

# KDE for Our Anomaly Detection Case Study

---

## A Few Tweaks

**We will work with log probabilities**

- This is what sklearn does by default
- ...And simplifies some operations
- I.e. products become sums

**Additionally, we will work with negated (log) probabilities**

- They can be interpreted as alarm signals
- ...Which is the customary approach in anomaly detection

**Overall, our anomaly detection condition becomes:**

$$-\log f(x, \omega) \geq \theta$$

...Which is equivalent to the previous formulation

# Training and Testing

## We will split our data in two segments

A **training set**:

- This will include only data about the normal behavior
- Ideally, there should be no anomalies here (we do not want to learn them!)
- We will use it to fit a KDE model

A **test set**:

- To assess how well the approach can generalize

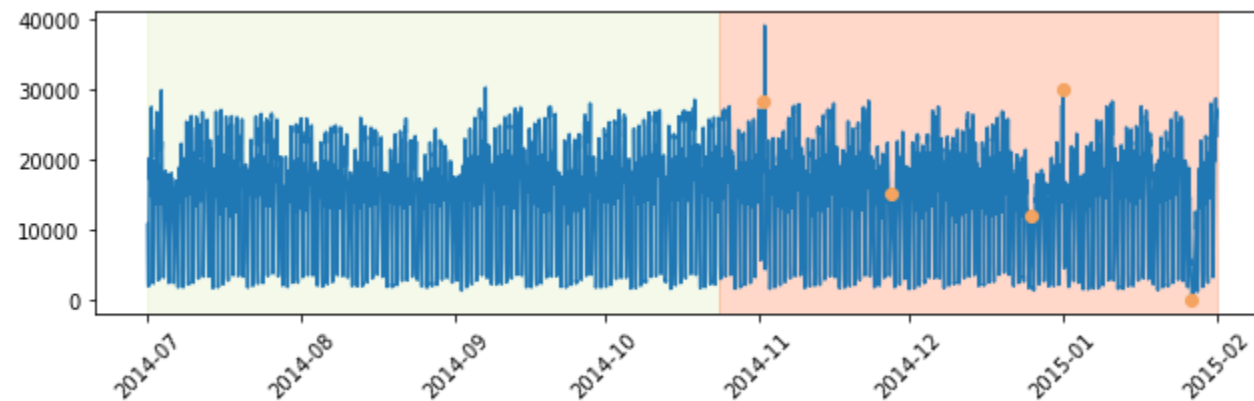
## If the training set contains some anomalies

- Things may still be fine, as long as they are very infrequent
- ...Since we will still learn that they have low probability

# Training and Testing

In time series data sets are often split chronologically:

```
In [2]: train_end = pd.to_datetime('2014-10-24 00:00:00')  
  
nab.plot_series(data, labels, test_start=train_end)
```



■ Green: training set, orange: test set

# Normalization

## We start by normalizing the data

- We isolate the training data
- We can use a date as a separator since we have a `datetime` index

```
In [3]: tmp = data[data.index < train_end]
```

- Then we compute the maximum on the training data:

```
In [4]: trmax = tmp['value'].max()
```

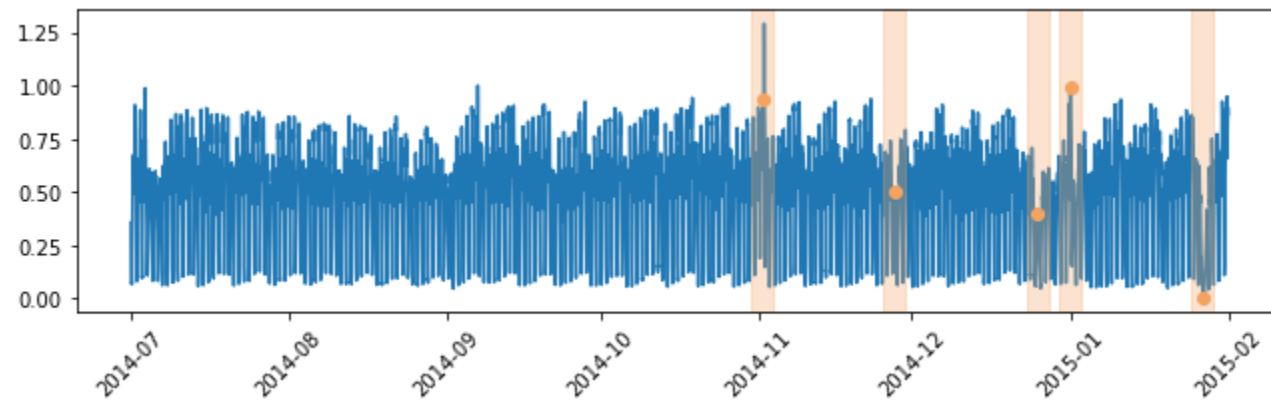
- Finally, we normalize all data
- ...And separate the normalized training data

```
In [5]: data['value'] = data['value'] / trmax  
data_tr = data[data.index < train_end]
```

# Normalization

Here we see the result:

```
In [6]: nab.plot_series(data, labels, windows)
```



Same as before, just with a different scale

# Fitting the Estimator

## Now, we need to estimate the bandwidth

Since we are in a univariate case, we will use the rule of thumb:

```
In [7]: q1 = data_tr['value'].quantile(0.25)
q3 = data_tr['value'].quantile(0.75)
sigma = data_tr['value'].std()
m = len(data_tr)
h = 0.9 * min(sigma, (q3-q1) / 1.34) * m**(-0.2)
print(f'The estimated bandwidth is {h:.3f}')
```

The estimated bandwidth is 0.035

## Then we fit a univariate KDE estimator

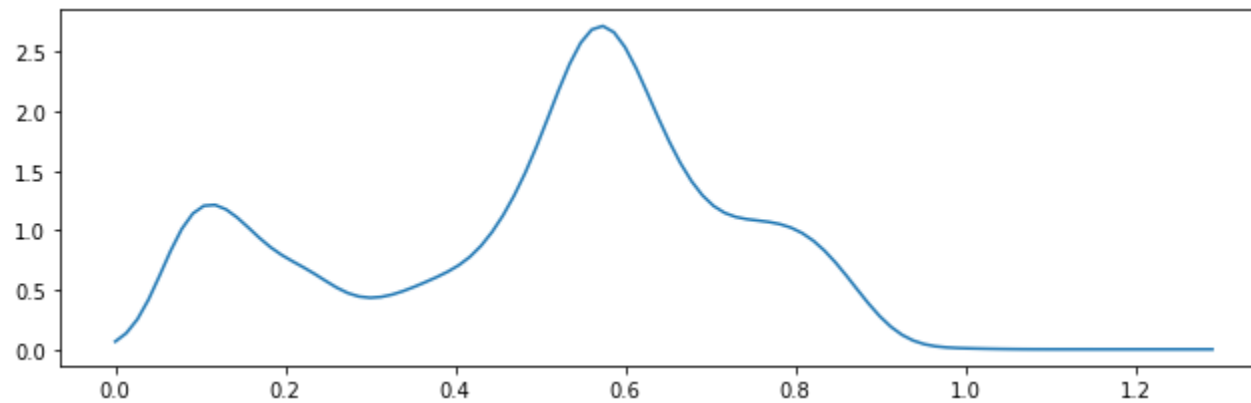
- We will use the scikit-learn library

```
In [8]: kde = KernelDensity(kernel='gaussian', bandwidth=h)
kde.fit(data_tr);
```

# Fitting the Estimator

Let's have a look at the estimate distribution:

```
In [9]: vmax = data['value'].max()
xr = np.linspace(0, vmax, 100)
nab.plot_density_estimator_1D(kde, xr)
```

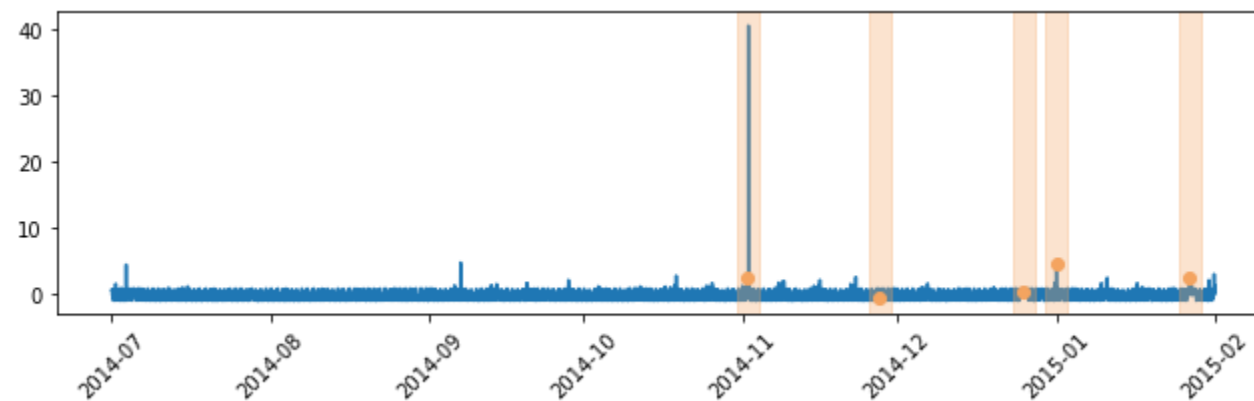




# Alarm Signal

We can now obtain (and plot) our alarm signal:

```
In [10]: ldens = kde.score_samples(data) # Obtain log probabilities  
signal = pd.Series(index=data.index, data=-ldens) # Build series with neg. prob.  
nab.plot_series(signal, labels=labels, windows=windows) # Plot
```



- Noticed how the the process (inference) was noticeably slower than training?
- Can you tell why this was the case?

# Detecting Anomalies

Let us pick a threshold (at random, for now) and try to detect some anomalies:

```
In [11]: thr = 4
pred = pd.Series(signal.index[signal >= thr])
print(pred)

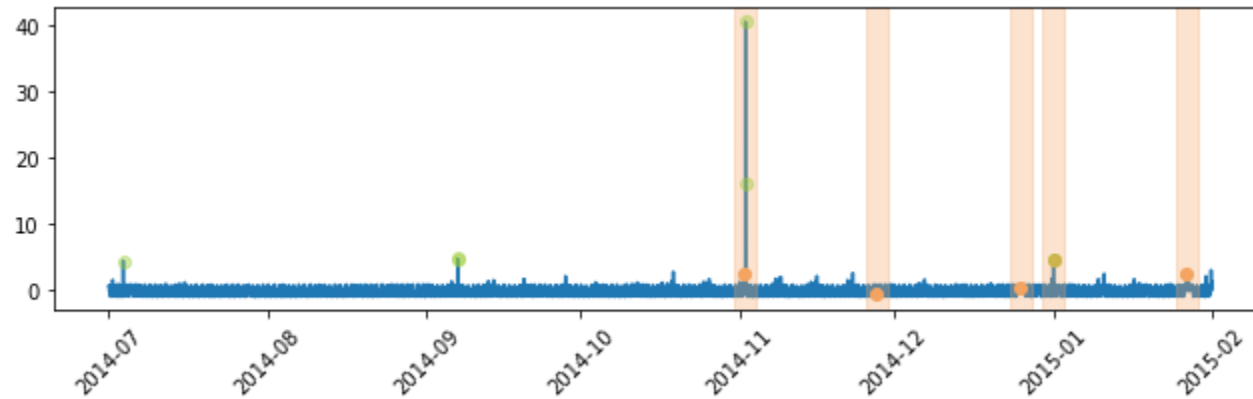
0    2014-07-03 19:00:00
1    2014-09-06 22:30:00
2    2014-09-06 23:00:00
3    2014-11-02 01:00:00
4    2014-11-02 01:30:00
5    2015-01-01 01:00:00
Name: timestamp, dtype: datetime64[ns]
```

- We just apply the filter `signal >= thr` to the signal index
- This yields a number of (predicted) anomalous timestamps
- `pred` is pandas `series` with the detected anomalous timestamps
- Our module contains a function for this step:

# Detecting Anomalies

Let us plot our predictions on the series:

```
In [12]: nab.plot_series(signal, labels=labels, windows=windows, predictions=pred)
```



- Not bad, especially for a random attempt!
- There are a few false positives, which are **very common** in anomaly detection

# Metrics for Anomaly Detection

---

# Metrics for Anomaly Detection

**Evaluating the quality of an Anomaly Detection system can be tricky**

- Usually, we do not need to match the anomalies exactly
- Sometimes we wish to anticipate anomalies
- ...But sometimes we just want to detect them in past data

There is no "catch-all" metric, like accuracy in classification

**It is much better to devise a cost model**

- We evaluate the cost and benefits of our predictions:
- By doing this, we focus on the value for our customer

**This is important for all industrial problems!**

# A Simple Cost Model

## We will use a simple cost model

Remember that our goals are:

- Analyzing anomalies
- Anticipating anomalies

First, we define:

- **True Positives** as windows for which we detect at least one anomaly
- **False Positives** as detected anomalies that do not fall in any window
- **False negatives** as anomalies that go undetected
- **Advance** as the time between an anomaly and when first we detect it

# A Simple Cost Model

They can be computed using a function in our module:

```
def get_metrics(pred, labels, windows)
```

```
In [13]: tp, fp, fn, adv = nab.get_metrics(pred, labels, windows)
print('True Positives', tp)
```

```
True Positives 0    2014-11-01 19:00:00
1    2015-01-01 01:00:00
dtype: datetime64[ns]
```

```
In [14]: print('False Positives', fp)
```

```
False Positives 0    2014-07-03 19:00:00
1    2014-09-06 22:30:00
2    2014-09-06 23:00:00
dtype: datetime64[ns]
```

# A Simple Cost Model

They can be computed using a function in our module:

```
def get_metrics(pred, labels, windows)
```

```
In [15]: print('False Negatives', fn)
```

```
False Negatives 0    2014-11-27 15:30:00
1    2014-12-25 15:00:00
2    2015-01-27 00:00:00
dtype: datetime64[ns]
```

```
In [16]: print('Advance', adv)
```

```
Advance 0    -1 days +18:00:00
1         0 days 00:00:00
dtype: timedelta64[ns]
```



# A Simple Cost Model

Then we introduce:

- A cost  $c_{alarm}$  for loosing time in analyzing false positives
- A cost  $c_{missed}$  for missing an anomaly
- A cost  $c_{late}$  for a late detection (partial loss of value)

Our cost is then given by:

```
In [17]: c_alarm = 1 # Cost of investigating a false alarm
c_missed = 10 # Cost of missing an anomaly
c_late = 5 # Cost for late detection

# A simple cost model
cost = c_alarm * len(fp) + \
      c_missed * len(fn) + \
      c_late * (len([a for a in adv if a.total_seconds() <= 0]))
print(f'The cost with the current predictions is: {cost}')
```

The cost with the current predictions is: 43

# Threshold Choice

---

# Refactoring the Cost Model

We will use this cost model for **choosing our threshold**

With this aim, we encapsulate the formula in a Python class in our module:

```
class ADSimpleCostModel:
    def __init__(self, c_alarm, c_missed, c_late):
        self.c_alarm = c_alarm
        self.c_missed = c_missed
        self.c_late = c_late

    def cost(self, signal, labels, windows, thr):
        pred = get_pred(signal, thr)
        tp, fp, fn, adv = get_metrics(pred, labels, windows)
        adv_det = [a for a in adv if a.total_seconds() <= 0]
        cost = self.c_alarm * len(fp) + \
            self.c_missed * len(fn) + \
            self.c_late * (len(adv_det))
        return cost
```

# Refactoring the Cost Model

**We can now evaluate our cost by:**

- Instantiating the class with our values for  $C_{alarm}$ ,  $C_{missed}$ ,  $C_{late}$
- Calling the `cost` function with different threshold values

```
In [18]: cmodel = nab.ADSimpleCostModel(c_alarm, c_missed, c_late)
cost = cmodel.cost(signal, labels, windows, thr)
print(f'The cost with the current predictions is: {cost}')
```

The cost with the current predictions is: 43

**The process is rather efficient:**

- When we change the threshold
- ...We do not need to rebuild the alarm signal

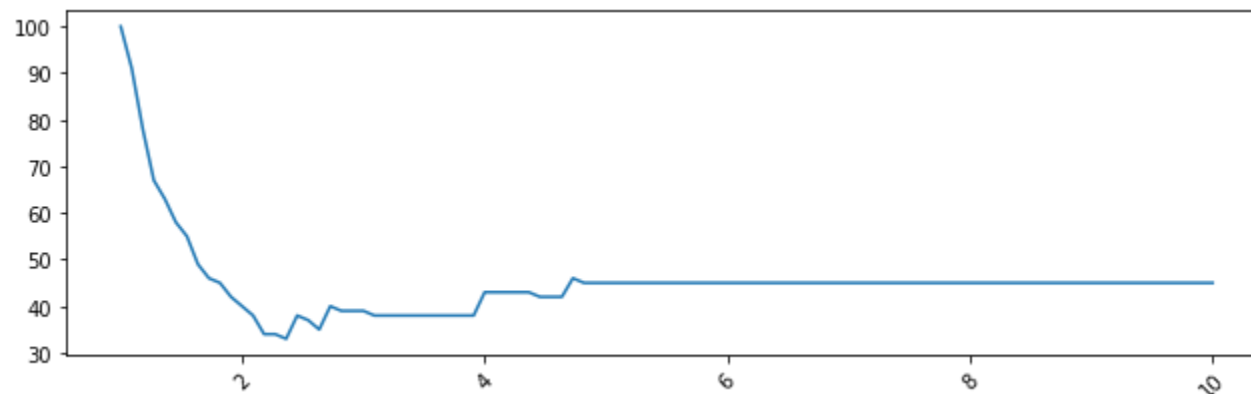
So, no need to repeat either training or inference

# Effect of the Threshold

We can now look (over all data) at the effect of changing the threshold:

- This is the response surface or cost function landscape
- ...For an idealized problem (i.e. over all the data)

```
In [19]: cmodel = nab.ADSimpleCostModel(c_alarm, c_missed, c_late)
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)
```



# Choosing the Threshold

**Ideally, we wish to choose the best threshold**

However, we have two problems:

- We cannot really use all the data
- ...we need to keep some away to test generalization
- The threshold-to-cost function is non-smooth and non-differentiable

**Luckily, both are easy to address**

We can:

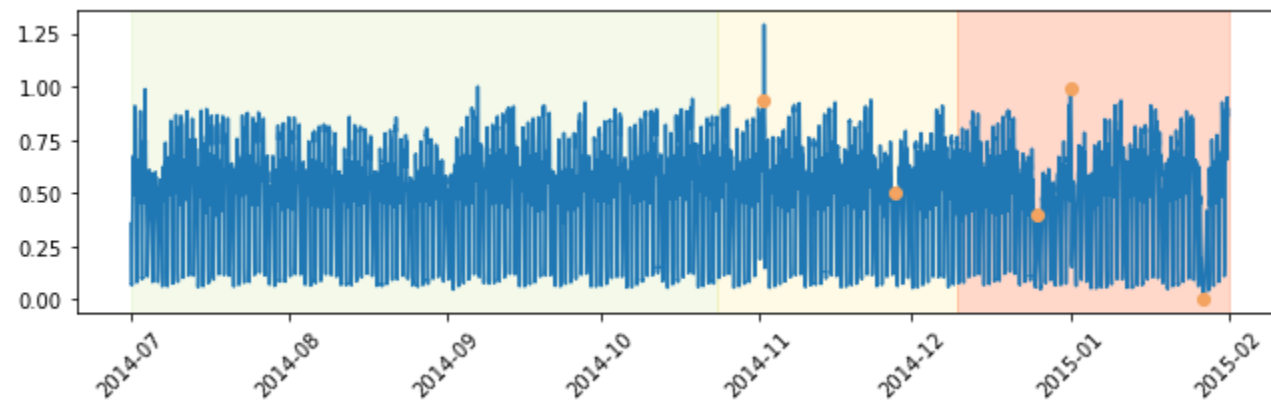
- Define a validation set to use when optimizing the threshold
- Use a simple line search approach, since  $\theta$  is a scalar

# Define a Validation Set

**First, we define our validation set:**

- It should contain some anomalies
- ...But some should be left out (for testing)

```
In [20]: val_end = pd.to_datetime('2014-12-10 00:00:00')  
nab.plot_series(data, labels, val_start=train_end, test_start=val_end)
```



# Optimize the Threshold

**We will use a line sampling approach:**

We build the validation set and define a range of "sampled" thresholds:

```
In [21]: signal_opt = signal[signal.index < val_end]
labels_opt = labels[labels < val_end]
windows_opt = windows[windows['end'] < val_end]
thr_range = np.linspace(0, 10, 100)
```

- Notice that we included the training data in the validation set
- ...This is ok, since we are not using the validation set to prevent overfitting

**Then, we use a function from our module to pick the best one:**

```
def opt_thr(signal, labels, windows, cmodel, thr_range):
    costs = [cmodel.cost(signal, labels, windows, thr)
              for thr in thr_range]
    costs = np.array(costs)
    best_idx = np.argmin(costs)
    return thr_range[best_idx], costs[best_idx]
```



# Optimize the Threshold

Let's see the results:

```
In [22]: 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: 4.747474747474747, corresponding cost: 15

- The reported cost is on the training set

**For all the data (yes, we are cheating a bit) we have:**

```
In [23]: ctst = cmodel.cost(signal, labels, windows, best_thr)
print(f'Cost on the whole dataset {ctst}')
```

Cost on the whole dataset 45

- This is, as expected, **suboptimal**
- ...But it works! We have our **first complete anomaly detection approach**