(4/26/2018)

Multitemporal Python Library

"Efficient, chainable time series processing of raster stacks"

1. General

This Python library provides an efficient means of flexibly performing time series analysis on stacks of gridded data. Multitemporal is composed of a core python application that breaks the processing job into pieces and launches workers to perform the processing, plus reusable modules that perform the steps. Each worker handles a configurable sequence of processing steps for a spatial chunk of data. All the inputs and each step are prescribed in a user-configured JSON files.

The code is available at https://github.com/Applied-GeoSolutions/multitemporal, distributed under a GPLv2 open source license.

Authors and contributors: Bobby Braswell, Justin Fisk, Ian Cooke, Nate Rubin, Beth Ziniti

Initial code development was performed at Applied Geosolutions LLC (AGS) and the University of New Hampshire, supported in part by NASA Interdisciplinary Science Grant "Tropical Forest Resilience and Vulnerability to Drought" (Michael Palace PI). Applied Geosolutions serves as the host and maintainer of the code base.

Possible new name: MultitemPyral (-:

2. Structure

The multitemporal repository is structured as a Python package:

```
multitemporal/
├── Dockerfile
├── LICENSE.txt
├── README.md
├── install.sh
├── multitemporal/
│   ├── __init__.py
│   ├── bin/
│   ├── mt.py
│   └── test/
├── requirements.txt
├── setup.cfg
└── setup.py
```

The `bin/` subdirectory contains a large and growing collection of Cython files which are compiled on package build and serve as the primary processing modules that are available for configuring the “steps” of a process..

The `test/` subdirectory currently contains very few cases but the goal is for developers to archive a test for each new module that demonstrates usage. A test case for multitemporal is represented by a single JSON configuration file.

The Python mt.py file is the primary wrapper script that serves as CLI entry point and library.

3. Data Flow

Chaining of modules allows each module to be written to perform a single task that can be reused within multiple chains. Processing and summarizing time series is a primary focus for modules, and the first step in the chain sequence always expects multitemporal data (hence the name of the package) but downstream modules can ingest and produce other products, for example summary statistics, e.g. quantiles or regression coefficients.

Every module has the same function prototype, and expect three inputs: data, missingval, and params, described below:

The data array has either 3 or 4 dimensions (more on this later) and represents the input to the module. Module inputs originate either from data sources or from upstream modules. If the array is 3-dimensional, then the indices represent: layer, year, and pixel, in that order. The layer dimension indexes over time, or over some other kind of values. If the module is first in the chain, then the layer will index over time. If the data array is 4-dimensional, then the indices represent: input, layer, year, and pixel. So the added dimension just represents multiple (strictly more than one) upstream inputs, either from sources or steps.

The missingval is a single value that represents “no data” in the input. Multitemporal will attempt to detect the missing value from the data source and propagate this through without the user having to worry about it. However the user can override this value on the command line.

The params array is always a one-dimensional array, that originates as a JSON list in the configuration file. The length of this list is dependent on the module requirements, and in some cases on the user choice. Typically the list is a relatively small number of control parameters that govern how the processing step operates. For example, a smoothing module will require the smoothing window size. Parameter values start out as floats but can represent integer values as well (this is up to the module developer).

5. Module processing

Within each module, the entire time series (or layer values in some cases for non-initial steps) are available for each pixel, and for each year. The module developer can apply whatever logic or numerical processing is needed to that time series in order to produce the desired output. For example, in some of the most simple cases, one summary value is produced for each year for each pixel (e.g. an annual mean). In other cases, the module might perform an analysis that results in the same number of output layers (e.g. time steps) as inputs.

The module developer controls two aspects of the model output: number of layers (where layer can be time steps or derived quantities), and the number of output years. The number of output layers is controlled by the nout function in the module, which allows the user flexibility to make the number of layers dependent on the input parameters, the number of inputs, or a constant chosen by the developer. Typically, the number of output years is set to be equal to the number of input years, but in some cases it is useful to set the number of output years to be 1, so that the result represents an aggregate or summary across years, not tied to a specific year. This is done by setting the nyrout function in the module.

Currently all modules are written in Cython for computational efficiency, but the module can be anything importable that meets the requirements above. In most cases it will be much easier to develop a new module based on an existing one. All the Cython modules in the repository are built on installation and obviously a module will need to be recompiled if it is modified.

6. Chaining modules

Each module expects inputs of a certain type and structure, and produces outputs of a certain type and structure, depending on the purpose of the module as implemented by the module developer. It is the responsibility of the developer to document those expectations, and the responsibility of the user to implement module steps (as captured in the JSON configuration file) that are consistent with those expectations. Some checks are implemented but there is no general protection against wiring together modules that don't fit together.

A typical module block, representing a single step in a multitemporal chain, looks like this:

```
{
   "name"   : "ndvi-MCD-runningmean",
   "module" : "runningmean",
   "params" : [7,3],
   "inputs" : ["ndvi-MCD-interpolate"],
   "output" : false
},
```

This block says, (1) "this processing step is called ndvi-MCD-runningmean; (2) the name of the modoule to use is runningmean; (3) the parameters are the window size 7 and the shift 3, so this is a moving window smoother that is centered; (4) take the output from the step called ndvi-MCD-interpolate and use it as input; (5) don't save the output to a file, which means this is probably an intermediate step.

If the block is the last step in the chain (or in *a* chain because there can be multiple chains in the config), you will need to make sure that output is set to true, assuming that you want to see the output. If the block is the first step in a chain, it will need to have inputs consisting only of data which are "sources", which are defined at the top of the config file in a block called sources.

A typical source block looks like this:

```
{
  "name"   : "ndvi-TOF",
  "regexp" : "^(\\d{7})_TOF_ndvi-toa-cmasked.tif$",
  "bandnum": 1
},
```

This block says, (1) the name of this source is called ndvi-TOF; (2) this source will be identified in the input directory as files that match the pattern given by regexp; (3) the band to use for this source will be the first band in the file. It is also possible to refer a band by name if that name is included in the file metadata. Instead of overloading bandnum, we use the keyword bandname, e.g.,

```
{
  "name"   : "ndvi-modis",
  "regexp" : "^(\\d{7})_MCD_ndvi.tif$",
  "bandname": "NDVI"
},
```

As mentioned above, some modules support more than one input (notice that the input field in the configuration file is a list). Those modules will expect input data arrays with 4 dimensions instead of three, with the length of the first dimension equal to the number of inputs. For example, the fusion module accepts two inputs, from two upstream steps.

```
{
    "name"   : "daysofgreen",
    "module" : "daysofgreen",
    "params" : [0.25, 0.35],
    "inputs" : ["ndvi-landsat-fused", "rc-landsat-fused"],
    "output" : true
}
```

Module inputs are always represented as a list, even when there is only one input source. Just to clarify terminology, the module inputs are the *layers* of all the sources listed. In this example the module inputs have two sources, but are equal to 1 layer of NDVI.

7. Usage

The main precondition for usage of multitemporal is the existence of a directory containing raster files, some subset of which will serve as the inputs to the multitemporal run. The specific run time options, including location of the input directory, location of output directory, data sources, and processing steps (see more detail below).

There are several multitemporal command line options:

```
parser.add_argument('--nproc', type=int,
                    help='Number of processors to use')

parser.add_argument('--blkrow', type=int,
                    help='Rows per block')

parser.add_argument('--compthresh', type=float,
                    help='Completeness required')

parser.add_argument('--dperframe', type=int,
                    help='Days per time step (frame)')

parser.add_argument('--projdir',
                    help='Directory containing timeseries')

parser.add_argument('--nongips', action="store_true",
                    help='Projdir is not gips compliant')

parser.add_argument('--ymd', action="store_true",
                    help='Date string is YYMMDD (not GIPS-compliant)')

parser.add_argument('--projname',
                    help='Project name to use for output files')

parser.add_argument('--outdir',
                    help='Directory in which to place outputs')
```

However, the most straightforward and most common usage of multitemporal is to simply specify the path to a configuration file, i.e.,

```
multitemporal --conf /path/to/conf.json
```

Additional options can be supplied on top of this configuration by supplying the desired additional flags on the command line (thanks Justin!). All the options have defaults that work for most cases. A few exceptions: to process data from a GIPS export directory set non-gips to false; if the file name date string is YYYYMMDD instead of the expected YYYYDDD, then use the --ymd option; if you want to set the implicit time step to something other than 1 day, use the --dperframe option.

8. Existing modules

This list is not intended to be complete. There are 43 modules. The purpose of the list in this document is to illustrate the different types of modules, and suggest examples to base future modules on based on the desired properties.

See also /multitemporal/multitemporal/bin/module_info.txt which was an early attempt to document modules and contains a longer listing.

name	nout	output	nyrout	params	inputtype	outputtype
annualstats	5	min, max, mean, std, count	nyr	thresh, nframe=1, maxframe=nfr	time	layer
crossings	params[0]	crossings,...	nyr	ncross, thresh, minframe	time	layer
gapfill	nin	input	nyr	minval, maxval, maxgapfrac	time	time
interannualslope	nin	input	1	minframe, maxframe	layer/time	layer
overallmean	2	mean, count	1	minframe, maxframe	layer/time	layer
phenology	5	annmean, height, start, duration, argmax	nyr	thresh, minval, maxval, minframe, maxframe	time	layers

recomposite	params[0]	recomposite, ...	nyr	nout	time	time

This table shows descriptive aspects of a few of the modules.

`nout` indicates the number of outputs per year for the module. This is equal to the number of layers in the output, or the length of the first dimension of the output array. In some cases it is a fixed number; for example the annualstats module always has an output array that is 5 x nyr x npixels. In some cases it is equal to the number of inputs; for example the gapfill module always returns an array that is nin x nyr x npixels. In some cases it is determined at run time by a configuration parameter; for example the recomposite module has an input parameter that determines the number of composites per year, so the output is always param[0] x nyr x npixels where param[0] in this case is the first parameter in the input parameter list.

`output` indicates the actual outputs produced by the module. If nout is a constant, then the outputs can be enumerated as in the table. If nout is equal to nin, then the outputs have the same number as the inputs. This is often a result of the module performing some direct transformation of the inputs, e.g. smoothing or gap filling.

`nyrout` indicates the number of output years for the module, and is usually equal to the number of input years. Alternatively it could be anything, but the most common example is having one output year that represents an aggregate across years.

`params` indicates the input parameters to the module. As mentioned above, a parameter can be used to set the number of output layers or output years. However, parameters are typically used to control the function of a particular numerical analysis.

`inputtype` indicates whether the module is expecting the layer dimension to represent time steps (e.g. days, 8-days, months) or some generic layer that is non-temporal (e.g. mean, median, min, max), or both. For example, you could perform basic statistics on temporal or non-temporal data.

All of the preceding discussion applies to module inputs that are derived from multiple data sources, except that when there are multiple data sources, the input arrays have an additional first dimension (as discussed above).