xarray-extras Documentation

Release 0.4.2

xarray-extras Developers

2019-06-03

CONTENTS

This module offers several extensions to xarray, which could not be included into the main module because they fall into one or more of the following categories:

- They're too experimental
- They're too niche
- They introduce major new dependencies (e.g. numba or a C compiler)
- They would be better done by doing major rework on multiple packages, and then one would need to wait for said changes to reach a stable release of each package - *in the right order*.

The API of xarray-extras is unstable by definition, as features will be progressively migrated upwards towards xarray, dask, numpy, pandas, etc.

CHAPTER

ONE

FEATURES

csv Multi-threaded CSV writer, much faster than `pandas.DataFrame.to_csv()`, with full support for dask and dask distributed.

cumulatives Advanced cumulative sum/productory/mean functions

interpolate dask-optimized n-dimensional spline interpolation

numba_extras Additions to numba

sort Advanced sort/take functions

stack Tools for stacking/unstacking dimensions

CHAPTER

TWO

INDEX

2.1 Installation

2.1.1 Required dependencies

- Python 3.5 or later
- scipy
- xarray
- dask
- numba
- C compiler (only if building from sources)

2.1.2 Deployment

- With pip: **`pip install xarray-extras`**
- With anaconda: **`conda install -c conda-forge xarray-extras`**

2.1.3 Testing

To run the test suite after installing xarray_extras, first install (via pypi or conda)

- py.test: Simple unit testing library

and run `py.test --pyargs xarray_extras`.

2.2 What's New

2.2.1 v0.4.2 (2019-06-03)

- Type annotations
- Mandatory mypy validation in CI
- CI unit tests for Windows now run on Python 3.7
- Compatibility with dask >= 1.1

- Suppress deprecation warnings with pandas >= 0.24
- *to_csv()* changes:
  - When invoked on a 1-dimensional DataArray, the default value for the `index` parameter has been changed from False to True, coherently to the default for pandas.Series.to_csv from pandas 0.24. This applies also to users who have pandas < 0.24 installed.
  - support for `line_terminator` parameter (all pandas versions);
  - fix incorrect line terminator in Windows with pandas >= 0.24
  - support for `compression='infer'` (all pandas versions)
  - support for `compression` parameter with pandas < 0.23

2.2.2 v0.4.1 (2019-02-02)

- Fixed build regression in readthedocs

2.2.3 v0.4.0 (2019-02-02)

- Moved `recursive_diff`, `recursive_eq` and `ncdiff` to their own package recursive_diff
- Fixed bug in *proper_unstack()* where unstacking coords with dtype=datetime64 would convert them to integer
- Mandatory flake8 in CI

2.2.4 v0.3.0 (2018-12-13)

- Changed license to Apache 2.0
- Increased minimum versions: dask >= 0.19, pandas >= 0.21, xarray >= 0.10.1, pytest >= 3.6
- New function *proper_unstack()*
- New functions `recursive_diff` and `ecursive_eq`
- New command-line tool `ncdiff`
- Blacklisted Python 3.7 conda-forge builds in CI tests

2.2.5 v0.2.2 (2018-07-24)

- Fixed segmentation faults in *to_csv()*
- Added conda-forge travis build
- Blacklisted dask-0.18.2 because of regression in argtopk(split_every=2)

2.2.6 v0.2.1 (2018-07-22)

- Added parameter nogil=True to *to_csv()*, which will switch to a C-accelerated implementation instead of pandas to_csv (albeit with caveats). Fixed deadlock in to_csv as well as compatibility with dask distributed. Pandas code (when using nogil=False) is not wrapped by a subprocess anymore, which means it won't be able

to use more than 1 CPU (but compression can run in pipeline). to_csv has lost the ability to write to a buffer - only file paths are supported now.

- AppVeyor integration

2.2.7 v0.2.0 (2018-07-15)

- New function *xarray_extras.csv.to_csv()*
- Speed up interpolation for k=2 and k=3
- CI: Rigorous tracking of minimum dependency versions
- CI: Explicit support for Python 3.7

2.2.8 v0.1.0 (2018-05-19)

Initial release.

2.3 csv

Multi-threaded CSV writer, much faster than `pandas.DataFrame.to_csv()`, with full support for dask and dask distributed.

`xarray_extras.csv.`**`to_csv`**(*x: xarray.core.dataarray.DataArray*, *path: str*, ***, *nogil: bool = True*, ***kwargs*)

Print DataArray to CSV.

When x has numpy backend, this function is functionally equivalent to (but much) faster than):

```
x.to_pandas().to_csv(path_or_buf, **kwargs)
```

When x has dask backend, this function returns a dask delayed object which will write to the disk only when its .compute() method is invoked.

Formatting and optional compression are parallelised across all available CPUs, using one dask task per chunk on the first dimension. Chunks on other dimensions will be merged ahead of computation.

Parameters

- **`x`** – xarray.DataArray with one or two dimensions
- **`path`** (*str*) – Output file path
- **`nogil`** (*bool*) – If True, use accelerated C implementation. Several kwargs won't be processed correctly (see limitations below). If False, use pandas to_csv method (slow, and does not release the GIL). nogil=True exclusively supports float and integer values dtypes (but the coords can be anything). In case of incompatible dtype, nogil is automatically switched to False.
- **`kwargs`** – Passed verbatim to `pandas.DataFrame.to_csv()` or `pandas.Series.to_csv()`

Limitations

- Fancy URIs are not (yet) supported.
- compression='zip' is not supported. All other compression methods (gzip, bz2, xz) are supported.

- When running with nogil=True, the following parameters are ignored: columns, quoting, quotechar, doublequote, escapechar, chunksize, decimal

Distributed

This function supports dask distributed, with the caveat that all workers must write to the same shared mountpoint and that the shared filesystem must strictly guarantee **close-open coherency**, meaning that one must be able to call write() and then close() on a file descriptor from one host and then immediately afterwards open() from another host and see the output from the first host. Note that, for performance reasons, most network filesystems do not enable this feature by default.

Alternatively, one may write to local mountpoints and then manually collect and concatenate the partial outputs.

2.4 cumulatives

Advanced cumulative sum/productory/mean functions

xarray_extras.cumulatives.**cummean**(*x: TV*, *dim: collections.abc.Hashable*, *skipna: Optional[bool] = None*) → TV

$$y_i = mean(x_0, x_1, ...x_i)$$

Parameters

- **x** – xarray.DataArray, xarray.Dataset, or xarray.Variable
- **dim** (*hashable*) – dimension along which to calculate the mean
- **skipna** (*bool*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).

Returns xarray object of the same type, dtype, and shape as x

xarray_extras.cumulatives.**compound_sum**(*x: T*, *c: xarray.core.dataarray.DataArray*, *xdim: collections.abc.Hashable*, *cdim: collections.abc.Hashable*) → T

Compound sum on arbitrary points of x along dim.

Parameters

- **x** – xarray.DataArray or xarray.Dataset containing the data to be compounded
- **c** (*xarray.DataArray*) – array where every row contains elements of x.coords[xdim] and is used to build a point of the output. The cells in the row are matched against x.coords[dim] and perform a sum. If different rows of c require different amounts of points from x, they must be padded on the right with NaN, NaT, or '' (respectively for numbers, datetimes, and strings).
- **xdim** (*hashable*) – dimension of x to acquire data from. The coord associated to it must be monotonic ascending.
- **cdim** (*hashable*) – dimension of c that represent the vector of points to be compounded for every point of dim

Returns xarray object of the same type and dtype as x, with all dims from x and c except xdim and cdim.

example:

```
>>> x = xarray.DataArray(
>>>     [10, 20, 30],
>>>     dims=['x'], coords={'x': ['foo', 'bar', 'baz']})
>>> c = xarray.DataArray(
>>>     [['foo', 'baz', None],
>>>      ['bar', 'baz', 'baz']],
>>>      dims=['y', 'c'], coords={'y': ['new1', 'new2']})
>>> compound_sum(x, c, 'x', 'c')
<xarray.DataArray (y: 2)>
array([40, 80])
Coordinates:
  * y        (y) <U4 'new1' 'new2'
```

xarray_extras.cumulatives.**compound_prod**(*x: T*, *c: xarray.core.dataarray.DataArray*, *xdim: collections.abc.Hashable*, *cdim: collections.abc.Hashable*) → T

Compound product among arbitrary points of x along dim See *compound_sum()*.

xarray_extras.cumulatives.**compound_mean**(*x: T*, *c: xarray.core.dataarray.DataArray*, *xdim: collections.abc.Hashable*, *cdim: collections.abc.Hashable*) → T

Compound mean among arbitrary points of x along dim See *compound_sum()*.

2.5 interpolate

xarray spline interpolation functions

xarray_extras.interpolate.**splrep**(*a: xarray.core.dataarray.DataArray*, *dim: collections.abc.Hashable*, *k: int = 3*) → xarray.core.dataset.Dataset

Calculate the univariate B-spline for an N-dimensional array

Parameters

- **a** (*xarray.DataArray*) – any DataArray
- **dim** – dimension of a to be interpolated. a.coords[dim] must be strictly monotonic ascending. All int, float (not complex), or datetime dtypes are supported.
- **k** (*int*) – B-spline order:

k	interpolation kind
0	nearest neighbour
1	linear
2	quadratic
3	cubic

Returns Dataset with t, c, k (knots, coefficients, order) variables, the same shape and coords as the input, that can be passed to *splev()*.

Example:

```
>>> x = np.arange(0, 120, 20)
>>> x = xarray.DataArray(x, dims=['x'], coords={'x': x})
>>> s = xarray.DataArray(np.linspace(1, 20, 5), dims=['s'])
>>> y = np.exp(-x / s)
>>> x_new = np.arange(0, 120, 1)
```

(continues on next page)

(continued from previous page)

```
>>> tck = splrep(y, 'x')
>>> y_new = splev(x_new, tck)
```

Features

- Interpolate a ND array on any arbitrary dimension
- dask supported on both on the interpolated array and x_new
- Supports ND x_new arrays
- The CPU-heavy interpolator generation (*splrep()*) is executed only once and then can be applied to multiple x_new (*splev()*)
- memory-efficient
- Can be pickled and used on dask distributed

Limitations

- Chunks are not supported along dim on the interpolated dimension.

xarray_extras.interpolate.**splev**(*x_new: xarray.core.dataarray.DataArray*, *tck: xarray.core.dataset.Dataset*, *extrapolate: Union[bool, str] = True*) → xarray.core.dataarray.DataArray

Evaluate the B-spline generated with *splrep()*.

Parameters

- **x_new** – Any DataArray with any number of dims, not necessarily the original interpolation dim. Alternatively, it can be any 1-dimensional array-like; it will be automatically converted to a DataArray on the interpolation dim.
- **tck** (*xarray.Dataset*) – As returned by *splrep()*. It can have been:
  - transposed (not recommended, as performance will drop if c is not C-contiguous)
  - sliced, reordered, or (re)chunked, on any dim except the interpolation dim
  - computed from dask to numpy backend
  - round-tripped to disk
- **extrapolate** –

  True Extrapolate the first and last polynomial pieces of b-spline functions active on the base interval

  False Return NaNs outside of the base interval

  'periodic' Periodic extrapolation is used

  'clip' Return y[0] and y[-1] outside of the base interval

Returns DataArray with all dims of the interpolated array, minus the interpolation dim, plus all dims of x_new

See *splrep()* for usage example.

2.6 numba_extras

Extensions to numba

xarray_extras.numba_extras.**guvectorize**(*signature: str*, *layout: str*, ***kwds*) → Callable[Callable, Any]

Convenience wrapper around `numba.guvectorize()`. Generate signature for all possible data types and set a few healthy defaults.

Parameters

- **signature** (*str*) – numba signature, containing {T}
- **layout** (*str*) – as in `numba.guvectorize()`
- **kwds** – passed verbatim to `numba.guvectorize()`. This function changes the default for cache from False to True.

example:

```
guvectorize("{T}[:], {T}[:]", "(i)->(i)")
```

Is the same as:

```
numba.guvectorize([
    "float32[:], float32[:]",
    "float64[:], float64[:]",
    ...
], "(i)->(i)", cache=True)
```

Note: Discussing upstream fix; see https://github.com/numba/numba/issues/2936.

2.7 sort

Sorting functions

xarray_extras.sort.**topk**(*a: T*, *k: int*, *dim: collections.abc.Hashable*, *split_every: Optional[int] = None*) → T

Extract the k largest elements from a on the given dimension, and return them sorted from largest to smallest. If k is negative, extract the -k smallest elements instead, and return them sorted from smallest to largest.

This assumes that k is small. All results will be returned in a single chunk along the given axis.

xarray_extras.sort.**argtopk**(*a: T*, *k: int*, *dim: collections.abc.Hashable*, *split_every: Optional[int] = None*) → T

Extract the indexes of the k largest elements from a on the given dimension, and return them sorted from largest to smallest. If k is negative, extract the -k smallest elements instead, and return them sorted from smallest to largest.

This assumes that k is small. All results will be returned in a single chunk along the given axis.

xarray_extras.sort.**take_along_dim**(*a: T*, *ind: T*, *dim: collections.abc.Hashable*) → T

Use the output of *argtopk()* to pick points from a.

Parameters

- **a** – any xarray object
- **ind** – array of ints, as returned by *argtopk()*
- **dim** – dimension along which argtopk was executed

An example that uses all of the above functions is *source attribution*. Given a generic function $y = f(x_0, x_1, ..., x_i)$, which is embarassingly parallel along a given dimension, one wants to find:

- the top k elements of y along the dimension
- the elements of all x's that generated the top k elements of y

```
>>> from xarray import DataArray
>>> from xarray_extras.sort import *
>>> x = DataArray([[5, 3, 2, 8, 1],
>>>                [0, 7, 1, 3, 2]], dims=['x', 's'])
>>> y = x.sum('x')  # y = f(x), embarassingly parallel among dimension 's'
>>> y
<xarray.DataArray (s: 5)>
array([ 5, 10,  3, 11,  3])
Dimensions without coordinates: s
>>> top_y = topk(y, 3, 's')
>>> top_y
<xarray.DataArray (s: 3)>
array([11, 10,  5])
Dimensions without coordinates: s
>>> top_x = take_along_dim(x, argtopk(y, 3, 's'), 's')
>>> top_x
<xarray.DataArray (x: 2, s: 3)>
array([[8, 3, 5],
       [3, 7, 0]])
Dimensions without coordinates: x, s
```

2.8 stack

Utilities for stacking/unstacking dimensions

xarray_extras.stack.**proper_unstack**(*array: T*, *dim: collections.abc.Hashable*) → T

Work around an issue in xarray that causes the data to be sorted alphabetically by label on unstack():

https://github.com/pydata/xarray/issues/906

Also work around issue that causes string labels to be converted to objects:

https://github.com/pydata/xarray/issues/907

Parameters

- **array** – xarray.DataArray or xarray.Dataset to unstack
- **dim** (*str*) – Name of existing dimension to unstack

Returns xarray.DataArray or xarray.Dataset with unstacked dimension

CHAPTER

THREE

CREDITS

- *proper_unstack()* was originally developed by Legal & General and released to the open source community in 2018.
- All boilerplate is from python_project_template, which in turn is from xarray.

CHAPTER

FOUR

LICENSE

xarray-extras is available under the open source Apache License.

PYTHON MODULE INDEX

X

INDEX