In [1]:
```
# ============================================================
# Notebook setup: run this before everything
# ============================================================

%load_ext autoreload
%autoreload 2

# Control figure size
figsize=(14, 5)

from sklearn.neighbors import KernelDensity
from util import util
import numpy as np
from matplotlib import pyplot as plt
import os

data_folder = os.path.join('..', 'data', 'nab')
file_name = os.path.join('realKnownCause', 'nyc_taxi.csv')
data, labels, windows = util.load_series(file_name, data_folder)
```

A Formalization for Anomaly Detection

How do we formalize our problem?

Problem Formalization

A possible approach: we characterize the data distribution

If we can *estimate the probability* of every occurring observation x

- We can choose a size for the car pool
- ...Then we can spot anomalies based on their *low probability*

After all, anomalies are *rare events*, by definition

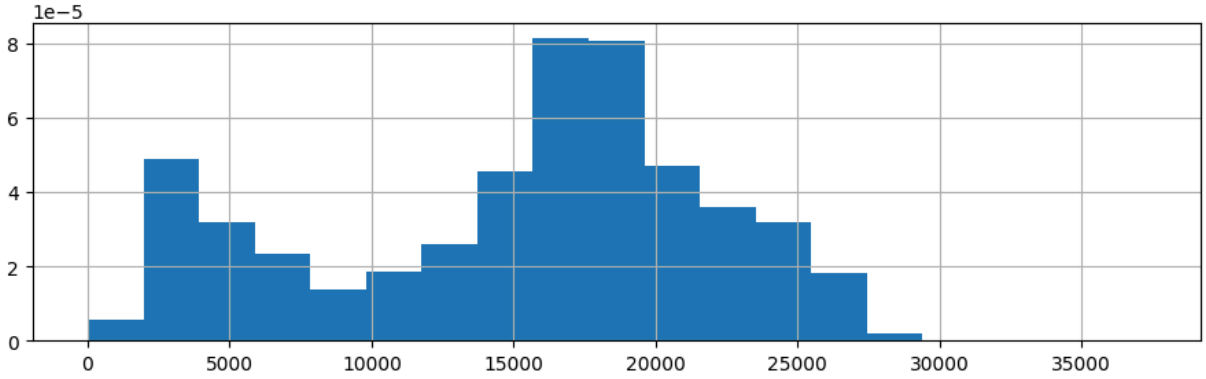
> **We turn a liability into a strenght!**

Problem Formalization

We can check our intuition on our data

This is (roughly) the distribution *over all the data*

In [2]:
```
vmax = data['value'].max()
util.plot_histogram(data['value'], vmax=vmax, bins=20)
```

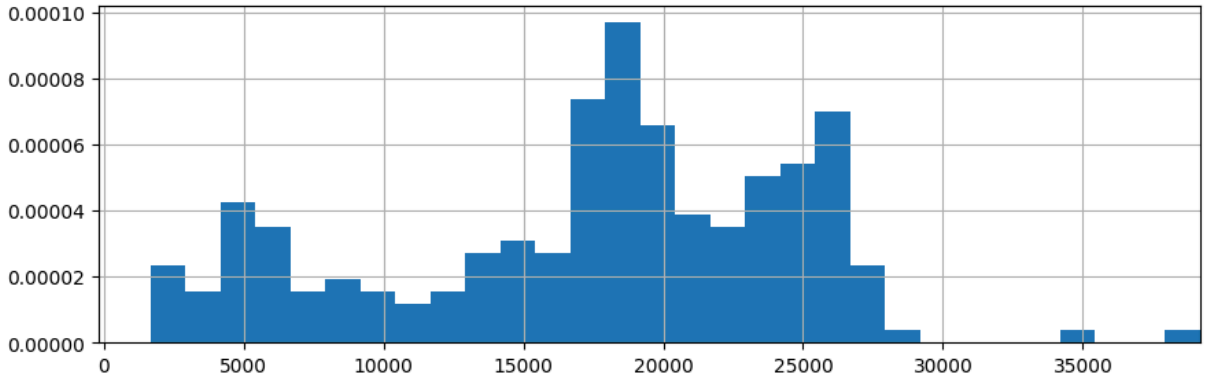


Problem Formalization

We can check our intuition on our data

This is (roughly) the distribution *around the first anomaly*:

In [9]:
```
w0_start, w0_end = windows.loc[0]['begin'], windows.loc[0]['end']
data_anomaly0 = data[(data.index >= w0_start) & (data.index < w0_end)]
util.plot_histogram(data_anomaly0['value'], vmax=vmax, bins=30)
```



- It seems indeed that there's a significant difference

What is the next step?

Problem Formalization

When we reach this stage, it's a good idea for *formalize* our problem

We can characterize a continuous distribution via its *density*

- Given a random variable X with values x
- ...We care about its *Probability Density Function* $f(x)$

Since anomalies are assumed to be unlikely

...Our detection condition can be stated as:

$$f(x) \leq \varepsilon$$

- Where ε is a (scalar) threshold

What do we need to make this work?

Density Estimation

We need one way to *estimate probability densities*

For some random process with n-dimensional variable x:

- Given the true density function $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^+$
- ...And a second function $\hat{f}(x, \theta)$ with the same input, and parameters θ

We want to *make the two as similar as possible*

Can we obtain that using supervised learning?

Given some suitable loss function $L(y, \hat{y})$, we would need to solve:

$$\text{argmin}_\theta L(\hat{f}(x, \theta), f^*(x))$$

- where x represents the training data

Density Estimation

Unfortunately, this approach *cannot work*

...Because typically *we do not have access* to the true density f^*

> **Density estimation is an *unsupervised* learning problem**

It can be solved via a number of techniques:

- Simple histograms
- Kernel Density Estimation
- Gaussian Mixture Models
- Normalizing Flows
- Non Volume Preserving (NVP) transformations

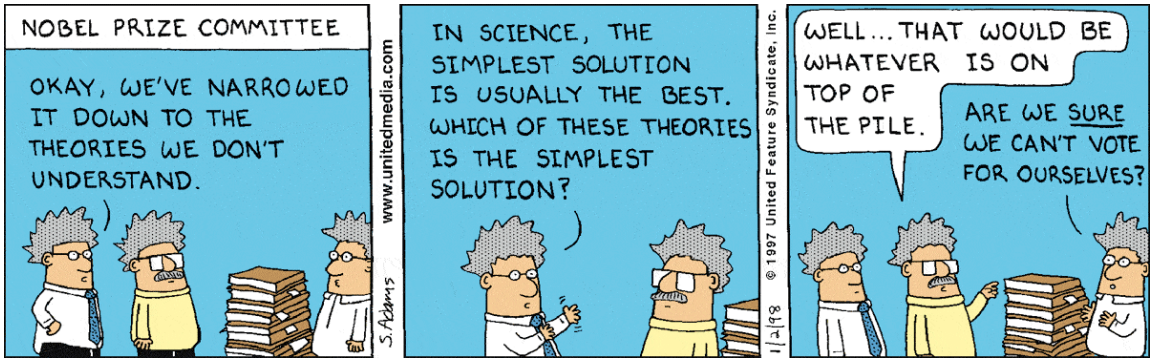
Which one shall we pick?

Our Friend, Occam's Razor

We will go with Occam's razor

It's a philosophical principle stating that:

> **Between two hypotheses, the simpler one is usually correct**



Our Friend, Occam's Razor

We will go with Occam's razor

It's a philosophical principle stating that:

> **Between two hypotheses, the simpler one is usually correct**

In practice, it's often a good idea to *start with a simple approach*

- If it works well, then you have *a solution*
- If it does not, they you have *a baseline*

In both cases, you win!

For this example, we will pick *Kernel Density Estimation*

Kernel Density Estimation

Kernal Density Estimation

In *Kernel Density Estimation (KDE)*, the main idea is that:

- Wherever (in input space) there is a sample
- ...It's likely that there are more

So, we assume that *each training sample is the center for a density "kernel"*

Formally, suck kernel $K(x, h)$ is just a valid PDF:

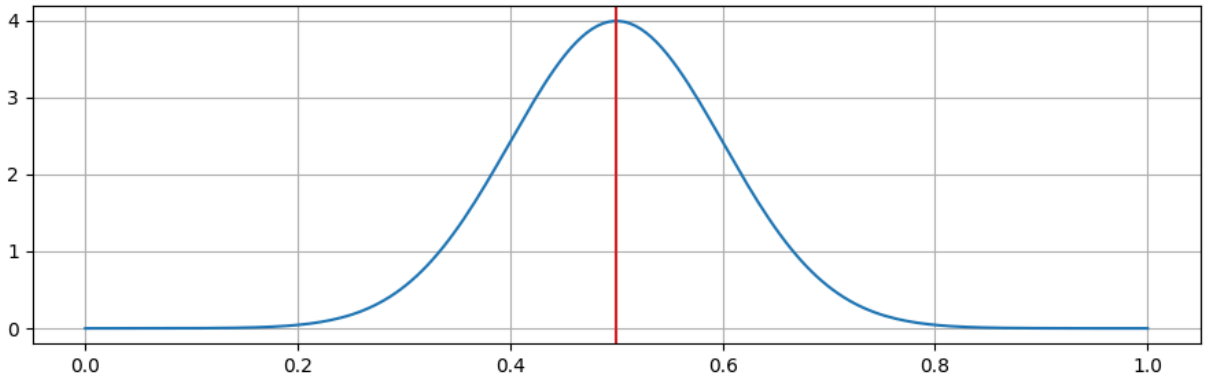
- x is the input variable (scalar or vector)
- h is a parameter (resp. scalar or matrix) called *bandwidth*

Typical kernels: Gaussian, exponential, cosine, linear...

Kernels

An example with one sample and a *Guassian* kernel:

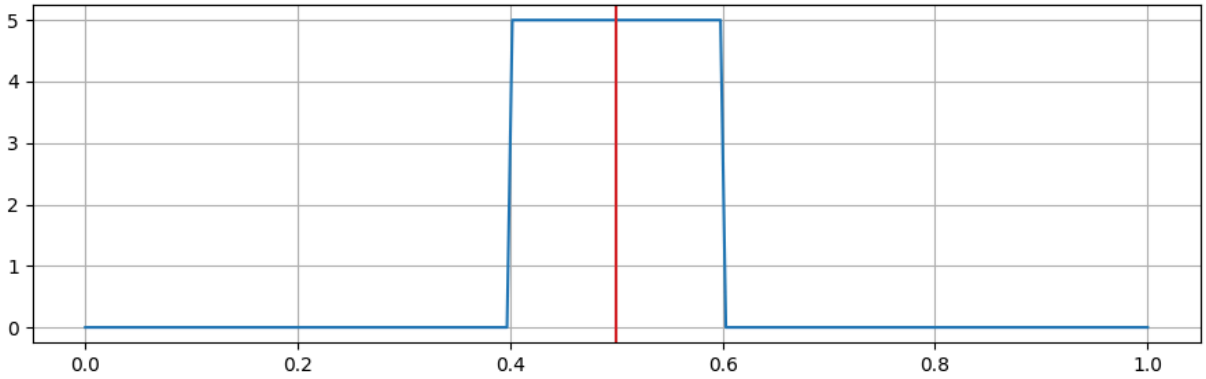
In [10]:
```
x = np.array(0.5).reshape(1,1) # single sample
kde = KernelDensity(kernel='gaussian', bandwidth=0.1) # build the estimator
kde.fit(x) # fit the estimator on the data
# We use a plotting function from our module
util.plot_density_estimator_1D(kde, xr=np.linspace(0, 1, 200))
ymin, ymax = plt.ylim()
plt.vlines(x, ymin, ymax, color='tab:red')
plt.ylim((ymin, ymax)); # ; = suppress output
```



Kernels

An example with one sample and a *Tophat* kernel:

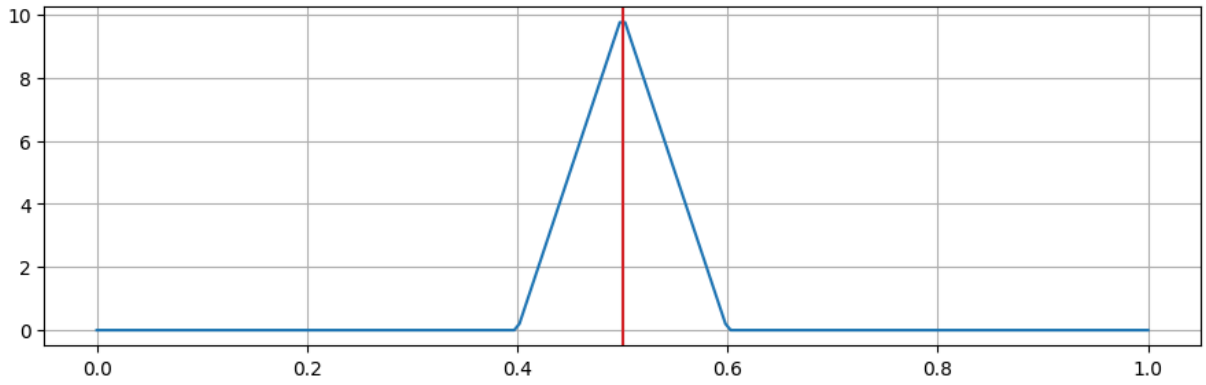
In [11]:
```
x = np.array(0.5).reshape(1,1) # single sample
kde = KernelDensity(kernel='tophat', bandwidth=0.1) # build the estimator
kde.fit(x) # fit the estimator on the data
# We use a plotting function from our module
util.plot_density_estimator_1D(kde, xr=np.linspace(0, 1, 200))
ymin, ymax = plt.ylim()
plt.vlines(x, ymin, ymax, color='tab:red')
plt.ylim((ymin, ymax)); # ; = suppress output
```



Kernel

An example with one sample and a *linear* kernel:

In [12]:
```
x = np.array(0.5).reshape(1,1) # single sample
kde = KernelDensity(kernel='linear', bandwidth=0.1) # build the estimator
kde.fit(x) # fit the estimator on the data
# We use a plotting function from our module
util.plot_density_estimator_1D(kde, xr=np.linspace(0, 1, 200))
ymin, ymax = plt.ylim()
plt.vlines(x, ymin, ymax, color='tab:red')
plt.ylim((ymin, ymax)); # ; = suppress output
```



Kernels

As an example, a *Gaussian kernel* in sklearn is given by:

$$K(x, h) \propto e^{-\frac{x^2}{2h^2}}$$

- The $\propto$ ("proportional to")

The function is similar to a the PDF of the Normal distribution:

- The mean can be interpreted as 0
- h plays the role of the standard deviation
- ...And scikit learn handles normalization

Kernel Re-centering

Since the "mean" is 0, the kernel is *centered on 0*

All kernels in KDE are by default *zero-centered*

- ...But we need to place then over each sample
- How can this be done?

We can use an *affine transformation* (like in the scale/location trick)

In practice, the expression:

$$K(x - \mu, h)$$

- ...Gives the value the value of a kernel *centered on* μ
- ...Computed for the value x

Kernel Density Estimation

The estimated density of any point is obtained as a *kernel average*:

$$f(x, \bar{x}, h) = \frac{1}{m} \sum_{i=0}^{m} K(x - \bar{x}_i, h)$$

- x is the input for which we want an estimate
- $\bar{x}_i$ is sequence of the m training samples
- $x - \bar{x}_i$ is the difference between x and the i-th training sample

By changing the kernel function:

- We can adjust the properties of the distribution (e.g. smoothness)
- Typically, the choice is based on prior domain knowledge

Kernel Density Estimation

KDE models are not trained in the usual sense

...But they store internally *all the training samples*

- I.e. the training set is *part of the model parameters*
- This is a property common to most kernel models

There is one thing that we need to train, i.e. the bandwidth h

- We will see a general approach later in the course
- ...But in the *univariate* case we can apply a rule of thumb:

$$h = 0.9 \min\left(\hat{\sigma}, \frac{IQR}{1.34}\right) m^{-\frac{1}{5}}$$

Where IQR is the inter-quartile range

Kernel Density Estimation

An example with two samples and a Guassian kernel:

In [13]:
```
x = np.array([0.25, 0.75]).reshape(-1,1) # two sample, univariate
kde = KernelDensity(kernel='gaussian', bandwidth=0.1) # build the estimator
kde.fit(x) # fit the estimator on the data
util.plot_density_estimator_1D(kde, xr=np.linspace(0, 1, 200))
ymin, ymax = plt.ylim()
plt.vlines(x, ymin, ymax, color='tab:red')
plt.ylim((ymin, ymax)); # ; = suppress output
```