in the Pipeline for transformations like elm.pipeline.steps.SelectCanvas which can be used to reindex all bands onto coordinates of one of the band's in the ElmStore.

An ElmStore has a data_vars attribute (inherited from xarray.Dataset - described here), and also has an attribute band_order. When elm.pipeline.steps.Flatten flattens the separate bands of an ElmStore, band_order becomes the order of the bands in the single flattened 2-D array.

```
In [5]: filename = '3B-MO.MS.MRG.3IMERG.20160101-S000000-E235959.01.V03D.HDF5'
In [6]: es = load_array(filename)
In [7]: es.data_vars
Out[7]:
Data variables:
    band_0    (y, x) int16 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
    band_1    (y, x) float32 -9999.9 -9999.9 -9999.9 -9999.9 -9999.9 -9999.9 ...
    band_2    (y, x) int16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
    band_3    (y, x) float32 -9999.9 -9999.9 -9999.9 -9999.9 -9999.9 -9999.9 ...

In [8]: es.band_order
Out[8]: ['band_0', 'band_1', 'band_2', 'band_3']
```

7.6 Common ElmStore Transformations

Flatten

elm.pipeline.steps.Flatten will convert an ElmStore of 2-D rasters in bands (each band as a DataArray) to an ElmStore with a single DataArray called flat. *Note: elm.pipeline.steps.Flatten() must be included in a Pipeline before scikit-learn based transforms on an ElmStore, where the scikit-learn transforms expect a 2-D array.

Here is an example of Flatten that continues the example above that defined sampler, a function returning a random ElmStore of 2-D DataArray bands:

```
es = sampler()
X_2d, y, sample_weight = steps.Flatten().fit_transform(es)

In [17]: X_2d.flat
Out[17]:
<xarray.DataArray 'flat' (space: 1000000, band: 4)>
array([[ 1.13465339, -0.1533531 ,  1.72809878, -0.7746218 ],
       [-0.12378515, -1.72588715,  0.07752273, -1.19004227],
       [ 2.16456385, -0.58083733,  0.03706811,  0.26274225],
       ...,
```

(continues on next page)

(continued from previous page)

```
       [ 0.45586256, -1.87248571,  1.27793313,  0.19892153],
       [ 2.11702651, -0.05300853, -0.92923591, -1.07152977],
       [-0.10245425, -1.27150399, -1.48745754,  1.00873062]])
Coordinates:
  * space    (space) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * band     (band) <U2 'b1' 'b2' 'b3' 'b4'
Attributes:
    old_dims: [('y', 'x'), ('y', 'x'), ('y', 'x'), ('y', 'x')]
    _dummy_canvas: True
    canvas: Canvas(geo_transform=(-180, 0.1, 0, 90, 0, -0.1), buf_xsize=10, buf_
↪ysize=100000, dims=('y', 'x'), ravel_order='C', zbounds=None, tbounds=None,␣
↪zsize=None, tsize=None, bounds=BoundingBox(left=-180.0, bottom=90.0, right=-179.1,␣
↪top=-9909.900000000001))
    old_canvases: [Canvas(geo_transform=(-180, 0.1, 0, 90, 0, -0.1), buf_xsize=10,␣
↪buf_ysize=100000, dims=('y', 'x'), ravel_order='C', zbounds=None, tbounds=None,␣
↪zsize=None, tsize=None, bounds=BoundingBox(left=-180.0, bottom=90.0, right=-179.1,␣
↪top=-9909.900000000001)), Canvas(geo_transform=(-180, 0.1, 0, 90, 0, -0.1), buf_
↪xsize=10, buf_ysize=100000, dims=('y', 'x'), ravel_order='C', zbounds=None,␣
↪tbounds=None, zsize=None, tsize=None, bounds=BoundingBox(left=-180.0, bottom=90.0,␣
↪right=-179.1, top=-9909.900000...
    flatten_data_array: True
    band_order: ['b1', 'b2', 'b3', 'b4']
```

InverseFlatten

`elm.pipeline.steps.InverseFlatten` converts an `ElmStore` that is flattened (typically the output of transform-flatten above) back to separate 2-D raster bands.

```
es = sampler()
X_2d, y, sample_weight = steps.Flatten().fit_transform(es)
restored, _, _ = steps.InverseFlatten().fit_transform(X_2d)
np.all(restored.b1.values == es.b1.values)
```

DropNaRows

`elm.pipeline.steps.DropNaRows` is a transformer that will drop any null rows from an `ElmStore` that has a `DataArray` called `flat` (see transform-flatten). It drops the null rows while keeping metadata to allow transform-inverseflatten in *predict_many* . An example usage of `DropNaRows` is given in the K-Means LANDSAT ‘‘elm‘ introduction<cluster_example>‘

Here is an example of using `DropNaRows` with the `sampler` function defined above.

```
es = sampler()
X_2d, _, _ = steps.Flatten().fit_transform(es)
X_2d.flat.values[:2, :] = np.NaN
X_no_na, _, _ = steps.DropNaRows().fit_transform(X_2d)
assert X_no_na.flat.shape[0] == X_2d.flat.shape[0] - 2
restored = inverse_flatten(X_no_na)
assert restored.b1.shape == es.b1.shape
val = restored.b1.values
assert val[np.isnan(val)].size == 2
```

Agg

Aggregation along a dimension can be done with `elm.pipeline.steps.Agg`, referencing either a `dim` or `axis` .

```
In [44]: es = sampler()

In [45]: agged, _, _ = steps.Agg(dim='y', func='median').fit_transform(es)

In [46]: agged
Out[46]:
ElmStore:
<elm.ElmStore>
Dimensions:  (x: 10)
Coordinates:
  * x        (x) int64 0 1 2 3 4 5 6 7 8 9
Data variables:
    b1       (x) float64 -0.00231 -0.00294 -0.002797 0.002472 -0.006088 ...
    b2       (x) float64 8.965e-06 0.0001929 -0.007133 0.001447 -0.001846 ...
    b3       (x) float64 -0.0009686 -0.003632 -0.0007322 -0.002221 -0.0039 ...
    b4       (x) float64 0.00667 0.001018 0.002702 0.009274 0.001481 ...
Attributes:
    _dummy_canvas: True
    band_order: ['b1', 'b2', 'b3', 'b4']
```

In the example above, `median` could have been replaced by any of the following:

- `all`
- `any`
- `argmax`
- `argmin`
- `max`
- `mean`
- `median`
- `min`
- `prod`
- `sum`
- `std`
- `var`

Read more on the implementation of the functions above in the xarray.DataArray methods here.

7.7 `ElmStore` and Metadata

This section describes `elm` functions useful for deriving information from file metadata.

set_na_from_meta: This function searches the `attrs` of each `DataArray` in an `ElmStore` or `xarray.Dataset` and sets `NaN` values in each `DataArray` where metadata indicates it is necessary. Currently `set_na_from_meta` searches `attrs` for the following keys using a case-, space- and punctuation-insenstive regular expression:

- `missing_value`: Any values in the `DataArray` equal to the missing value will be set to `NaN`.
- `valid_range` and `invalid_range`: If `attrs` have a key like `valid_range` or `invalid_range`, the function will check to see if it is a sequence of length 2 or a string that can be split on comma or spaces to form

a sequnce of length 2. If a sequence of length 2, then the invalid / valid ranges will be used to set `NaN` values appropriately.

```
from earthio.tests.util import HDF4_FILES
from earthio import load_array, set_na_from_meta
es = load_array(HDF4_FILES[0])
set_na_from_meta(es) # modifies ElmStore instance in place
```

meta_is_day: This function takes a single argument, a dict that is typically the `attrs` of an `ElmStore`, and searches for keys/values indicating whether the `attrs` correspond to a day or night sample.

```
from earthio.tests.util import HDF4_FILES
from earthio import load_array
from earthio.metadata_selection import example_meta_is_day
from scipy.stats import describe
es3 = load_array(HDF4_FILES[0])
es3.DayNightFlag # prints "Day"
meta_is_day(es3) # prints True
```

CHAPTER 8

Pipeline

8.1 Overview of `Pipeline` in `elm`

`elm.pipeline.Pipeline` allows a sequence of transformations on samples before fitting, transforming, and/or predicting from an scikit-learn estimator. `elm.pipeline.Pipeline` is similar to the concept of the `Pipeline` in scikit-learn (`sklearn.pipeline.Pipeline`) but differs in several ways described below.

- **Data sources for a Pipeline**: In `elm`, the fitting expects `X` to be an *ElmStore* or xarray.Dataset rather than a `numpy` array as in scikit-learn. This allows the `Pipeline` of transformations to include operations on cubes and other data structures common in satellite data machine learning.
- **Transformations**: In scikit-learn each step in a `Pipeline` passes a numpy array to the next step by way of a `fit_transform` method. In `elm`, a `Pipeline` always passes a tuple of *(X, y, sample_weight)* where *X* is an *ElmStore* or `xarray.Dataset` and `y` and `sample_weight` are numpy arrays or `None`.
- **Partial Fit for Large Samples**: In `elm` a transformer with a `partial_fit` method, such as `sklearn.decomposition.IncrementalPCA` may be partially fit several times as a step in a `Pipeline` and the final estimator may also use `partial_fit` several times with `dask-distributed` for parallelization.
- **Multi-Model / Multi-Sample Fitting: In `elm`, a `Pipeline` can be fit with:**
  - *fit_ensemble*: This method repeats model fitting over a series of samples and/or a ensemble of `Pipeline` instances. The `Pipeline` instances in the ensemble may or may not have the same initialization parameters. *fit_ensemble* can run in generations, optionally applying user-given model selection logic between generations. This *fit_ensemble* method is aimed at improved model fitting in cases where a representative sample is large and/or there is a need to account for parameter uncertainty.
  - *fit_ea*: This method uses Distributed Evolutionary Algorithms in Python (`deap`) to run a genetic algorithm, typically NSGA-2, that selects the best `Pipeline` instance(s). The interface for *fit_ea* and *fit_ensemble* are similar, but *fit_ea* takes an `evo_params` argument to configure the genetic algorithm.
- **Multi-Model / Multi-Sample Prediction**: `elm`'s `Pipeline` has a method *predict_many* that can use dask-distributed to predict from one or more `Pipeline` instances and/or one or more samples (ElmStore will predict for all models in the final ensemble output by *fit_ensemble*.

The following discusses each step of making a `Pipeline` that uses most of the features described above.

8.2 Data Sources for a `Pipeline`

`Pipeline` can be used for fitting / transforming / predicting from a single sample or series of samples. For the *fit_ensemble*, *fit_*

- To fit to a single sample, use the `X` keyword argument, and optionally `y` and `sample_weight` keyword arguments.
- To fit to a series of samples, use the `args_list` and `sampler` keyword arguments.

If `X` is given it is assumed to be an *ElmStore* or *xarray.Dataset*

If `sampler` is given with `args_list`, then each element of `args_list` is unpacked as arguments to the callable `sampler`. There is a special case of giving `sampler` as `earthio.band_selection.select_from_file` which allows using the functions from `earthio` for reading common formats and selecting bands from files (the `band_specs` argument). Here is an example that uses `select_from_file` to load multi-band `HDF4` arrays:

```
from earthio import LayerSpec
from earthio.metadata_selection import meta_is_day
band_specs = list(map(lambda x: LayerSpec(**x),
        [{'search_key': 'long_name', 'search_value': "Band 1 ", 'name': 'band_1'},
         {'search_key': 'long_name', 'search_value': "Band 2 ", 'name': 'band_2'},
         {'search_key': 'long_name', 'search_value': "Band 3 ", 'name': 'band_3'},
         {'search_key': 'long_name', 'search_value': "Band 4 ", 'name': 'band_4'},
         {'search_key': 'long_name', 'search_value': "Band 5 ", 'name': 'band_5'},
         {'search_key': 'long_name', 'search_value': "Band 6 ", 'name': 'band_6'},
         {'search_key': 'long_name', 'search_value': "Band 7 ", 'name': 'band_7'},
         {'search_key': 'long_name', 'search_value': "Band 9 ", 'name': 'band_9'},
         {'search_key': 'long_name', 'search_value': "Band 10 ", 'name': 'band_10'},
         {'search_key': 'long_name', 'search_value': "Band 11 ", 'name': 'band_11'}]))
HDF4_FILES = [f for f in glob.glob(os.path.join(ELM_EXAMPLE_DATA_PATH, 'hdf4', '*hdf
↪'))
              if meta_is_day(load_hdf4_meta(f))]
data_source = {
    'sampler': select_from_file,
    'band_specs': band_specs,
    'args_list': HDF4_FILES,
}
```

Alternatively, to train on a single HDF4 file, we could have done:

```
from earthio import load_array
from earthio.metadata_selection import example_meta_is_day
HDF4_FILES = [f for f in glob.glob(os.path.join(ELM_EXAMPLE_DATA_PATH, 'hdf4', '*hdf
↪'))
              if example_meta_is_day(load_hdf4_meta(f))]
data_source = {'X': load_array(HDF4_FILES[0], band_specs=band_specs)}
```

8.3 Transformations

A `Pipeline` is created by giving a list of steps - the steps before the final step are known as transformers and the final step is the estimator. See also the full docs on *elm.pipeline.steps*.

- Transformer steps must be taken from one of the classes in `elm.pipeline.steps`. The purpose of `elm.pipeline.steps` is to wrap preprocessors and transformers from scikit-learn for use with :doc:‘ElmStore<elm-store>‘s or ‘‘xarray.Dataset‘‘s.

Here is an example `Pipeline` of transformations before K-Means

```
from elm.pipeline import steps, Pipeline
pipeline_steps = [steps.Flatten(),
                  ('scaler', steps.StandardScaler()),
                  ('pca', steps.Transform(IncrementalPCA(n_components=4), partial_fit_
↪batches=2)),
                  ('kmeans', MiniBatchKMeans(n_clusters=4, compute_labels=True)),]
```

The example above calls:

- `steps.Flatten` first (See transformers-flatten) first, as utility for flattening our multi-band raster HDF4 sample(s) into an *ElmStore* with a single xarray.DataArray, called `flat`, with each band as a column in `flat`.
- StandardScaler with default arguments from `sklearn.prepreprocessing` (all other transformers from sklearn.preprocessing and sklearn.feature_selection are also attributes of `elm.pipeline.steps` and could be used here)
- PCA with `elm.pipeline.steps.Transform` to wrap scikit-learn transformers to allow multiple calls to `partial_fit` within a single fitting task of the final estimator - `steps.Transform` is initialized with:
  - A scikit-learn transformer as an argument
  - `partial_fit_batches` as a keyword, defaulting to 1. Note: using `partial_fit_batches != 1` requires a transformer with a `partial_fit` method
- Finally MiniBatchKMeans

8.4 Multi-Model / Multi-Sample Fitting

There are two multi-model approaches to fitting that can be used with a `Pipeline`: *fit_ensemble* or *fit_ea*. The examples above with a data source to a `Pipeline` and the transformation steps within one `Pipeline` instance work similarly in *fit_ensemble* and *fit_ea*.

Other similarities between *fit_ea* and *fit_ensemble* include the following common keyword arguments:

- `scoring` a callable with a signature like `elm.model_selection.kmeans.kmeans_aic` (See *API docs*) or a string like `f_classif` attribute name from `sklearn.metrics`
- `scoring_kwargs` kwargs passed to the `scoring` callable if needed
- `saved_ensemble_size` an integer indicating how many `Pipeline` estimators to retain in the final ensemble

Read more on controlling ensemble or evolutionary algorithm approaches to fitting:

- *fit_ensemble*
- *fit_ea*
- *Controlling Ensemble Initialization*

8.5 Multi-Model / Multi-Sample Prediction

After *fit_ensemble* or *fit_ea* has been called on a `Pipeline` instance, the instance will have the attribute `ensemble` a list of *(tag, pipeline)* tuples which are the final `Pipeline` instances selected by either of the fitting functions (see also `saved_ensemble_size` - See *Controlling Ensemble Initialization*). With a fitted `Pipeline` instance, *predict_many* can be called on the instance to predict from every ensemble member (`Pipeline` instance) on a single `X` sample or from every ensemble member and every sample if `sampler` and `args_list` are given in place of `X`.

Read more on controlling *predict_many*.

CHAPTER 9

Customizable Pipeline Steps

The examples below assume you have created a random *ElmStore* as follows:

```
from earthio.filters.make_blobs import random_elm_store
X = random_elm_store()
```

9.1 Operations to reshape an ElmStore

- `Flatten` - Flatten each 2-D `DataArray` in an *ElmStore* to create an *ElmStore* with a single `DataArray` called `flat` that is 2-D (each band raster is raveled from 2-D to a 1-D column in `flat`). Example:

```
steps.Flatten().fit_transform(X)
```

- `Agg` - Aggregate over a dimension or axis. Example:

```
steps.Agg(axis=0, func='mean').fit_transform(X)
```

- `DropNaRows` - Remove null / NaN rows from an *ElmStore* that has been through `steps.Flatten()`:

```
steps.DropNaRows().fit_transform(*steps.Flatten().fit_transform(X))
```

- `InverseFlatten` - Convert a flattened *ElmStore* back to 2-D rasters as separate `DataArray` values in an *ElmStore*. Example:

```
steps.InverseFlatten().fit_transform(*steps.Flatten().fit_transform(X)
```

9.2 Use an unsupervised feature extractor

- `Transform` - `steps.Transform` allows one to use any sklearn.decomposition method in an `elm` *Pipeline*. Partial fit of the feature extractor can be accomplished by giving `partial_fit_batches` at initialization:

```
from sklearn.decomposition import IncrementalPCA
X, y, sample_weight = steps.Flatten().fit_transform(X)
pca = steps.Transform(IncrementalPCA(),
                      partial_fit_batches=2)
pca.fit_transform(X)
```

9.3 Run a user-given callable

There are two choices for running a user-given callable in a *Pipeline* . Using `ModifySample` is the most general, taking any shape of `X`, `y` and `sample_weight` arguments, while `FunctionTransformer` requires that the *ElmStore* has been through `steps.Flatten()`

- `ModifySample` - The following shows an example function with the required signature for use with `ModifySample` . It divides all the values in each `DataArray` by their sum. Note the function always returns a tuple of `(X, y, sample_weight)` , even if `y` and `sample_weight` are not used by the function:

```
def modifier(X, y=None, sample_weight=None, **kwargs):
    for band in X.data_vars:
        arr = getattr(X, band)
        if kwargs.get('normalize'):
            arr.values /= arr.values.max()
    return X, y, sample_weight

steps.ModifySample(modifier, normalize=True).fit_transform(X)
```

- `FunctionTransformer` - Here is an example using the FunctionTransformer from `sklearn` :

```
import numpy as np
Xnew, y, sample_weight = steps.Flatten().fit_transform(X)
Xnew, y, sample_weight = steps.FunctionTransformer(func=np.log).fit_transform(Xnew)
```

9.4 Preprocessing - Scaling / Normalization

Each of the following classes from scikit-learn have been wrapped for usage as a *Pipeline* step. Each requires that the *ElmStore*

The examples below continue with `Xnew` a flattened *ElmStore* :

```
Xnew, y, sample_weight = steps.Flatten().fit_transform(X)
```

- `KernelCenterer` - See also KernelCenterer scikit-learn docs.

```
steps.KernelCenterer().fit_transform(Xnew)
```

- `MaxAbsScaler` - See also MaxAbsScaler scikit-learn docs.

```
steps.MaxAbsScaler().fit_transform(*steps.Flatten().fit_transform(X))
```

- `MinMaxScaler` - See also MinMaxScaler scikit-learn docs.

```
steps.MinMaxScaler().fit_transform(Xnew)
```

- `Normalizer` - See also Normalizer scikit-learn docs.

```
steps.Normalizer().fit_transform(Xnew)
```

- `RobustScaler` - See also RobustScaler scikit-learn docs.

```
steps.RobustScaler().fit_transform(Xnew)
```

- `PolynomialFeatures` - See also PolynomialFeatures scikit-learn docs.

```
step = steps.PolynomialFeatures(degree=3,
                                interaction_only=False)
step.fit_transform(Xnew)
```

- `StandardScaler` - See also StandardScaler scikit-learn docs.

```
steps.StandardScaler().fit_transform(Xnew)
```

9.5 Encoding Preprocessors from `sklearn`

Each method here requires that the *ElmStore* has been through `steps.Flatten()` as follows:

```
Xnew, y, sample_weight = steps.Flatten().fit_transform(X)
```

- `Binarizer` - Binarize features. See also Binarizer docs from `sklearn` .

```
steps.Binarizer().fit_transform(Xnew)
```

- `Imputer` - Impute missing values. See also Imputer docs from `sklearn` .

```
steps.Imputer().fit_transform(Xnew)
```

9.6 Feature selectors

The following list shows the feature selectors that may be used in a *Pipeline* . The methods, with the exception of `VarianceThreshold` each require `y` to be not `None`.

Setup for the examples:

```
X, y = random_elm_store(return_y=True)
X = steps.Flatten().fit_transform(X)[0]
```

- `RFE` - See also RFE in `sklearn` docs. Example:

```
steps.RFE(estimator=LinearRegression()).fit_transform(X, y)
```

- `RFECV` - See also RFECV in `sklearn` docs. Example:

```
steps.RFECV(estimator=LinearRegression()).fit_transform(X, y)
```

- `SelectFdr` - See also SelectFdr in `sklearn` docs. Example:

```
steps.SelectFdr().fit_transform(X, y)
```

- `SelectFpr` - See also SelectFpr in `sklearn` docs. Example:

```
steps.SelectFpr().fit_transform(X, y)
```

- `SelectFromModel` - See also SelectFromModel in `sklearn` docs. Example:

```
steps.SelectFromModel(estimator=LinearRegression()).fit_transform(X, y)
```

- `SelectFwe` - See also SelectFwe in `sklearn` docs. Example:

```
steps.SelectFwe().fit_transform(X, y)
```

- `SelectKBest` - See also SelectKBest in `sklearn` docs. Example:

```
steps.SelectKBest(k=2).fit_transform(X, y)
```

- `SelectPercentile` - See also SelectPercentile in `sklearn` docs. Example:

```
steps.SelectPercentile(percentile=50).fit_transform(X, y)
```

- `VarianceThreshold` - See also VarianceThreshold in `sklearn` docs. Example:

```
steps.VarianceThreshold(threshold=6.92).fit_transform(X)
```

9.7 Normalizing time dimension of 3-D Cube

The following two functions take an *ElmStore* with a `DataArray` of any name that is a 3-D cube with a time dimension. The functions run descriptive stats along the time dimension and flatten the spatial `(x, y)` dims to *space* (essentially a `ravel` of the `(x, y)` points).

Setup - make a compatible *ElmStore*:

```
from earthio import ElmStore
import numpy as np
import xarray as xr
def make_3d():
    arr = np.random.uniform(0, 1, 100000).reshape(100, 10, 100)
    return ElmStore({'band_1': xr.DataArray(arr,
                            coords=[('time', np.arange(100)),
                                    ('x', np.arange(10)),
                                    ('y',np.arange(100))],
                            dims=('time', 'x', 'y'),
                            attrs={})}, attrs={}, add_canvas=False)
X = make_3d()
```

- `TSDescribe` - Run `scipy.stats.describe` and other stats along the time axis of a 3-D cube `DataArray`. Example:

```
s = steps.TSDescribe(band='band_1', axis=0)
Xnew, y, sample_weight = s.fit_transform(X)
Xnew.flat.band
```

The above code would show the `band` dimension of `Xnew` consists of different summary statistics, mostly from `scipy.stats.describe`:

```
<xarray.DataArray 'band' (band: 8)>
array(['var', 'skew', 'kurt', 'min', 'max', 'median', 'std', 'np_skew'],
      dtype='<U7')
Coordinates:
  * band     (band) <U7 'var' 'skew' 'kurt' 'min' 'max' 'median' 'std' 'np_skew'
```

- `TSProbs` - `TSProbs` will run bin, count and return probabilities associated with bin counts. An example:

```
fixed_bins = steps.TSProbs(band='band_1',
                           bin_size=0.5,
                           num_bins=152,
                           log_probs=True,
                           axis=0)
Xnew, y, sample_weight = fixed_bins.fit_transform(X)
```

The above would create the `DataArray` `Xnew.flat` with 152 columns consisting of the `log` transformed bin probabilities (152 bins of 0.5 width).

And the following would use irregular (`numpy.histogram`) bins rather than fixed bins and return probabilities without `log` transform first:

```
irregular_bins = steps.TSProbs(band='band_1',
                               num_bins=152,
                               log_probs=False,
                               axis=0)
Xnew, y, sample_weight = irregular_bins.fit_transform(X)
```

CHAPTER 10

Multi-Model Fitting I: Ensemble Fitting

Ensemble fitting may be helpful in cases where the representative sample is large and/or model parameter or fitting uncertainty should be considered.

Ensemble fitting may:

- Use one or more samples,
- Use one or more models (*Pipeline* instances), and/or
- Use one or more generations of fitting, with model selection logic on each generation

It is helpful to first read the section Data Sources for a *Pipeline* showing how to use either a single `X` matrix or a series of samples from a `sampler` callable.

10.1 Define a Sampler

The example below uses a `sampler` function and `args_list` (list of unpackable args to `sampler`) to fit to many samples. The full script can be found here. First the script does some imports and sets up a `sampler` function that uses `band_specs` (see also *ElmStore*) to select a subset of bands in `HDF4` files.

```
import os
from sklearn.cluster import MiniBatchKMeans
from sklearn.decomposition import IncrementalPCA
import numpy as np
from elm.config.dask_settings import client_context
from elm.model_selection.kmeans import kmeans_model_averaging, kmeans_aic
from elm.pipeline import steps, Pipeline
from earthio import *
from earthio.filters.band_selection import select_from_file
from earthio.metadata_selection import example_meta_is_day

ELM_EXAMPLE_DATA_PATH = os.environ['ELM_EXAMPLE_DATA_PATH']

band_specs = list(map(lambda x: LayerSpec(**x),
```

(continues on next page)

(continued from previous page)

```
        [{'search_key': 'long_name', 'search_value': "Band 1 ", 'name': 'band_1'},
         {'search_key': 'long_name', 'search_value': "Band 2 ", 'name': 'band_2'},
         {'search_key': 'long_name', 'search_value': "Band 3 ", 'name': 'band_3'},
         {'search_key': 'long_name', 'search_value': "Band 4 ", 'name': 'band_4'},
         {'search_key': 'long_name', 'search_value': "Band 5 ", 'name': 'band_5'},
         {'search_key': 'long_name', 'search_value': "Band 6 ", 'name': 'band_6'},
         {'search_key': 'long_name', 'search_value': "Band 7 ", 'name': 'band_7'}]))
# Just get daytime files
HDF4_FILES = [f for f in glob.glob(os.path.join(ELM_EXAMPLE_DATA_PATH, 'hdf4', '*hdf
↪'))
              if example_meta_is_day(load_hdf4_meta(f))]
data_source = {
    'sampler': select_from_file,
    'band_specs': band_specs,
    'args_list': HDF4_FILES,
}
```

10.2 Define `Pipeline` Steps

Next a *Pipeline* is configured that flattens the separate band rasters to a single 2-D `DataArray` (See also transform-flatten, uses standard scaling from scikit-learn, then transforms with `IncrementalPCA` with 2 `partial_fit` batches before K-Means. The *Pipeline* constructor also takes a `scoring` callable and optional `scoring_kwargs`.

```
pipeline_steps = [steps.Flatten(),
                  ('scaler', steps.StandardScaler()),
                  ('pca', steps.Transform(IncrementalPCA(n_components=4), partial_fit_
↪batches=2)),
                  ('kmeans', MiniBatchKMeans(n_clusters=4, compute_labels=True)),]
pipe = Pipeline(pipeline_steps, scoring=kmeans_aic, scoring_kwargs=dict(score_
↪weights=[-1]))
```

See the signature for kmeans_aic here to write a similar scoring function, otherwise `scoring` defaults to calling the estimator's `.score` callable or exception if `.score` is not defined.

10.3 Configure Ensemble

Now we can call `fit_ensemble` after choosing some controls on the size of the ensemble, the number of generations, and the logic for selecting models after each generation.

Here's an example:

```
ensemble_kwargs = {
    'model_selection': kmeans_model_averaging,
    'model_selection_kwargs': {
        'drop_n': 2,
        'evolve_n': 2,
    },
    'init_ensemble_size': 4,
    'ngen': 3,
    'partial_fit_batches': 2,
    'saved_ensemble_size': 4,
```

(continues on next page)

(continued from previous page)

```
    'models_share_sample': True,
}
```

In the example above:

- `ngen` sets the number of generations to 3
- There are 4 initial ensemble members (`init_ensemble_size`),
- After each generation `kmeans_model_averaging` (See *API docs*) is called on the ensemble with `model_selection_kwargs` are passed.
- There are 3 `partial_fit` batches for `MiniBatchKMeans` on every *Pipeline* instance (`partial_fit` within the `IncrementalPCA` was configured in the initialization of `steps.Transform` above)
- `models_share_sample` is set to `True` so in each generation every ensemble member is fit to the same sample, then on the next generation, every model is fit to the next sample determined by `sampler` and `args_list` in this case. If `models_share_sample` were `False`, then in each generation every ensemble member would be copied and fit to every sample, repeating the process on each generation.

10.4 Fitting with Dask-Distributed

In the snippets above, we have a `data_source` `dict` with `sampler`,‘‘band_specs‘‘ and `args_list` key / values. We can pass this with the `ensemble_kwargs` ensemble configuration to `fit_ensemble` as well as *predict_many* . The data source for *predict_many* does not necessarily have to be the same one given to `fit_ensemble` or `fit_ea`).

Note : If you want `dask-distributed` as a client, first make sure you are running a `dask-scheduler` and `dask-worker` . Read more here on dask-distributed and follow instructions in *environment variables* .

```
with client_context() as client:
    ensemble_kwargs['client'] = client
    pipe.fit_ensemble(**data_source, **ensemble_kwargs)
    pred = pipe.predict_many(client=client, **data_source)
```

Fitting with `dask` parallelizes over the ensemble members (*Pipeline* instances) and over the calls to `partial_fit` - currently transformers in the `Pipeline` are not parallelized with `dask` .

10.5 Controlling Ensemble Initialization

To initialize the ensemble with *Pipeline* instances that do not all share the same parameters (as above), we could replace `init_ensemble_size` above with `ensemble_init_func`

```
n_clusters_choices = tuple(range(4, 9))
def ensemble_init_func(pipe, **kwargs):
    models = []
    for c in n_clusters_choices:
        new_pipe = pipe.new_with_params(kmeans__n_clusters=c)
        models.append(new_pipe)
    return models
ensemble_kwargs = {
    'model_selection': kmeans_model_averaging,
    'model_selection_kwargs': {
```

(continues on next page)

(continued from previous page)

```
        'drop_n': 2,
        'evolve_n': 2,
    },
    'ensemble_init_func': ensemble_init_func,
    'ngen': 3,
    'partial_fit_batches': 2,
    'saved_ensemble_size': 4,
    'models_share_sample': True,
}
with client_context() as client:
    ensemble_kwargs['client'] = client
    pipe.fit_ensemble(**data_source, **ensemble_kwargs)
    pred = pipe.predict_many(client=client, **data_source)
```

In the example above, `Pipeline.new_with_params(kmeans__n_clusters)` uses the scikit-learn syntax for parameter modifications of named steps in a pipeline. In the initialization of *Pipeline* in the example above, the `MiniBatchMeans` step was named `kmeans`, so `kmeans__n_clusters=c` sets the `n_clusters` parameter to the K-Means step and the ensemble in this case consists of one *Pipeline* for each of `n_clusters` choices in *(4, 5, 6, 7, 8)*.

CHAPTER 11

Multi-Model Fitting II: Evolutionary Algorithms

`elm` can use an evolutionary algorithm for hyperparameterization. This involves using the `fit_ea` method of *Pipeline*. It is helpful at this point to first read about *Pipeline* and how to configure a data source for the multi-model approaches in `elm`. That page summarizes how *fit_ea* and *fit_ensemble* may be fit to a single `X` matrix (when the keyword `X` is given) or a series of samples (when `sampler` and `args_list` are given).

The example below walks through configuring an evolutionary algorithm to select the best K-Means model with preprocessing steps inclusive of standard scaling and PCA. First it sets up a sampler from HDF4 files (note the set up of a data source is the same as in *fit_ensemble*)

11.1 Example

```
import os

from sklearn.cluster import MiniBatchKMeans
from sklearn.feature_selection import SelectPercentile, f_classif
import numpy as np

from elm.config.dask_settings import client_context
from elm.model_selection.evolve import ea_setup
from elm.model_selection.kmeans import kmeans_model_averaging, kmeans_aic
from elm.pipeline import Pipeline, steps
from earthio import *
from earthio.filters.band_selection import select_from_file
from earthio.metadata_selection import example_meta_is_day

ELM_EXAMPLE_DATA_PATH = os.environ['ELM_EXAMPLE_DATA_PATH']

band_specs = list(map(lambda x: LayerSpec(**x),
        [{'search_key': 'long_name', 'search_value': "Band 1 ", 'name': 'band_1'},
         {'search_key': 'long_name', 'search_value': "Band 2 ", 'name': 'band_2'},
         {'search_key': 'long_name', 'search_value': "Band 3 ", 'name': 'band_3'},
         {'search_key': 'long_name', 'search_value': "Band 4 ", 'name': 'band_4'},
```

(continues on next page)

(continued from previous page)

```
        {'search_key': 'long_name', 'search_value': "Band 5 ", 'name': 'band_5'},
        {'search_key': 'long_name', 'search_value': "Band 6 ", 'name': 'band_6'},
        {'search_key': 'long_name', 'search_value': "Band 7 ", 'name': 'band_7'}]))
# Just get daytime files
HDF4_FILES = [f for f in glob.glob(os.path.join(ELM_EXAMPLE_DATA_PATH, 'hdf4', '*hdf
↪'))
              if example_meta_is_day(load_hdf4_meta(f))]
data_source = {
    'sampler': select_from_file,
    'band_specs': band_specs,
    'args_list': HDF4_FILES,
}
```

Next the example sets up a *Pipeline* of transformations

```
def make_example_y_data(X, y=None, sample_weight=None, **kwargs):
    fitted = MiniBatchKMeans(n_clusters=5).fit(X.flat.values)
    y = fitted.predict(X.flat.values)
    return (X, y, sample_weight)

pipeline_steps = [steps.Flatten(),
                  steps.ModifySample(make_example_y_data),
                  ('top_n', steps.SelectPercentile(percentile=80,score_func=f_
↪classif)),
                  ('kmeans', MiniBatchKMeans(n_clusters=4))]
pipeline = Pipeline(pipeline_steps, scoring=kmeans_aic, scoring_kwargs=dict(score_
↪weights=[-1]))
```

The example above uses `elm.pipeline.steps.ModifySample` to return a `y` data set corresponding to `X` `ElmStore` so that the example can show `SelectPercentile` for feature selection.

Next `evo_params` need to be called by passing a `param_grid` dict to `elm.model_selection.evolve.ea_setup`. The `param_grid` uses scikit-learn syntax for parameter replacement (i.e. a named step like "kmeans" then a double underscore then a parameter name for that step ["n_clusters"]), so this `param_grid` could potentially run models with `n_clusters` in `range(3, 10)` and `percentile` in `range(20, 100, 5)`. The `control` dict sets parameters for the evolutionary algorithm (described below).

```
param_grid =  {
    'kmeans__n_clusters': list(range(3, 10)),
    'top_n__percentile': list(range(20, 100, 5)),
    'control': {
        'select_method': 'selNSGA2',
        'crossover_method': 'cxTwoPoint',
        'mutate_method': 'mutUniformInt',
        'init_pop': 'random',
        'indpb': 0.5,
        'mutpb': 0.9,
        'cxpb':  0.3,
        'eta':   20,
        'ngen':  2,
        'mu':    4,
        'k':     4,
        'early_stop': {'abs_change': [10], 'agg': 'all'},
        # alternatively early_stop: {percent_change: [10], agg: all}
        # alternatively early_stop: {threshold: [10], agg: any}
    }
```

(continues on next page)

(continued from previous page)

```
}

evo_params = ea_setup(param_grid=param_grid,
                      param_grid_name='param_grid_example',
                      score_weights=[-1]) # minimization
```

Running with `dask` to parallelize over the individual solutions (*Pipeline* instances) and their calls to `partial_fit` .

Note : If you want `dask-distributed` as a client, first make sure you are running a `dask-scheduler` and `dask-worker` . Read more here on dask-distributed and follow instructions in *environment variables* .

```
with client_context() as client:
    fitted = pipeline.fit_ea(evo_params=evo_params,
                             client=client,
                             **data_source)
    preds = pipeline.predict_many(client=client, **data_source)
```

11.2 Reference `param_grid` - `control`

In the example above the `param_grid` has a `control` dictionary specifying parameters of the evolutionary algorithm. The `control` dict names the functions to be used for crossover, mutation, and selection, and the other arguments are passed to the those methods as needed. The following section describes each key/value of a `control` dictionary.

Note While it is possible to change the `select_method`, `crossover_method` and `mutate_method` below from the example shown, it is important to use methods that are consistent with how `fit_ea` expresses parameter choices. For each parameter in the `param_grid`, such as `kmeans__n_clusters=list(range(3, 10))`, `fit_ea` optimizes with *indices* into `kmeans__n_clusters` list, i.e. choosing among `list(range(7))`, *not* optimizing an integer parameter between 3 and 10. This allows `fit_ea` to avoid custom treatment of string, float, or integer data types in the parameters' lists of choices. If changing the `mutate_method` keep in mind that it needs to take individuals that are sequences of integers as arguments and return the same.

- **select_method**: Selection method on each generation of evolutionary algorithm. The selection method is typically `selNSGA2` but can be any `deap.tools` selection method (see the **'list of selection methods here'_**)
- **crossover_method**: Crossover method between two individuals, e.g. `cxTwoPoint`, or any crossover method from deap.tools
- **mutate_method**: Mutation method, typically `mutUniformInt`, or another mutation method from `deap.tools` mutation methods
- **init_pop**: Placeholder for initialization features- must always be `random` (random initialization of solutions)
- **indpb**: Proability each attribute (feature) is mutated when an individual is mutated, e.g. `0.5` (passed to mutation methods in `deap.tools`)
- **mutpb**: When two individuals crossover, this is the probability they will mutate immediately after crossover, e.g. `0.9`
- **cxpb**: Probabity of crossover `0.3`
- **eta**: Tuning parameter in NSGA-2 - passed to mutate and mate methods. With a higher `eta` crowding is penalized and offspring are more different from their parents
- **ngen**: Number of generations in genetic algorithm
- **mu**: Size of the population of solutions (individuals) initially

- **k**: Select the top `k` on each generation
- **early_stop**: Control stopping of algorithm before `ngen` number of generations is completed. Examples are below (note `agg` refers to aggregation as `all` or `any` in the case it is a multi-objective problem)
  - *Stop on absolute change in objective*: `{'abs_change': [10], 'agg': 'all'}`
  - *Stop on percent change in objective*: `early_stop: {percent_change: [10], agg: all}`
  - *Stop on reaching objective threshold*: `early_stop: {threshold: [10], agg: any}`

11.3 More Reading

`fit_ea` relies on `deap` for Pareto sorting and the genetic algorithm components described above. Read more about `deap`:

- deap Docs
- deap source code
- deap NSGA-2 example on which fit_ea is based

CHAPTER 12

Multi-Model Prediction

`elm`'s *predict_many* predicts for each estimator in a trained ensemble for one or more samples. *predict_many* takes some of the same data source keyword arguments that *fit_ea* and *fit_ensemble* use. See also *Data Sources for a Pipeline* - it discusses using a single sample by giving the keyword arguments `X` or giving a `sampler` and `args_list` (list of unpackable args to the `sampler` callable). The same logic applies for *predict_many*.

predict_many has a feature `to_cube` argument that is useful in prediction for spatial data. `to_cube=True` (`True` by default) means to convert the 1-D numpy array of predictions from the estimator of a *Pipeline* instance to a 2-D raster with the coordinates of the input data before the input data were flattened for training. This makes it easy to make xarray-pcolormesh plots of predictions in spatial coordinates that are derived from models trained on spatial satellite and weather data.

12.1 Example - `SGDClassifier`

The following example shows fitting a stochastic gradient descent classifier in ensemble with `partial_fit`, varying the `alpha` and `penalty` parameters to `sklearn.linear_model.SGDClassifier` and finally predicting from the best models of the ensemble over several input samples.

12.2 Import from `elm` and `sklearn`

This is a common set of `import` statements when working with `elm`

```
from collections import OrderedDict
from elm.pipeline import Pipeline, steps
from earthio import *
from sklearn.datasets import make_blobs
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score
import numpy as np
import xarray as xr
```

12.3 Define model selection

We can define a callable with a signature like `model_selection` below to control which models are passed from generation to generation in an ensemble. This function just uses `best_idxes` (Pareto sorted model fitness from the `accuracy_score`):

```
def model_selection(models, best_idxes=None, **kwargs):
    top_n = kwargs['top_n']
    return [models[idx] for idx in best_idxes[:top_n]]
```

See also `model_selection` in *Controlling Ensemble Initialization*.

12.4 Define initial ensemble

To vary the parameters of the initial ensemble of *Pipeline* instances, provide an `ensemble_init_func`. `pipe.new_with_params` is used here to create a variety of *Pipeline* objects that have different `SGDClassifier` `alpha` and `penalty` parameters.

```
def ensemble_init_func(pipe, **kwargs):
    models = []
    for penalty in ('l1', 'l2'):
        for alpha in (0.0001, 0.001, 0.01):
            new_pipe = pipe.new_with_params(sgd__penalty=penalty, sgd__alpha=alpha)
            models.append(new_pipe)
    return models
```

See also `ensemble_init_func` in *Controlling Ensemble Initialization*.

12.5 Control `partial_fit` and ensemble

The following `dict` are keywords to pass to *fit_ensemble*, including setting the number of generations `ngen`, using `partial_fit` twice per fitting of each model, and retaining finally the 2 best models (`saved_ensemble_size`). Note also that `partial_fit` requires giving the keyword argument `classes`, a sequence of all known classes, so this is passed via `method_kwargs`:

```
ensemble_kwargs = {
    'model_selection': model_selection,
    'model_selection_kwargs': {
        'top_n': 2,
    },
    'ensemble_init_func': ensemble_init_func,
    'ngen': 3,
    'partial_fit_batches': 2,
    'saved_ensemble_size': 2,
    'method_kwargs': {'classes': np.arange(5)},
    'models_share_sample': True,
}
```

See also `ensemble_kwargs` in *Controlling Ensemble Initialization*.

12.6 Define a `sampler`

The following lines of code use the synthetic data helper `make_blobs` from `sklearn.datasets` to create an `ElmStore` with 5 bands (each band is a `DataArray`)

```
rand_X_y = lambda n_samples: make_blobs(centers=[[1,2,3,4,5], [2,3,6,8,9], [3,4,5,10,
↪12]], n_samples=n_samples)
def sampler_train(width, height, **kwargs):
    X, y = rand_X_y(width * height)
    bands = ['band_{}'.format(idx + 1) for idx in range(X.shape[1])]
    es_data = OrderedDict()
    for idx, band in enumerate(bands):
        arr = xr.DataArray(X[:, idx].reshape(height, width),
                   coords=[('y', np.arange(height)),
                           ('x', np.arange(width))],
                   dims=('y', 'x'))
        es_data[band] = arr
    # No geo_transform in attrs of arr, so add_canvas = False
    es = ElmStore(es_data, add_canvas=False)
    sample_weight = None
    return es, y, sample_weight
```

Testing out `sampler_train`:

```
In [42]: X, y, _ = sampler_train(10, 12)

In [43]: X, y
Out[43]:
(ElmStore:
 <elm.ElmStore>
 Dimensions:  (x: 10, y: 12)
 Coordinates:
   * y        (y) int64 0 1 2 3 4 5 6 7 8 9 10 11
   * x        (x) int64 0 1 2 3 4 5 6 7 8 9
 Data variables:
     band_1   (y, x) float64 0.5343 -1.21 1.241 2.191 3.364 2.115 3.579 3.086 ...
     band_2   (y, x) float64 3.657 3.575 1.164 4.786 4.354 3.74 1.924 3.674 ...
     band_3   (y, x) float64 4.909 2.258 2.761 4.313 5.379 4.145 6.515 5.137 ...
     band_4   (y, x) float64 9.872 5.329 4.786 10.41 10.96 6.878 7.356 10.11 ...
     band_5   (y, x) float64 7.343 5.88 3.924 11.82 11.53 10.16 10.78 11.74 ...
 Attributes:
     _dummy_canvas: True
     band_order: ['band_1', 'band_2', 'band_3', 'band_4', 'band_5'],
 array([1, 0, 0, 2, 2, 1, 1, 2, 2, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 2, 2, 2, 0,
        0, 0, 2, 1, 0, 2, 0, 2, 2, 1, 2, 1, 2, 0, 2, 2, 0, 0, 2, 1, 1, 2, 2,
        0, 1, 2, 0, 1, 0, 1, 2, 0, 0, 0, 1, 1, 1, 2, 1, 1, 2, 2, 2, 0, 1, 1,
        2, 0, 2, 2, 1, 0, 1, 2, 1, 0, 0, 1, 1, 1, 2, 1, 0, 2, 1, 0, 1, 2, 0,
        0, 2, 1, 1, 0, 1, 2, 2, 1, 0, 2, 0, 1, 0, 1, 1, 2, 0, 0, 2, 1, 1, 1,
        2, 2, 1, 0, 2]))
```

12.7 Pipeline with scoring

The example below sets up `accuracy_score` for scoring a *Pipeline* that will flatten the sample and run `SGDClassifier`. The `scoring_kwargs` include `greater_is_better` (passed to `sklearn.model_selection.make_scorer` and `score_weights` determining whether sort models from minimum to

maximum fitness (-1) or maximum to minimum (1). Here we are maximimizing the `accuracy_score`:

```
pipe = Pipeline([steps.Flatten(),
                 ('sgd', SGDClassifier())],
                 scoring=accuracy_score,
                 scoring_kwargs=dict(greater_is_better=True, score_weights=[1]))
```

Read more *documentation here* on all the options available in `elm.pipeline.steps` .

12.8 Call fit_ensemble

Calling *fit_ensemble* with an `args_list` of length 3, we are fitting all models in the ensemble to the same sample in one generation, then proceeding to fitting all models against the next sample in the next generation. In this case we have 3 generations (`ngen` above) and 3 samples (`len(args_list)` below) and `models_share_sample=True`. Each generation will have be a different sample and all models in a generation will be fitted to that sample.

```
data_source = dict(sampler=sampler_train, args_list=[(100, 120)] * 3)
fitted = pipe.fit_ensemble(**data_source, **ensemble_kwargs)
```

12.9 Call predict_many

We currently have 2 models in the ensemble (see `saved_ensemble_size` above that set the top N models to keep) and an `args_list` that will generate 3 samples: *predict_many* will predict 6 sample - model combinations.

```
preds = pipe.predict_many(**data_source)
```

Checking the number of predictions returned:

```
In [7]: len(preds)
Out[7]: 6
```

Each item in `preds` is an `ElmStore` with a `DataArray` called `predict`. In this case that `DataArray` is a 2-D raster because we used the default keyword argument `to_raster=True` when *predict_many* was called. The next snippet shows using the *plot* attribute of the `predict DataArray`:

See also documentation on plotting with xarray

```
p = preds[0]
p.predict.plot.pcolormesh()
```

12.10 Predicting from an Ensemble Subset

By default *predict_many* will look for an attribute on the *Pipeline* instance called `.ensemble`, which is expected to be a list of `(tag, pipeline)` tuples, and predict from each trained *Pipeline* instance in `.ensemble`. Alternatively you can pass a list of `(tag, pipeline)` tuples as `ensemble` keyword argument. The example below predicts only from the best model in the ensemble (the final ensemble is sorted by model score if `scoring` was given to *Pipeline* initialization). There are 3 predictions because there are 3 samples.

```
In [16]: subset = pipe.ensemble[:1]
In [17]: preds = pipe.predict_many(ensemble=subset, **data_source)
In [18]: len(preds)
Out[18]: 3
```

12.11 Predictions Too Large For Memory

In the examples above, *predict_many* has returned a list of `ElmStore` objects. If the number of samples and/or models is large then keeping them all predictions in memory in a list is infeasible. In these cases, pass a `serialize` argument (callable) to *predict_many* to serialize prediction `ElmStore` outputs as they are generated. `serialize` should have a signature exactly like the example below:

```
import os
from sklearn.externals import joblib
def serialize(y, X, tag, elm_predict_path):
    dirr = os.path.join(elm_predict_path, tag)
    if not os.path.exists(dirr):
        os.mkdir(dirr) # assuming ELM_PREDICT_PATH in environment
    base = "_".join('{:.02f}'.format(_) for _ in sorted(X.canvas.bounds))
    joblib.dump(y, os.path.join(dirr, base + '.xr'))
    return X.canvas
preds = pipe.predict_many(ensemble=pipe.ensemble[:1], serialize=serialize,**data_
↪source)
```

In predicting over 3 samples and one model, we have created 3 `joblib` dump prediction files and returned three `Canvas` objects

```
In [27]: preds
Out[27]:
(Canvas(geo_transform=(-180, 0.1, 0, 90, 0, -0.1), buf_xsize=10, buf_ysize=10, dims=(
↪'y', 'x'), ravel_order='C', zbounds=None, tbounds=None, zsize=None, tsize=None,␣
↪bounds=BoundingBox(left=-180.0, bottom=90.0, right=-179.1, top=89.1)),
 Canvas(geo_transform=(-180, 0.1, 0, 90, 0, -0.1), buf_xsize=10, buf_ysize=10, dims=(
↪'y', 'x'), ravel_order='C', zbounds=None, tbounds=None, zsize=None, tsize=None,␣
↪bounds=BoundingBox(left=-180.0, bottom=90.0, right=-179.1, top=89.1)),
 Canvas(geo_transform=(-180, 0.1, 0, 90, 0, -0.1), buf_xsize=10, buf_ysize=10, dims=(
↪'y', 'x'), ravel_order='C', zbounds=None, tbounds=None, zsize=None, tsize=None,␣
↪bounds=BoundingBox(left=-180.0, bottom=90.0, right=-179.1, top=89.1)))
(Canvas(geo_transform=(-180, 0.1, 0, 90, 0, -0.1), buf_xsize=10, buf_ysize=10, dims=(
↪'y', 'x'), ravel_order='C', zbounds=None, tbounds=None, zsize=None, tsize=None,␣
↪bounds=BoundingBox(left=-180.0, bottom=90.0, right=-179.1, top=89.1)),
 Canvas(geo_transform=(-180, 0.1, 0, 90, 0, -0.1), buf_xsize=10, buf_ysize=10, dims=(
↪'y', 'x'), ravel_order='C', zbounds=None, tbounds=None, zsize=None, tsize=None,␣
↪bounds=BoundingBox(left=-180.0, bottom=90.0, right=-179.1, top=89.1)),
 Canvas(geo_transform=(-180, 0.1, 0, 90, 0, -0.1), buf_xsize=10, buf_ysize=10, dims=(
↪'y', 'x'), ravel_order='C', zbounds=None, tbounds=None, zsize=None, tsize=None,␣
↪bounds=BoundingBox(left=-180.0, bottom=90.0, right=-179.1, top=89.1)))
```

Here are some notes on the arguments passed to `serialize` if given:

- *y* is an `ElmStore` either 1-D or 2-D (see `to_raster` keyword to *predict_many*)
- *X* is the `X` `ElmStore` that was used for prediction (the *Pipeline* will preserve `attrs` in `X` useful for serializing `y` as in the example above which used the *.canvas* attribute of `X`)
- *tag* is a unique tag of sample and *Pipeline* instance

- *elm_predict_path* is the root dir for serialization output - `ELM_PREDICT_PATH` from *environment variables*.

12.12 Parallel Prediction

To run *predict_many* (or *fit_ensemble* or *fit_ea*) in parallel using a dask-distributed client or dask `ThreadPool` client, use `elm.config.client_context` as shown here (continuing with the namespace defined by the snippets above)

First make sure you are running a `dask-scheduler` and `dask-worker` . Read more here on dask-distributed.

```
with client_context(dask_executor='DISTRIBUTED', dask_scheduler='10.0.0.10:8786') as␣
↪client:
    ensemble_kwargs['client'] = client
    fitted = pipe.fit_ensemble(**data_source, **ensemble_kwargs)
    preds = pipe.predict_many(client=client, **data_source)
```

In the example above, `client_context` could have been called with no arguments if `DASK_EXECUTOR` and `DASK_SCHEDULER` *environment variables*.

With parallel `predict_many` , each ensemble member / sample combination is a separate task - there is no parallelism within transformations of the `Pipeline` .

CHAPTER 13

elm yaml Specs - *Deprecated Temporarily*

> *elm-main is deprecated temporarily while ‘‘elm‘‘ and ‘‘earthio‘‘ undergo significant churn and changes in usage patterns. Around August 1, 2017 ‘‘elm-main‘‘ will be revisited as it provides a ‘‘yaml‘‘ based interface to ‘‘elm‘‘ and may assist in ‘‘elm‘‘ UI contexts or in interoperability.*

Workflows involving ensemble and evolutionary methods and *predict_many* can also be specified in a `yaml` config file for running with the *elm-main* console entry point. The `yaml` config can refer to functions from `elm` or user-given packages or modules. Read more the yaml configuration file format here

The repository elm examples has a number of example `yaml` configuration files for GeoTiff and `HDF4` files as input to K-Means or stochastic gradient descent classifiers.

This page walks through each part of a valid `yaml` config.

13.1 ensembles

The `ensembles` section creates named dicts of keyword arguments to *fit_ensemble*. The example below creates `example_ensemble`, an identifier we can use elsewhere in the config. If passing the keyword `ensemble_init_func` in an ensemble here, then it should be given in *“package.subpackage.module:callable”* notation like a setup.py console entry point, e.g. `"my_kmeans_module:make_ensemble"`.

```
ensembles: {
  example_ensemble: {
    init_ensemble_size: 1,
    saved_ensemble_size: 1,
    ngen: 3,
    partial_fit_batches: 2,
  },
}
```

13.2 data_sources

The dicts in data_sources create a named sampler with their keyword arguments.

In the config, args_list can be a callable. In this case, it is iter_files_recursively a function which takes top_dir and file_pattern as arguments. The filenames returned by iter_files_recursively will be filtered by example_meta_is_day an example function for detecting whether a satellite data file is night or day based on its metadata. If args_list is callable, it should take a variable number of keyword arguments (**kwargs).

This examples creates ds_example which selects from files to get bands 1 through 6, iterating recursively over .hdf files in ELM_EXAMPLE_DATA_PATH from the environment (env:SOMETHING means take SOMETHING from environment variables).

band_specs in the data source are passed to earthio.LayerSpec (See also *ElmStore* and *LANDSAT Example*) and determine which bands (subdatasets in this HDF4 case) to include in a sample.

```
data_sources: {
 ds_example: {
  sampler: "earthio.filters.band_selection:select_from_file",
  band_specs: [{search_key: long_name, search_value: "Band 1 ", name: band_1},
  {search_key: long_name, search_value: "Band 2 ", name: band_2},
  {search_key: long_name, search_value: "Band 3 ", name: band_3},
  {search_key: long_name, search_value: "Band 4 ", name: band_4},
  {search_key: long_name, search_value: "Band 5 ", name: band_5},
  {search_key: long_name, search_value: "Band 6 ", name: band_6},],
  args_list: "earthio.local_file_iterators:iter_files_recursively",
  top_dir: "env:ELM_EXAMPLE_DATA_PATH",
  metadata_filter: "earthio.metadata_selection:example_meta_is_day",
  file_pattern: "\\.hdf",
 },
}
```

See also *Creating an ElmStore from File*

13.3 model_scoring

Each dict in model_scoring has a scoring callable and the other keys/values are passed as scoring_kwargs. These in turn become the scoring and scoring_kwargs to initialize a Pipeline instance. This example creates a scorer called kmeans_aic

```
model_scoring: {
  kmeans_aic: {
    scoring: "elm.model_selection.kmeans:kmeans_aic",
    score_weights: [-1],
  }
}
```

13.4 transform

This section allows using transform model, such as IncrementalPCA from sklearn.decomposition. model_init_kwargs can include any keyword argument to the model_init_class, as well as partial_fit_batches (partial_fit operations on each Pipeline fit or partial_fit).

```
transform: {
  pca: {
    model_init_class: "sklearn.decomposition:IncrementalPCA",
    model_init_kwargs: {"n_components": 2, partial_fit_batches: 2},
  }
}
```

13.5 sklearn_preprocessing

This section configures scikit-learn preprocessing classes (`sklearn.preprocessing`), such as `PolynomialFeatures`, for use elsewhere in the config. Each key is an identifer and each dictionary contains a `method` (imported from `sklearn.preprocessing`) and keyword arguments to that `method`.

```
sklearn_preprocessing: {
  min_max: {
    method: MinMaxScaler,
    feature_range: [0, 1],
  },
  poly2_interact: {
    method: PolynomialFeatures,
    degree: 2,
    interaction_only: True,
    include_bias: True,
  },
}
```

13.6 train

The `train` dict configures the final estimator in a `Pipeline`, in this case `MiniBatchKMeans`. This example shows how to run that estimator with the `example_ensemble` keyword arguments from above, model scoring section from above (`kmeans_aic`), passing `drop_n` and `evolve_n` to the `model_selection` callable.

```
train: {
  train_example: {
    model_init_class: "sklearn.cluster:MiniBatchKMeans",
    model_init_kwargs: {
      compute_labels: True
    },
    ensemble: example_ensemble,
    model_scoring: kmeans_aic,
    model_selection: "elm.model_selection.kmeans:kmeans_model_averaging",
    model_selection_kwargs: {
      drop_n: 4,
      evolve_n: 4,
    }
  }
}
```

13.7 feature_selection

Each key in this section is an identifier and the each dict is a feature selector configuration, naming a `method` to be imported from `sklearn.preprocessing` and keyword arguments to that `method`.

```
feature_selection: {
    top_half: {
        method: SelectPercentile,
        percentile: 50,
        score_func: f_classif
    }

}
```

13.8 run

The `run` section names fitting and prediction jobs to be done by using identifiers created in the config's dictionaries reviewed above.

About the `run` section:

- It is a list of actions
- Each action in the list is a dict
- Each action should have the key `pipeline` that is a list of dictionaries specifying steps (analogous to the interactive session *Pipeline*)
- Each action should have a `data_source` key pointing to one of the `data_sources` named above
- Each action can have `predict` and/or `train` key/value with the value being one of the named `train` dicts above

```
run:
  - {pipeline: [{select_canvas: band_1},
      {flatten: True},
      {drop_na_rows: True},
      {sklearn_preprocessing: poly2_interact},
      {sklearn_preprocessing: min_max},
      {transform: pca}],
     data_source: ds_example,
     predict: train_example,
     train: train_example}
```

The example above showed a `run` configuration with a `pipeline` of transforms inclusive of flattening rasters, dropping null rows, adding polynomial interaction terms, min-max scaling, and PCA.

13.9 Valid steps for run - pipeline

This section shows all of the valid steps that can be a config's `run - pipeline` lists (items that could have appeared in teh `pipeline` list in preceding example).

flatten

Flattens 2-D each `DataArray` raster to a column within a single `DataArray` called `flat` in an *ElmStore*.

```
{flatten: True}
```

See also transform-flatten.

See also: **:docs:'elm.pipeline.steps<pipeline-steps>'**

drop_na_rows

Drops null rows from an `ElmStore` or `xarray.Dataset` with a `DataArray` called `flat` (often this step follows `{flatten: True} in a ``pipeline`).

```
{drop_na_rows: True}
```

See also transform-dropnarows.

modify_sample

Provides a callable and optionally keyword arguments to modify `X` and optionally `y` and `sample_weight`. See example of interactive use of `elm.pipeline.steps.ModifySample` here - TODO LINK and the function signature for a `modify_sample` callable here - TODO LINK. This example shows how to run `normalizer_func` imported from a package and subpackage, passing `keyword_1` and `keyword_2`.

```
{modify_sample: "mypackage.mysubpkg.mymodule:normalizer_func", keyword_1: 4, keyword_
↪2: 99}
```

See also `ModifySample` usage in a K-Means LANDSAT example .

transpose

Transpose the dimensions of the `ElmStore`, like this example for converting from `("y", "x")` dims to `("x", "y")` dims.

```
{transpose: ['x', 'y']}
```

sklearn_preprocessing

If a config has a dict called `sklearn_preprocessing` as in the example above, then named preprocessors in that dict can be used in the `run - pipeline` lists as follows:

```
{sklearn_preprocessing: poly2_interact}
```

where `poly2_interact` is a key in `sklearn_preprocessing`

See also: `elm.pipeline.steps.PolynomialFeatures` in *elm.pipeline.steps*

feature_selection

If a config has a dict called `feature_selection` as in the example above, then named feature selectors there can be used in the `run - pipeline` section like this:

```
{feature_selection: top_half}
```

where `top_half` is a named feature selector in `feature_selection`.

transform

Note the config's `transform` section configures transform models like PCA but they are not used unless the config's `run - pipeline` lists have a `transform` action (dict) in them. Here is an example:

```
{transform: pca}
```

where `pca` is a key in the config's `transform` dict.

CHAPTER 14

elm-main Entry Point *Deprecated Temporarily*

elm-main is deprecated temporarily while ‘‘elm‘‘ and ‘‘earthio‘‘ undergo significant churn and changes in usage patterns. Around August 1, 2017 ‘‘elm-main‘‘ will be revisited as it provides a ‘‘yaml‘‘ based interface to ‘‘elm‘‘ and may assist in ‘‘elm‘‘ UI contexts or in interoperability.

elm-main runs the training and prediction steps configured in a `yaml` file, as described in the *yaml config examples*. If you have not used `yaml` before, read about yaml here first.

The simple use of *elm-main* is to run one `yaml` config:

```
elm-main --config elmexamples/configs/kmeans_hdf4.yaml
```

elm-main uses the *environment variables described here*, many of which may be overriden by optional arguments to *elm-main* (see below).

The next section goes over all the command line options for *elm-main*

14.1 Config(s) To Run

elm-main can run a single `yaml` config or a directory of `yaml` config files. To run with a single `yaml` file, use the `--config` argument as above, or to run with a directory of config `yaml` files, use `--config-dir`

14.2 Controlling Train vs. Predict

The following arguments control which parts of the config are being run:

- `--train-only`: Run only the training actions specified in the `run` section of config
- `--predict-only`: Run only the predict actions specified in config

14.3 Overriding Arguments to `fit_ensemble`

The following arguments, if given, will override similarly named values in the `yaml` config that are associated with the `train` section (configuration of a final estimator in an *Pipeline*:

- `--partial-fit-batches`: Number of `partial_fit` batches for final estimator (this does not control `partial_fit` batches within a `transform` step in `run` - `pipeline` steps)
- `--init-ensemble-size`: Initial ensemble size (ignored if `ensemble_init_func` is given in config(s))
- `--saved-ensemble-size`: Final number of trained models to serialize in each `train` action of the `run` section

14.4 Use Dask Client

To use a `dask-distributed` or dask `ThreadPool` client, use the *environment variables described here* - or override them with command line arguments to *elm-main*:

- `--dask-executor`: One of [`DISTRIBUTED` `SERIAL` or `THREAD_POOL`]
- `--dask-scheduler`: Dask-distributed scheduler url, e.g. `10.0.0.10:8786`

14.5 Directories for Serialization

The following arguments control where trained models and predictions are saved:

- `--elm-train-path`: Trained `Pipeline` instances are saved here - see also `ELM_TRAIN_PATH` in *environment variables*.
- `--elm-predict-path`: Predictions are saved here - see also `ELM_PREDICT_PATH` in *environment variables*.

14.6 Help for elm-main

Here is the full help for *elm-main*:

```
$ elm-main --help
usage: elm-main [-h] [--config CONFIG | --config-dir CONFIG_DIR]
                [--train-only | --predict-only]
                [--partial-fit-batches PARTIAL_FIT_BATCHES]
                [--init-ensemble-size INIT_ENSEMBLE_SIZE]
                [--saved-ensemble-size SAVED_ENSEMBLE_SIZE] [--ngen NGEN]
                [--dask-threads DASK_THREADS]
                [--max-param-retries MAX_PARAM_RETRIES]
                [--dask-executor {DISTRIBUTED,SERIAL,THREAD_POOL}]
                [--dask-scheduler DASK_SCHEDULER]
                [--elm-example-data-path ELM_EXAMPLE_DATA_PATH]
                [--elm-train-path ELM_TRAIN_PATH]
                [--elm-predict-path ELM_PREDICT_PATH]
                [--elm-logging-level {INFO,DEBUG}]

Pipeline classifier / predictor using ensemble and partial_fit methods
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  --train-only          Run only the training, not prediction, actions
                        specified by config
  --predict-only        Run only the prediction, not training, actions
                        specified by config
  --echo-config         Output running config as it is parsed

Inputs:
  Input config file or directory

  --config CONFIG       Path to yaml config
  --config-dir CONFIG_DIR
                        Path to a directory of yaml configs

Run:
  Run options

Control:
  Keyword arguments to elm.pipeline.ensemble

  --partial-fit-batches PARTIAL_FIT_BATCHES
                        Partial fit batches (for estimator specified in
                        config's "train"
  --init-ensemble-size INIT_ENSEMBLE_SIZE
                        Initial ensemble size (ignored if using
                        "ensemble_init_func"
  --saved-ensemble-size SAVED_ENSEMBLE_SIZE
                        How many of the "best" models to serialize
  --ngen NGEN           Number of ensemble generations, defaulting to ngen
                        from ensemble_kwargs in config

Environment:
  Compute settings (see also help on environment variables)

  --dask-threads DASK_THREADS
                        See also env var DASK_THREADS
  --dask-processes DASK_PROCESSES
                        See also env var DASK_PROCESSES
  --max-param-retries MAX_PARAM_RETRIES
                        See also env var MAX_PARAM_RETRIES
  --dask-executor {DISTRIBUTED,SERIAL,THREAD_POOL}
                        See also DASK_EXECUTOR
  --dask-scheduler DASK_SCHEDULER
                        See also DASK_SCHEDULER
  --elm-example-data-path ELM_EXAMPLE_DATA_PATH
                        See also ELM_EXAMPLE_DATA_PATH
  --elm-train-path ELM_TRAIN_PATH
                        See also ELM_TRAIN_PATH
  --elm-predict-path ELM_PREDICT_PATH
                        See also ELM_PREDICT_PATH
  --elm-logging-level {INFO,DEBUG}
                        See also ELM_LOGGING_LEVEL
  --elm-configs-path ELM_CONFIGS_PATH
                        See also ELM_CONFIGS_PATH
  --elm-large-test ELM_LARGE_TEST
                        See also ELM_LARGE_TEST
```

(continues on next page)

(continued from previous page)

```
Pipeline classifier / predictor using ensemble and partial_fit methods

optional arguments:
  -h, --help            show this help message and exit
  --config CONFIG       Path to yaml config
  --config-dir CONFIG_DIR
                        Path to a directory of yaml configs
  --dask-threads DASK_THREADS
                        See also env var DASK_THREADS
  --dask-processes DASK_PROCESSES
                        See also env var DASK_PROCESSES
  --max-param-retries MAX_PARAM_RETRIES
                        See also env var MAX_PARAM_RETRIES
  --dask-executor {DISTRIBUTED,SERIAL,THREAD_POOL}
                        See also DASK_EXECUTOR
  --dask-scheduler DASK_SCHEDULER
                        See also DASK_SCHEDULER
  --ladsweb-local-cache LADSWEB_LOCAL_CACHE
                        See also LADSWEB_LOCAL_CACHE
  --hashed-args-cache HASHED_ARGS_CACHE
                        See also HASHED_ARGS_CACHE
  --elm-example-data-path ELM_EXAMPLE_DATA_PATH
                        See also ELM_EXAMPLE_DATA_PATH
  --elm-train-path ELM_TRAIN_PATH
                        See also ELM_TRAIN_PATH
  --elm-transform-path ELM_TRANSFORM_PATH
                        See also ELM_TRANSFORM_PATH
  --elm-predict-path ELM_PREDICT_PATH
                        See also ELM_PREDICT_PATH
  --elm-logging-level {INFO,DEBUG}
                        See also ELM_LOGGING_LEVEL
  --elm-configs-path ELM_CONFIGS_PATH
                        See also ELM_CONFIGS_PATH
  --echo-config         Output running config as it is parsed
```

Help & Reference

- *API*
- *Environment Variables*
- *py.test Unit Tests*
- *Contributing to elm*
- *Release Procedure*

CHAPTER 15

API

The API help information in this page is autogenerated from comments in the source.

The packages include:

- *earthio*, a collection of tools for reading HDF4, HDF5, NetCDF and GeoTiff files and directories of files
- *elm.pipeline*, the public interface for ensemble learning models (*elm*) and its *Pipeline* class for preprocessing transform steps before fitting in ensemble or in an evolutionary algorithm
- `elm.model_selection`, tools for scoring models, Pareto sorting models, and selecting the best ensemble members
- *earthio.filters*, modules of helper functions to define classes / functions for `elm.pipeline.steps` (the possible steps in a `Pipeline`)

15.1 earthio

15.2 elm.pipeline

15.3 elm.model_selection

15.4 earthio.filters

CHAPTER 16

Environment Variables

The following are environment variables control `elm-main` and are also inputs to other `elm` functions like `elm.config.client_context` (a dask client context):

- `DASK_EXECUTOR`: Dask executor to use. Choices `[DISTRIBUTED, SERIAL, THREAD_POOL]` (default: `SERIAL`)
- `DASK_SCHEDULER`: Dask scheduler URL, such as `10.0.0.10:8786`, if using `DASK_EXECUTOR=DISTRIBUTED`
- `DASK_THREADS`: Number of threads if using `DASK_EXECUTOR==THREAD_POOL`
- `ELM_EXAMPLE_DATA_PATH`: Path to local clone of hhttps://github.com/ContinuumIO/elm/tree/master/examples (used for `py.test`)
- `ELM_LOGGING_LEVEL`: Either `INFO` (default) or `DEBUG`
- `ELM_PREDICT_PATH`: Base path for saving prediction output
- `ELM_TRAIN_PATH`: Base path for saving trained ensembles
- `MAX_PARAM_RETRIES`: How many times to retry in genetic algorithm when parameters are repeatedly infeasible
- `IS_TRAVIS`: If `IS_TRAVIS=1` , then dask's distributed client is not used (the client if started in CI tests can cause hanging)

Testing `elm`

CHAPTER 17

py.test Unit Tests

These testing instructions assume you have cloned the `elm` repository locally and *installed from source*.

Note: Many tests are skipped if you have not defined the environment variable `ELM_EXAMPLE_DATA_PATH` (referring to your local clone of **'elm-examples'**_)

Run the faster running tests:

```
py.test -m "not slow"
```

Running all tests:

```
py.test
```

or get the verbose test output

```
py.test -v
```

and cut and paste a test mark to run a specific test:

```
py.test -k test_bad_train_config
```

When running `py.test` the environment variables related to dask determine whether dask-distributed or thread pool client or serial evaluation is used (See also dask-distributed).

CHAPTER 18

Longer Running Tests

Deprecated with elm-main temporarily

The `elm-run-all-tests` console entry point can automate running of some or all python scripts and `yaml elm-main` config files in `elm's examples` and/or the `py.test` unit tests.

Here is an example that is run from inside the cloned `elm` repository with `elm-examples` cloned in the current directory (see the first two arguments: *./* - cloned `elm` repo and `examples` - the location of cloned `elm-examples`)

```
ELM_LOGGING_LEVEL=DEBUG elm-run-all-tests ./ elm-examples/ --skip-pytest --skip-
↪scripts --dask-clients SERIAL DISTRIBUTED --dask-scheduler 10.0.0.10:8786
```

Here is the full help on `elm-run-all-tests` entry point:

```
$ elm-run-all-tests --help
usage: elm-run-all-tests [-h] [--pytest-mark PYTEST_MARK]
                         [--dask-clients {ALL,SERIAL,DISTRIBUTED,THREAD_POOL} [{ALL,
↪SERIAL,DISTRIBUTED,THREAD_POOL} ...]]
                         [--dask-scheduler DASK_SCHEDULER] [--skip-pytest]
                         [--skip-scripts] [--skip-configs]
                         [--add-large-test-settings]
                         [--glob-pattern GLOB_PATTERN]
                         [--remote-git-branch REMOTE_GIT_BRANCH]
                         repo_dir elm_examples_path

Run longer-running tests of elm

positional arguments:
  repo_dir              Directory that is the top dir of cloned elm repo
  elm_examples_path     Path to a directory which contains subdirectories
                        "scripts", "scripts", and "example_data" with yaml-
                        configs, python-scripts, and example data,
                        respectively

optional arguments:
  -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  --pytest-mark PYTEST_MARK
                        Mark to pass to py.test -m (marker of unit tests)
  --dask-clients {ALL,SERIAL,DISTRIBUTED,THREAD_POOL} [{ALL,SERIAL,DISTRIBUTED,THREAD_
↪POOL} ...]
                        Dask client(s) to test: ['ALL', 'SERIAL',
                        'DISTRIBUTED', 'THREAD_POOL']
  --dask-scheduler DASK_SCHEDULER
                        Dask scheduler URL
  --skip-pytest         Do not run py.test (default is run py.test as well as
                        configs)
  --skip-scripts        Do not run scripts from elm-examples
  --skip-configs        Do not run configs from elm-examples
  --add-large-test-settings
                        Adjust configs for larger ensembles / param_grids
  --glob-pattern GLOB_PATTERN
                        Glob within repo_dir
  --remote-git-branch REMOTE_GIT_BRANCH
                        Run on a remote git branch
```

CHAPTER 19

Contributing to elm

The goals for elm over the next 12 to 18 months are described in the NASA Phase II SBIR summary here regarding elm funding. We welcome pull requests to elm from the community following the github forking model. Please use the [github issues](https://github.com/ContinuumIO/elm/issues) for reporting problems or suggestions related to elm and see the Waffle Board issue tracking system for the status of several repositories related to elm . Feel free to contact us (Continuum Analytics) for more information:

- Peter Steinberg psteinberg [at] continuum [dot] io
- Greg Brener gbrener [at] continuum [dot] io

Related Information:

- *Installation*
- *Releasing elm packcages*
- *Testing*
- *Use Cases*

CHAPTER 20

Release Procedure

- Ensure all tests pass.
- Tag commit and push to github

```
git tag -a x.x.x -m 'Version x.x.x'
git push upstream master --tags
```

- Build conda packages

Define platform/setup specific environment variables *(fill in with your specifics)*

```
# Location of your conda install. For me it's `~/anaconda/`
CONDA_DIR=~/anaconda/

# Platform code. For me it's `osx-64`
PLATFORM=osx-64

# Version number of elm being released (e.g. 0.2.0)
VERSION=0.2.0
```

```
# requires conda-build (conda install conda-build)
conda build conda.recipe/ --python 3.5 --no-anaconda-upload -c conda-forge
```

Next, *cd* into the folder where the builds end up.

```
cd $CONDA_DIR/conda-bld/$PLATFORM
```

Use `conda convert` to convert over the missing platforms (skipping the one for the platform you're currently on):

```
conda convert --platform osx-64 elm-$VERSION*.tar.bz2 -o ../
conda convert --platform linux-64 elm-$VERSION*.tar.bz2 -o ../
conda convert --platform linux-32 elm-$VERSION*.tar.bz2 -o ../
conda convert --platform win-64 elm-$VERSION*.tar.bz2 -o ../
conda convert --platform win-32 elm-$VERSION*.tar.bz2 -o ../
```

Use `anaconda upload` to upload the build to the `elm` channel. This requires you to be setup on *anaconda.org*, and have the proper credentials to push to the `elm` channel.

```
# requires anaconda-client (conda install anaconda-client)
anaconda login
anaconda upload $CONDA_DIR/conda-bld/*/elm-$VERSION*.tar.bz2 -u elm
```

- Repeat `conda build` and `anaconda upload` steps above for `--python 3.4` as well
- Write the release notes:

1. Run `git log` to get a listing of all the changes
2. Remove any covered in the previous release
3. Summarize the rest to focus on user-visible changes and major new features
4. Paste the notes into github, under *n* `releases`, then `Tags`, then `Edit release notes`.