Combining Observations

Combining Observations

An anomaly may be linked to a sequence of observations

In [5]:
```
zstart = windows.loc[4]['begin']
zend = windows.loc[4]['end']
zsignal = signal[(signal.index >= zstart) & (signal.index < zend)]
nab.plot_series(zsignal, labels=labels.loc[4:5])
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

The `loc` field in pandas addresses the index of a `DataFrame` or `Series`

Combining Observations

An anomaly may be linked to a sequence of observations

It's a frequent case in real life:

- Isolated outliers may be due to measurement noise
  - e.g. faulty sensors, human mistakes
- Real anomalies often persist for a while

In this case it may be worth to combine multiple probabilities

- We will start by seeing a simple approach
- ...Which makes the assumption that the observations are i.i.d.
  - I.i.d.: Independent and Identically Distributed

Combining Observations

- Let $\mathbf{x}$ be a random variable corresponding to $\boldsymbol{n}$ subsequent observations
- We can formulate our new detection condition as follows:

$$P(\mathbf{x}) \leq \theta^n$$

Since we are assuming i.i.d. observations, we get:

$$\prod_{i=1}^{n} P(x_i) < \theta^n$$

With a log transformation:

$$\sum_{i=1}^{n} \log P(x_i) < n \log \theta$$

Combining Observations

Finally, we get:

$$\frac{1}{n}\sum_{i=1}^{n}\log P(x_i) < \theta$$

Intuitively:

- Considering multiple (independet, identical) observations
- ...It's the same as smoothing our signal using a moving average

Convolutions

We can implement the smoothing via a convolution:

Given a sequence $\{x_i\}_{i=1}^n$ and a sequence $\{f_j\}_{j=1}^m$ (a filter)

- A convolution is an operation that "slides" the filter over the main series
- ...And computes dot products to yield a third sequence $\{y_k\}_{k=m}^n$ s.t.:

$$y_k = f \cdot \left(x_{k-m} \quad x_{k-m+1} \quad x_{k-m+2} \quad \ldots \quad x_k \right)$$

- I.e. the filter is applied to the first m terms...
- ...Then we move one time step forward and we repeat

Normally we need at least m values before the first filter application

- Hence, the output series will be shorter than the input one
- This is depicted by having the y sequence start from index m
- There are other ways (not discussed) to handle the series boundaries

Convolutions

We want to compute a moving average

...Which is just the average of the last few values:

- Let m be the length of the time window for the moving average
- Let us choose as filter: $\left(\frac{1}{m}, \frac{1}{m}, \ldots\right)$

The convolution will compute an output sequence $\{y_k\}_{k=m}^{m}$, s.t.:

$$y_k = \frac{1}{m} \sum_{i=k-m}^{k} x_i$$

This is exactly what we need!

Convolutions

First we build the filter:

In [6]:
```
avg_win_len = 24
flt = np.ones(avg_win_len) / avg_win_len
flt
```

Out[6]:
```
array([0.04166667, 0.04166667, 0.04166667, 0.04166667, 0.04166667,
       0.04166667, 0.04166667, 0.04166667, 0.04166667, 0.04166667,
       0.04166667, 0.04166667, 0.04166667, 0.04166667, 0.04166667,
       0.04166667, 0.04166667, 0.04166667, 0.04166667, 0.04166667,
       0.04166667, 0.04166667, 0.04166667, 0.04166667])
```

The we apply the convolution:

In [7]:
```
smooth_signal = -np.convolve(ldens, flt, mode='valid')
smooth_signal_idx = data.index[avg_win_len-1:]
smooth_signal = pd.Series(index=smooth_signal_idx, data=smooth_signal)
```

- The convolution needs n observations before it can be first applied
- Hence, we need to update the index for our smoothed signal

Convolutions

In pandas, we can streamline this process via the `rolling` iterator

```
In [24]: signal.rolling(avg_win_len)

Out[24]: Rolling [window=24,center=False,axis=0,method=single]
```

- This will iterate over the series by groups of `avg_win_len` observations
- Then we can compute the average with:

```
In [23]: signal.rolling(avg_win_len).mean()

Out[23]: timestamp
         2014-07-01 00:00:00         NaN
         2014-07-01 00:30:00         NaN
         2014-07-01 01:00:00         NaN
         2014-07-01 01:30:00         NaN
         2014-07-01 02:00:00         NaN
                                    ...
         2015-01-31 21:30:00    0.104761
         2015-01-31 22:00:00    0.159036
         2015-01-31 22:30:00    0.232360
         2015-01-31 23:00:00    0.280827
         2015-01-31 23:30:00    0.307642
         Length: 10320, dtype: float64
```

Combining Observations

Let's plot the smoothed signal

In [25]:
```
nab.plot_series(smooth_signal, labels=labels, windows=windows)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

Some anomalies are now more evident!

Threshold Effect

We can now measure the effect of changing the threshold

First, we consider the "idealized" cost surface

In [29]:
```
smooth_thr_range = np.linspace(0.5, 2, 100)
smooth_cost_range = [cmodel.cost(smooth_signal, labels, windows, thr)
              for thr in smooth_thr_range]
smooth_cost_range = pd.Series(index=smooth_thr_range, data=smooth_cost_range)
nab.plot_series(smooth_cost_range)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

Threshold Effect

It is worth to compare it with our orignial cost surface:

In [30]:

```
thr_range = np.linspace(1, 10, 100)
cost_range = [cmodel.cost(signal, labels, windows, thr) for thr in thr_range]
cost_range = pd.Series(index=thr_range, data=cost_range)
nab.plot_series(cost_range)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

The minimum is a bit lower with the smoothed signal

Threshold Optimization

We can now optimize the threshold

In [31]:
```
smooth_signal_opt = smooth_signal[smooth_signal.index < val_end]
smooth_best_thr, smooth_best_cost = nab.opt_thr(smooth_signal_opt, labels_opt,
                                  windows_opt,  cmodel, smooth_thr_range)
print(f'Best threshold: {smooth_best_thr}, corresponding cost: {smooth_best_cost}')
```

```
Best threshold: 0.5303030303030303, corresponding cost: 15
```

- Same cost as before on the trainining set
- On the whole dataset, however:

In [32]:
```
smooth_ctst = cmodel.cost(smooth_signal, labels, windows, smooth_best_thr)
print(f'Cost on the whole dataset {smooth_ctst}')
```

```
Cost on the whole dataset 37
```

- The cost with our original approach used to be 45

Some Considerations

It may worth combining multiple observations:

- If we notice that the data is noise
- ...Or if we are interested in persistent anomalies

Combining multiple observations with an i.i.d. assumption...

- ...Is equivalent to smoothing with a moving average
- ...And the other way round!
- Warning: the assumption may not be valid

The approach introduces an extra parameter (window length):

- In principle, we should optimize over that as well
- We skipped that part for sake of simplicity