Sequence Input in KDE

Sequence Input in KDE

Can we take sequence input into account in KDE?

There is straightforward approach, using multivariate KDE

- Treat each sequence as a vector variable
- Learn an estimator as usual

Individual sequences in the new dataset are treated as independent:

- This is due to the basic assumptions behind KDE
- In practice, for a sufficiently high window length
- ...The dependencies become negligible

Does it sound familiar?

Sequence Input in KDE

Can we take sequence input into account in KDE?

There is straightforward approach, using multivariate KDE

- Treat each sequence as a vector variable
- Learn an estimator as usual

Individual sequences in the new dataset are treated as independent:

- This is due to the basic assumptions behind KDE
- In practice, for a sufficiently high window length
- ...The dependencies become negligible

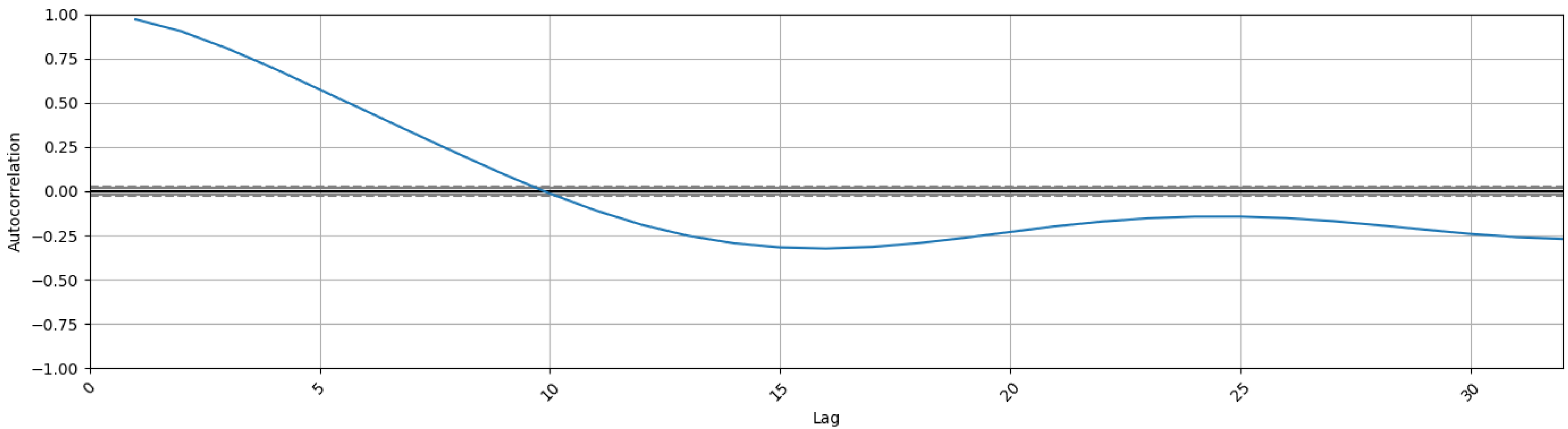
Does it sound familiar?

This is simply the Markov property!

Picking a Window Length

This suggests a way to select the window length

In [15]:
```
nab.plot_autocorrelation(data, max_lag=32, figsize=figsize)
```



I.e. we end the window where the correlation becomes too low (e.g. 10 in our case)

Bandwidth Choice in Multivariate KDE

We now need to learn our multivariate KDE estimator

First, we need to choose a bandwidth

- We cannot use the (univariate) rule of thumb
- ...But we can use a more general approach

The basic intuition is that a good bandwidth

...Will make the actual data register as more likely

- Therefore we can pick a validation set
- ...And tune the bandwidth for maximum likelihood

To avoid overfitting, there should be no overlap with the training data

Bandwidth Choice in Multivariate KDE

Formally, let $\tilde{x}$ be a validation set of m examples:

Assuming independent observations, its likelihood is:

$$L(\tilde{\mathbf{x}}, \hat{\mathbf{x}}, h) = \prod_{i=1}^{m} f(\tilde{x}_i, \hat{\mathbf{x}}, h)$$

The likelihood is the estimated probability of a sample

- ...As a function of the model parameters
- f is the density estimator (which outputs a probability)
- $\hat{\mathbf{x}}$ the training set, h is the bandwidth

Bandwidth Choice in Multivariate KDE

We can then choose h so as to maximize the likelihood

Meaning that the training problem is given by:

$$\arg\max_h \mathbb{E}_{\tilde{x} \sim D} \left[L(\tilde{\mathbf{x}}, \hat{\mathbf{x}}, h) \right]$$

- Where D is ideally the true distribution

As many training problem, it cannot be solved in an exact fashion

- Instead we will approximate $\mathbb{E}$ by sampling multiple $\tilde{x}$
- ...And pick the bandwidth h^* leading to the maximum average likelihood

In a pinch, we could even use a single $\tilde{x}$

Bandwidth Choice in Multivariate KDE

A simple approach consist in using grid search

- It's the same approach that we used for optimizing the threshold
- scikit learn provides a convenient implementation
- ...Which resorts to cross-fold validation to define $\tilde{x}$

First, we separate the training set as usual:

```
In [16]: wdata_tr = wdata[wdata.index < train_end]
```

Then we specify the values we want to consider for each parameter:

```
In [17]: params = {'bandwidth': np.linspace(400, 800, 20)}
```

Training Multivariate KDE

Finally, we can run the grid search routine

```
In [11]: gs_kde = GridSearchCV(KernelDensity(kernel='gaussian'), params, cv = 5)
         gs_kde.fit(wdata_tr)
         gs_kde.best_params_

Out[11]: {'bandwidth': 568.421052631579}
```

- `cv` is the number of folds
- After training, `GridSearchCV` acts as a proxy for the best estimator

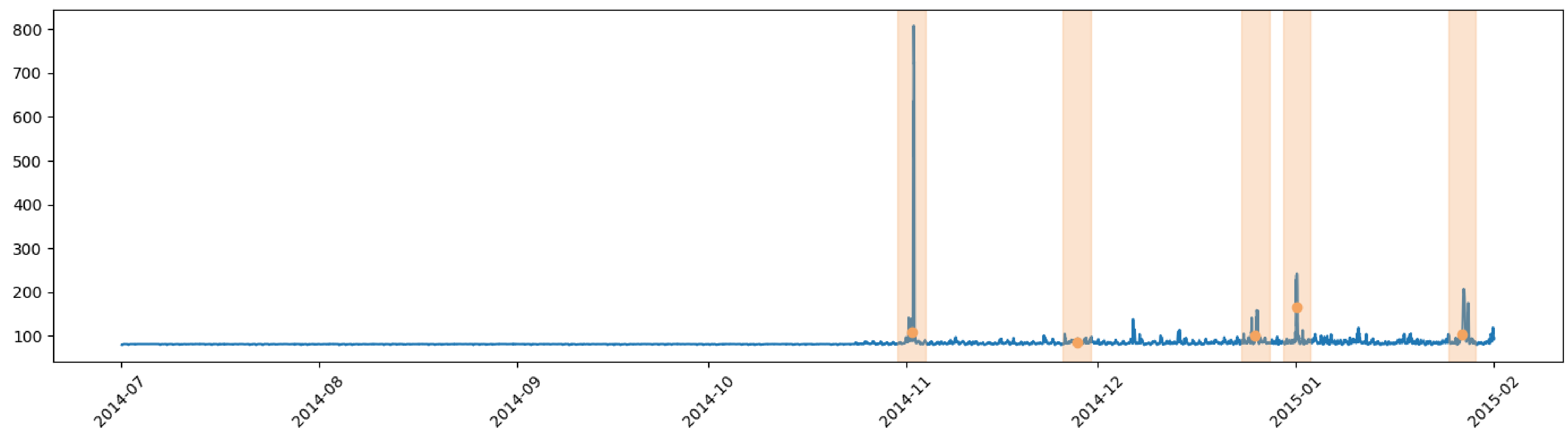
This is an expensive operation

- We need to test multiple bandwidth values
- For each one, we need to perform cross-validation
- ...And finally adding dimensions makes KDE slower

Sequences via Multivariate KDE

Now we can use the best estimator to generate the alarm signal

```
In [12]: ldens = gs_kde.score_samples(wdata)
         signal = pd.Series(index=wdata.index, data=-ldens)
         nab.plot_series(signal, labels, windows, figsize=figsize)
```



- The signal seems visibly better than before (but a bit noisy)

Threshold Optimization

Finally, we can do threshold optimization as usual

In [13]:
```
signal_opt = signal[signal.index < val_end]
labels_opt = labels[labels < val_end]
windows_opt = windows[windows['end'] < val_end]
thr_range = np.linspace(50, 200, 100)

best_thr, best_cost = nab.opt_thr(signal_opt, labels_opt, windows_opt,  cmodel, thr_range)
print(f'Best threshold: {best_thr}, corresponding cost: {best_cost}')
```

```
Best threshold: 104.54545454545455, corresponding cost: 7
```

Cost on the whole dataset

In [14]:
```
ctst = cmodel.cost(signal, labels, windows, best_thr)
print(f'Cost on the whole dataset {ctst}')
```

```
Cost on the whole dataset 30
```