xarray-events

Mar 21, 2020

Contents

xarray-events is an open-source API based on **xarray**. It provides sophisticated mechanisms to handle *events* easily.

Events data is something very natural to conceive, yet it's rather infrequent to see native support for it in common data analysis libraries. Our aim is to fill this gap in a very general way, so that scientists from any domain can take benefit from this. We're building all of this on top of **xarray** because this is already a well-established open-source library that provides exciting new ways of handling multi-dimensional labelled data, with applications in a wide range of domains of science.

This library makes it possible to *extend* a **Dataset** by introducing events based on the data. Internally it works as an *accessor* to **xarray** that provides new methods to deal with new data in the form of events and also extends the existing ones already provided by it to add compatibility with this new kind of data.

We hope that this project inspires you to rethink how you currently handle data and, if needed, improve it.

CHAPTER 1

Example

Assume we have a **DataFrame** (in a variable called **ds**) of events and a **Dataset** (in a variable called **events**) of sports data in such a way that the events are a meaningful complement to the data stored in the **Dataset**.

```
events = pd.DataFrame(
    {
        'event_type': ['pass', 'goal', 'pass', 'pass'],
        'start_frame': [1, 175, 251, 376],
        'end_frame': [174, 250, 375, 500]
    }
)

ds = xr.Dataset(
    data_vars={
        'ball_trajectory': (
            ['frame', 'cartesian_coords'],
            np.exp(np.linspace((-6, -8), (3, 2), 500))
        )
    },
    coords={'frame': np.arange(1, 501), 'cartesian_coords': ['x', 'y']},
    attrs={'match_id': 12, 'resolution_fps': 25, '_events': events}
)
```

With this API we can do the following:

```
ds
.events.load(events, {'frame': ('start_frame', 'end_frame')})
.events.sel({
    'frame': range(175, 376),
    'start_frame': lambda frame: frame >= 175,
    'end_frame': lambda frame: frame < 376
})
.events.groupby_events('ball_trajectory')
.mean()
```

This will:

- Load the events DataFrame specifying that the columns *start_frame* and *end_frame* define the span of the events as per the Dataset's coordinate *frame*.
- Perform a selection constraining the frames to be only in the range [175, 375].
- Group the **DataVariable** *ball_trajectory* by the events.
- Compute the *mean* of each group.

```
<xarray.DataArray 'ball_trajectory' (event_index: 2, cartesian_coords: 2)>
array([[0.12144595, 0.02556095],
       [0.84426861, 0.22346441]])
Coordinates:
  * cartesian_coords  (cartesian_coords) <U1 'x' 'y'
  * event_index       (event_index) int64 1 2
```

This result can be interpreted as the mean 2D position of the ball over the span of each event during the frames [175, 375]. This is a very powerful set of operations performed via some simple and intuitive function calls. This is the beauty of this API.

1.1 Getting Started

Using `xarray-events` is simple. Here's how to get started.

1.1.1 What are events?

In this section we shall provide a description of what events are.

`xarray`

This library is based on `xarray`. As such, we encourage you to first become familiar with it before continuing with this guide. Their official documentation does already a fantastic job at explaining in detail every aspect of their approach on managing data.

Concept

Now that you're familiar with `xarray`, let's describe what we mean by events.

The dictionary definition goes along the lines of the occurrence of something. Ours doesn't deviate much from this. Depending on the kind of data that the `Dataset` stores, events can be seen as occurrences of systematic processes that share the same characteristics of the data. Hence, we expect the attributes or dimensions of an event to be shared to some extent with the dimensions of the data described by the `Dataset`.

1.1.2 Installation

You can install `xarray-events` using Pypi:

```
pip install xarray-events
```

1.2 Tutorials

In this section we provide tutorials that cover the main functionality of this API in multiple domains of science.

1.2.1 Sports data

In this tutorial we'll be working with sports data.

Raw data

`Dataset`

We're going to work with a `Dataset` that describes a football match consisting of the following:

- The following two coordinates:
  - The *frame* at which the ball is at. There are 250 possible frames in this recording.
  - The *cartesian coordinates* (x,y) of the ball's position.
- A `DataVariable` that describes the *trajectory* of the ball. It is described by the two coordinates of the `Dataset`.
- The following two attributes:
  - The match ID, which is 12.
  - The resolution (25 Hz) of the recording in frames per second.

We can create it manually like this:

```
ds = xr.Dataset(
    data_vars={
        'ball_trajectory': (
            ['frame', 'cartesian_coords'],
            np.exp(np.linspace((-6, -8), (3, 2), 2450))
        )
    },
    coords={
        'frame': np.arange(1, 2451),
        'cartesian_coords': ['x', 'y'],
        'player_id': [2, 3, 7, 19, 20, 21, 22, 28, 34, 79]
    },
    attrs={'match_id': 12, 'resolution_fps': 25}
)
```

The object looks like this:

```
<xarray.Dataset>
Dimensions:           (cartesian_coords: 2, frame: 2450)
Coordinates:
  * frame             (frame) int64 1 2 3 4 5 6 ... 2446 2447 2448 2449 2450
  * cartesian_coords  (cartesian_coords) <U1 'x' 'y'
Data variables:
    ball_trajectory   (frame, cartesian_coords) float64 0.002479 ... 7.389
Attributes:
```

(continues on next page)

(continued from previous page)

```
    match_id:         12
    resolution_fps:   25
```

Events

We're going to create events directly in a `DataFrame` consisting of the following attributes:

- The event *type*, which can be: penalty, pass or goal.
- The frame where the event starts.
- The frame where the event ends.
- The ID of the responsible player.

We can create it manually like this:

```
events = pd.DataFrame({
    'event_type':
        ['pass', 'goal', 'pass', 'pass', 'pass',
        'penalty', 'goal', 'pass', 'pass', 'penalty'],
    'start_frame': [1, 425, 600, 945, 1100, 1280, 1890, 2020, 2300, 2390],
    'end_frame': [424, 599, 944, 1099, 1279, 1889, 2019, 2299, 2389, 2450],
    'player_id': [79, 79, 19, 2, 3, 2, 3, 79, 2, 79]
})
```

The object is just a table that looks like this:

(index)	event_type	start_frame	end_frame	player_id
0	pass	1	424	79
1	goal	425	599	79
2	pass	600	944	19
3	pass	945	1099	2
4	pass	1100	1279	3
5	penalty	1280	1889	2
6	goal	1890	2019	3
7	pass	2020	2299	79
8	pass	2300	2389	2
9	penalty	2390	2450	79

Loading

We can load the events `DataFrame` into our `Dataset` like this:

```
ds = ds.events.load(events)
```

At this point, `ds` contains the (private) attribute `_events` storing `events`.

Selecting

We now move on to the most popular action in `xarray`: selection. Here is where we start grasping the benefits of `xarray-events`. The method provided by `xarray` is very powerful and useful when we need to perform selections on a Dataset only. However, the extended sel() in xarray-events allows you to make selections that also take into account the existence of events data.

Constraints that match only the events

The following sections show examples of how we can make selections in the events DataFrame.

Constraints that specify single values

Say we want to select all passes. We can do it like this:

```
ds.events.sel({'event_type': 'pass'})
```

This returns a Dataset object. To actually see the filtered result, we can do this:

```
ds
.events.sel({'event_type': 'pass'})
.events.df
```

And the resulting DataFrame looks like this:

(index)	event_type	start_frame	end_frame	player_id
0	pass	1	424	79
2	pass	600	944	19
3	pass	945	1099	2
4	pass	1100	1279	3
7	pass	2020	2299	79
8	pass	2300	2389	2

See? We are now using the accessor twice, once every time we need to access any of its methods. First we access the method sel() and then the property df(). This is because the result of sel() is actually a (stateful) Dataset, as mentioned before, so we use the accessor again on it in a chain-like fashion. Very convenient!

Constraints that specify collections

Selecting in the events DataFrame by the values in any arbitrary Collection is possible. The usefulness of this possibility becomes more evident when there is a complex process behind obtaining such a collection. For the sake of an example, let's assume such a collection is already given.

Say we want to select all events where the player is in some specified list. We can do it like this:

```
ds
.events.sel({'player_id': [2, 3]})
.events.df
```

And the resulting DataFrame looks like this:

(index)	event_type	start_frame	end_frame	player_id
3	pass	945	1099	2
4	pass	1100	1279	3
5	penalty	1280	1889	2
6	goal	1890	2019	3
8	pass	2300	2389	2

Constraints that specify *lambda functions*

Selecting ranges in the events `DataFrame` is possible. We provide support for lambda functions in order to empower the user to specify arbitrary conditions. In the example below we're going to use a simple condition but, evidently, it can be as complex as needed.

Say we want to select all events that occurred between frames 327 and 1327. We can do it like this:

```
ds
.events.sel({
    'start_frame': lambda frame: frame > 327,
    'end_frame': lambda frame: frame < 1327
})
.events.df
```

And the resulting `DataFrame` looks like this:

(index)	event_type	start_frame	end_frame	player_id
0	pass	1	424	79
1	goal	425	599	79
2	pass	600	944	19
3	pass	945	1099	2
4	pass	1100	1279	3

Constraints that match only the `Dataset`

We may also perform a regular selection on the `Dataset` as we would without this accessor. In that case, `sel()` works just as it does on `xarray`. Although we support this functionality, it's simpler to just stick to the `xarray` method.

Note: We also support the regular method arguments of the `sel()` in `xarray`.

Constraints that match everything

Let's now see how we put together all of the functionality of our accessor to make useful (though complex) queries.

Given that now `sel()` considers two different search spaces (i.e. the events `DataFrame` and the `Dataset`), we can make the search be so complex that it searches in both spaces. This is a powerful feature of our accessor.

Say we may wish to perform a selection with the following specification:

- An event of type *pass*.
- The frames are within 1728 and 2378.

Moreover, we want the result to be consistent across the `Dataset` and the events `DataFrame`. In that case, we can achieve this like this:

```
ds
.events.sel({
    'frame': range(1729, 2378),
    'start_frame': lambda frame: frame > 1728,
    'end_frame': lambda frame: frame < 2378,
    'event_type': 'pass'
})
```

Internally, `sel()` filters the events `DataFrame` and also the `Dataset`, each with its corresponding attributes.

The resulting `Dataset` looks like this:

```
<xarray.Dataset>
Dimensions:            (cartesian_coords: 2, frame: 650, player_id: 10)
Coordinates:
  * frame              (frame) int64 1728 1729 1730 1731 ... 2374 2375 2376 2377
  * cartesian_coords   (cartesian_coords) <U1 'x' 'y'
  * player_id          (player_id) int64 2 3 7 19 20 21 22 28 34 79
Data variables:
    ball_trajectory    (frame, cartesian_coords) float64 1.414 0.3875 ... 5.484
Attributes:
    match_id:         12
    resolution_fps:   25
    _events:            event_type  start_frame  end_frame  player_id\n7      ...
```

And the resulting `DataFrame` looks like this:

(index)	event_type	start_frame	end_frame	player_id
7	pass	2020	2299	79

We want to emphasize how we give the user the power to do things exactly as they want them since the constraints have to be properly specified for both the `Dataset` and also the `DataFrame`. `sel()` does not assume that they may want to select both or anything like that. It all must be specified. This provides great flexibility.

Expanding an events column to match the `Dataset`'s shape

In this section we're going to demonstrate how to use `expand_to_match_ds()`. Some observations first:

- The events `DataFrame` doesn't have a custom index name, so we're going to let `fill_value_col` be "event_index".
- We're going to fill the output `DataArray` with the index of each event in a forward-fill way. It's important that this column be unique, which is the case this way.
- We're going to use *start_frame* as the `dimension_matching_col` by previously specifying that it maps to `frame` in the `Dataset`. This mapping is consistent since the values of *start_frame* form a subset of the values of `Dataset`.

By calling `expand_to_match_ds()` this way we'll be constructing a `DataArray` with the following properties:

- The coordinate is `frame`.
- At each position, there's the (unique) index of each event repeated forward until a new index needs to be placed. Therefore, **each value represents the event that is currently taking place at the frame determined by the coordinate**.

To do this, we first need to make sure to call `load()` specifying the mapping and then call `expand_to_match_ds()` with the values already discussed:

```
ds
.events.load(events, {'start_frame': 'frame'})
.events.expand_to_match_ds('start_frame', 'event_index', 'ffill')
```

This will produce the following `DataArray`:

```
<xarray.DataArray 'event_index' (frame: 2450)>
array([0, 0, 0, ..., 9, 9, 9])
Coordinates:
* frame    (frame) int64 1 2 3 4 5 6 7 ... 2444 2445 2446 2447 2448 2449 2450
```

Grouping a data variable by the events in the `DataFrame`

In this section we're going to illustrate how to use the method `groupby_events()`.

Let's start by recalling that our `Dataset` contains the data variable `ball_trajectory` that has shows for each frame the cartesian coordinates of the ball. Our goal in this tutorial is to find out where the majority of these points lie for each group.

We can start by generating the groups. These groups will tell us which positions in `ball_trajectory` correspond to the frames determined by each event. Then we'll compute the median of them. We can do that like this:

```
ds
.events.groupby_events('ball_trajectory', 'start_frame', 'ffill')
.median()
```

The resulting `DataArray` looks like this:

```
<xarray.DataArray 'ball_trajectory' (event_index: 10, cartesian_coords: 2)>
array([[5.39252098e-03, 7.95626970e-04],
       [1.62105883e-02, 2.70288906e-03],
       [4.21467117e-02, 7.81448485e-03],
       [1.05625415e-01, 2.16889707e-02],
       [1.95479999e-01, 4.29810185e-02],
       [8.34698826e-01, 2.15651098e-01],
       [3.25129820e+00, 9.76995999e-01],
       [6.90622435e+00, 2.25647449e+00],
       [1.36302613e+01, 4.80287169e+00],
       [1.79888284e+01, 6.53714824e+00]])
Coordinates:
  * cartesian_coords  (cartesian_coords) <U1 'x' 'y'
  * event_index       (event_index) int64 0 1 2 3 4 5 6 7 8 9
```

1.3 Development

Welcome to the developer section of xarray-events! Here you'll find a comprehensive technical reference of this API.

1.3.1 API Reference

The following are the main methods of `EventsAccessor`. The contents of each section have been automatically generated from the docstrings. For more details, refer to *Tutorials*.

Each section additionally includes the class definition as well as its properties, so that each method definition is easy to follow.

load

sel

groupby_events

1.3.2 Requirements

This API is based on the following dependencies:

- python
- xarray
- pandas

Additionally, the tests also require the following dependencies:

- pytest
- numpy

Note: the tests have been done using the latest version of each dependency.

CHAPTER 2

License

This API is licensed under Apache 2.0.