Artificial Intelligence in Industry

Matteo Donati

February 24, 2023

Contents

1 Anomaly Detection via Simple Methods

1.1 Problem and Data

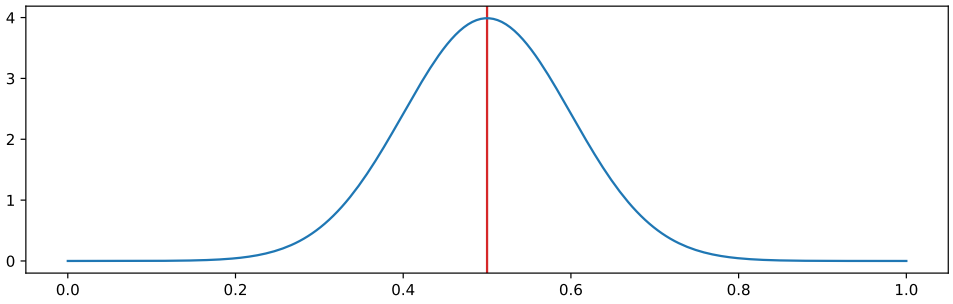
The goal of anomaly detection is to detect, analyze and anticipate abnormal situations (i.e. **anomalies**). Usually, anomaly detection is based on time-series analysis, where a time-series is a sequence whose index represents time. Time-series have one difference with respect to classical table datasets: their row index is meaningful, since it represents the position of the example in the sequence. The labels associated with such a dataset usually indicate an instant of time at which an anomaly occurs. Considering a specific anomaly, one could also compute the window of time inside which the specific anomaly occurs.

1.2 Anomaly Detection and Kernel Density Estimation

A possible approach to detect anomalies is based on the fact that anomalies are often unlikely. If one can estimate the probability of every occurring observation x, then one can spot anomalies based on their low probability. Formally, a detection condition can be states as $f(x) \leq \theta$, where $f(x)$ is a **probability density function** (**PDF**), and θ is a scalar threshold. Given some training data $\hat{\boldsymbol{x}}$, the true density function $f^*(x) : \mathbb{R}^n \to \mathbb{R}^+$, and a second function $f(x, \omega)$, a supervised learning approach to estimate the probability densities considers a suitable loss function, $L(y, y^*)$, that has to be optimized so to find the best set of parameters ω that minimizes the considered loss:

$$\text{argmin}_\omega \, L(f(\hat{\boldsymbol{x}}, \omega), f^*(\hat{\boldsymbol{x}})) \tag{1}$$

However, this approach cannot work, because usually one does not have access to the true density f^*. Thus, density estimation is an unsupervised learning problem. Such problem can be solves via a number of techniques (e.g. via Kernel Density Estimation). In **Kernel Density Estimation** (**KDE**) the main idea is that wherever, in the input space, there is a sample, then it is likely that there are more samples, so one can assume that each training sample is the center for a density kernel. Formally, the kernel $K(x, h)$ is just a valid PDF, where x is the input variable (scalar or vector), and h is a parameter (scalar or matrix respectively) called *bandwidth*. For example, given a single sample $x = 0.5$, then a Gaussian estimator with $h = 0.1$ will produce the following:



Indeed, in `sklearn`, a Gaussian kernel is given by:

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

which is similar to the PDF of the Normal distribution, where the mean can be interpreted as zero, and h controls the standard deviation of the distribution. However, since the mean is zero, the kernel will be centered on zero. To solve this, one can use an affine transformation, $K(x - \mu, h)$, which gives the value of a kernel computed for the value x and centered on μ. Moreover, the estimated density of any point is obtained as a kernel average:

$$f(x, \hat{\boldsymbol{x}}, h) = \frac{1}{m} \sum_{i=0}^{m} K(x - \hat{x}_i, h) \tag{3}$$

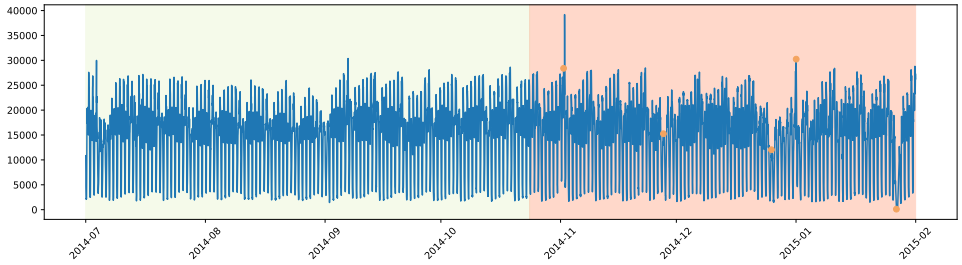
where x is the input for which to compute the estimate, $\hat{\boldsymbol{x}}$ is the matrix containing the training samples, $x - \hat{x}_i$ is the difference between x and the i-th training sample. Thus, KDE models are not trained in the usual sense: the training set is part of the model parameters. The only thing that one needs to train is h. For the univariate case, one can apply the following rule of thumb:

$$h = 0.9 \min\left(\hat{\sigma}, \frac{IQR}{1.34}\right) m^{-\frac{1}{5}} \tag{4}$$

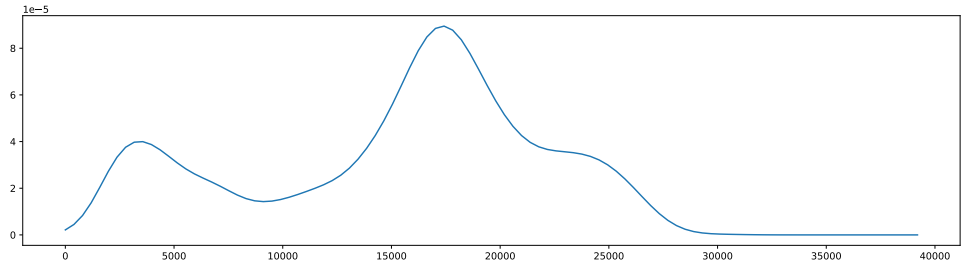
where $\hat{\sigma}$ is the standard deviation computed using the training data, and IQR is the inter-quartile range. Lastly, to avoid taking products, one can work with negated log probabilities, so that the anomaly detection condition becomes $-\log f(x, \omega) \geq \theta$.

1.3 KDE for Anomaly Detection

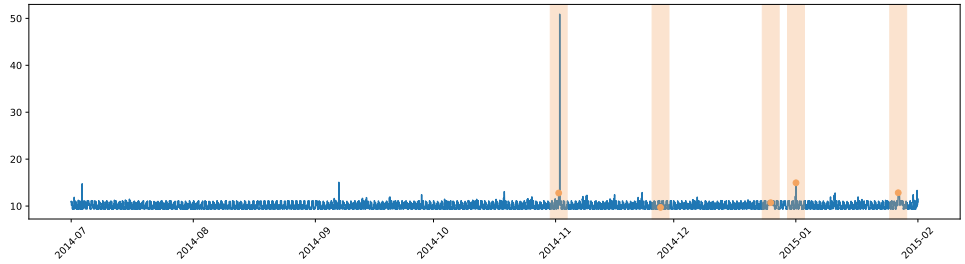
Considering a time-series, one can split it into training set and test set, where the training set only includes data about normal behavior and it will be used to fit a KDE model, while the test is used to asses how well the approach can generalize. If the training set contains some anomalies, then it is fine, as long as these are very infrequent.



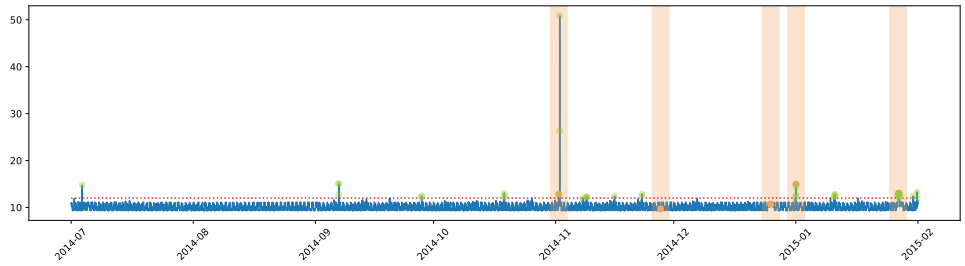
At this point, a univariate KDE is fit to the training data, obtaining the following estimated distribution:



The alarm signal can be then computed from such estimated distribution:



One could also pick a threshold and try to detect some anomalies:



However, the result contains many false positives, which are usually common in anomaly detection.

1.4 Metrics for Anomaly Detection

In order to evaluate costs and benefits of one's predictions, one can rely on a cost model. A simple cost model is based on the concepts of true positives (i.e. windows for which one detects at least one anomaly), false positives (i.e. detected anomalies that do not fall in any window), false negatives (i.e. anomalies that go undetected), and advance (i.e. time between an anomaly and when first it was detected). Then, one can introduce: a cost c_{alarm} for loosing time in analyzing false positives, a cost c_{missed} for missing anomalies, and a cost c_{late} for a late detection. This simple cost model can be used for choosing the threshold to detect anomalies. Indeed, the best threshold is the one that minimizes

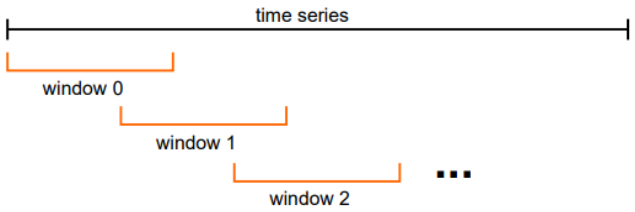
$c_{alarm} \times |FP| + c_{missed} \times |FN| + c_{late} \times |\text{late detections}|$. To do so, one can define a validation set, and apply a linear search, to tune the θ threshold.

1.5 Sliding Windows

Given a time-series, nearby points tend to have similar values, meaning that they are correlated. A useful tool to study such correlation are the autocorrelation plots:

1. Consider a range of possible **lags**.
2. For each lag value l:
   (a) Make a copy of the series and shift it by l time-steps.
   (b) Compute the Pearson correlation coefficient (i.e. linear correlation coefficients) with the original series.
3. Plot the correlation coefficients over the lag values.

Where the curve is far from zero, there is a significant correlation, and where it gets close to zero, no significant correlation exists. These correlations are a source of information and can be exploited to improve the estimated probabilities. Indeed, to take advantage of such information, one could feed one's model with sequences of observations, instead of individual observations. A common approach consist in using a **sliding window**:



In general, let m be the number of examples and w be the window length, the result of this approach is the table

	$\boldsymbol{s_0}$	$\boldsymbol{s_1}$	$\cdots$	$\boldsymbol{s_{w-1}}$
$\boldsymbol{t_{w-1}}$	x_0	x_1	$\cdots$	x_{w-1}
$\boldsymbol{t_w}$	x_1	x_2	$\cdots$	x_w
$\boldsymbol{t_{w+1}}$	x_2	x_3	$\cdots$	x_{w+1}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\boldsymbol{t_{m-1}}$	x_{m-w}	x_{m-w+1}	$\cdots$	x_{m-1}

where $\boldsymbol{t_i}$ is the time window index, and $\boldsymbol{s_j}$ is the position of an observation within a window. In general, one can pick the window length to be equal to the number of lags for which the aforementioned correlation is still high.

1.6 Sequence Input in KDE

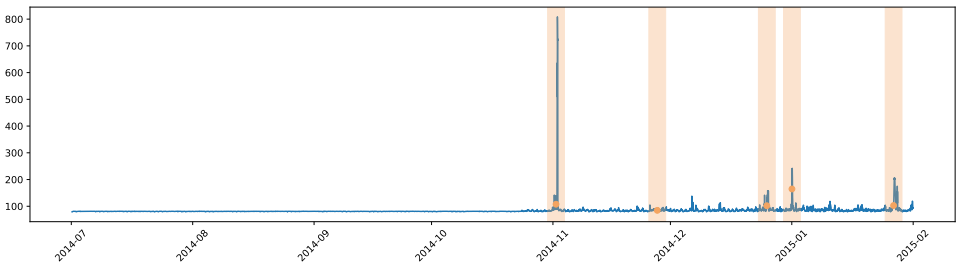
When dealing with KDE, there exists a straightforward approach to take sequences input into account. In particular, this can be done by using multivariate KDE: each sequence is treated as a vector variable, and individual sequences are treated as independent (Markov property). Then, a multivariate KDE estimator is learned. First of all, a suitable bandwidth has to be chosen: i. pick a validation set, ii. tune th bandwidth for maximum likelihood. Formally, let $\tilde{\boldsymbol{x}}$ be a validation set of m samples. Assuming independent observations, its likelihood is:

$$L(\tilde{\boldsymbol{x}}, \hat{\boldsymbol{x}}, h) = \prod_{i=1}^{m} f(\tilde{x}_i, \hat{\boldsymbol{x}}, h) \tag{5}$$

Then, h is chosen to maximize such likelihood:

$$\text{argmax}_h \, \mathbb{E}_{\tilde{\boldsymbol{x}} \sim \mathcal{D}}[L(\tilde{\boldsymbol{x}}, \hat{\boldsymbol{x}}, h)] \tag{6}$$

where $\mathcal{D}$ is the true distribution of samples. However, as many training problems, this cannot be solves in an exact fashion. Instead, one can approximate $\mathbb{E}$ by sampling multiple $\tilde{\boldsymbol{x}}$ and picking the bandwidth h^* leading to the maximum average likelihood. This approximation can be implemented by applying a grid search and searching for h^*. For example, the alarm signal produced when considering sequences input instead of single observations is the following:

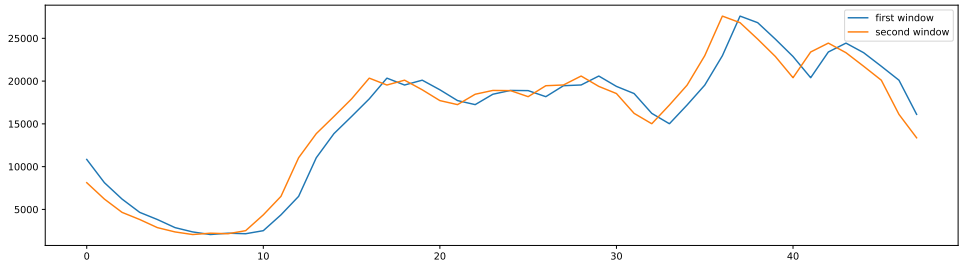


This signal shows much less noise with respect to the one computed when considering single observations.

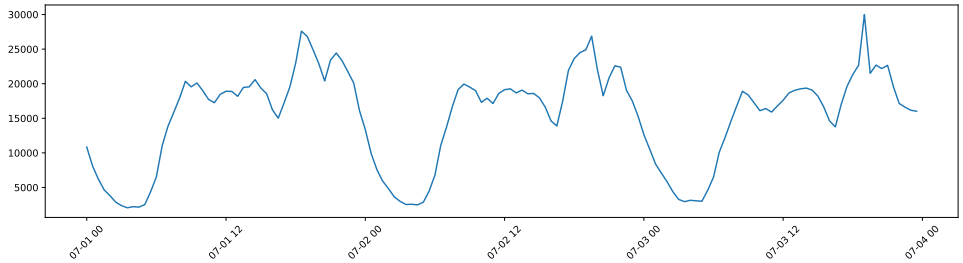
2 Anomaly Detection via Advanced Methods

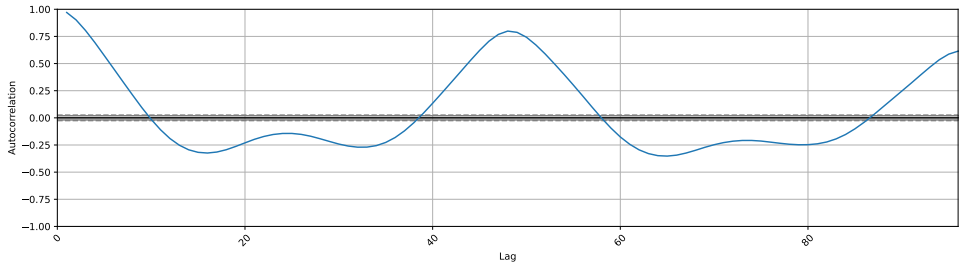
2.1 A Time-Dependent Estimator

Considering a sequence-based estimator, one can notice how this estimator learns from all the training data. This means, for example, that considering two subsequent windows it will learn from both these subsequent sequences:



In particular, in the first window, the observations are x_0, x_1, ..., while in the second window the observations are x_1, x_2, ..., which means that by moving the window forward one learn the distribution of each point (and its correlations) multiple times. Moreover, it could happen that the considered time-series is approximately periodic:



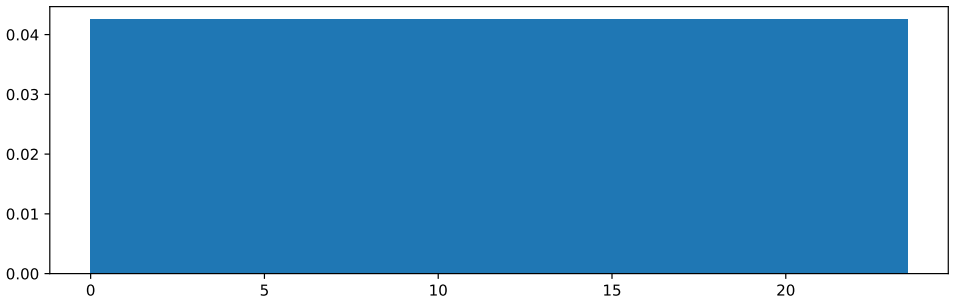


Thus, the previous sequence-based estimator is solving a uselessly complicated problem, and it is not using all the available knowledge. In order to solve these two problems, one could start by adding the

notion of time to the input: $y = (t, x)$, where t represents time and x is the usual input. One can then use this information to build a time-dependent estimator $f(x|t)$. This is a **conditional density**, namely the density value of the observed value of x assuming that the time t is known (indeed, t is a **controlled variable**, i.e. it is completely predictable). Thus, the anomaly detection conditions becomes $f(x|t) \leq \theta$. Considering the definition of conditional probability, $f(t, x) = f(x|t)f(t)$, one can derive:

$$\frac{f(t, x)}{f(t)} \leq \theta \tag{7}$$

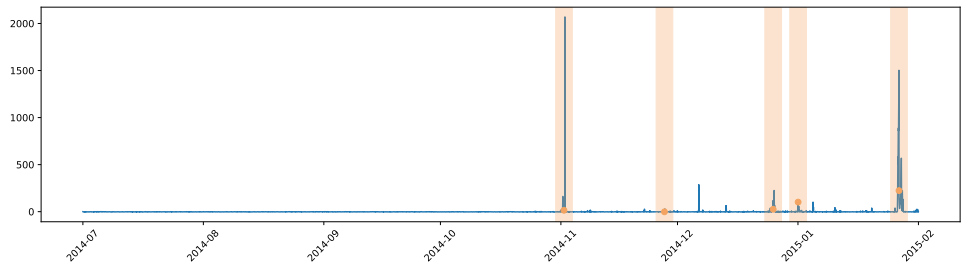
In order implement this, one needs an estimator for $f(t, x)$ (e.g. using KDE), and one estimator for $f(t)$ which one can easily obtain (e.g. using KDE again).



From equation 7:

$$f(t, x) \leq \theta f(t) \quad \xRightarrow{\text{Being } f(t) \text{ constant}} \quad f(t, x) \leq \theta' \tag{8}$$

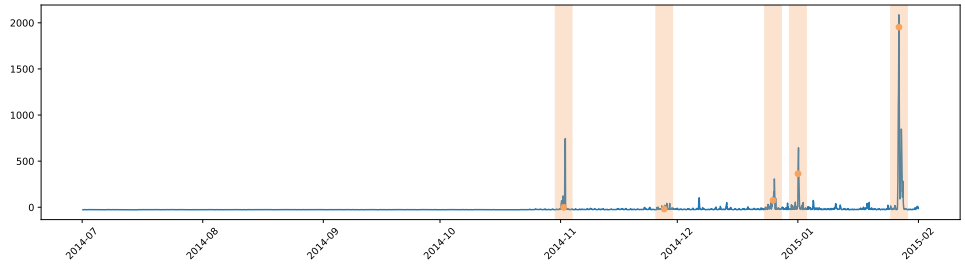
Now, having chosen a specific bandwidth through grid-search, one can produce the specific alarm signal:



Lastly, one can search for the best threshold θ' to minimize the overall cost given by the specific cost model discussed above.

2.2 Time-Indexed Models

A second approach to handle time consists in learning many density estimators, where each estimator is specialized for a given time. Formally, this is an ensemble model. In particular, one can obtain the estimated probabilities by evaluating $f_{g(t)}(x)$, where each f_i function is an estimator, and the $g(t)$ retrieves the correct f_i based on the time value. Each f_i estimator works with smaller amounts of data, but the individual problems are easier. For example, given a range of time of 24 hours, one could learn a different estimator for 00:00, 00:30, 01:00, etc. producing 48 specialized estimators, where each estimator is learned using a different portion of the training set. Lastly, one can estimate the log probabilities for each possible timestamp and concatenate the results so to produce the alarm signal



and then to search for the best possible threshold.

2.3 Gaussian Mixture Models

KDE-based approaches work well, but have some trouble with high-dimensional data: with a larger dimensionality, prediction time grows and more data is needed to obtain reliable results. Moreover, KDE has trouble with large training sets, and gives nothing more than an anomaly signal (i.e. determining the cause of the anomaly is up to a domain expert). The first two problems are due to the fact that KDE makes no attempt to compress the information from the training data. Indeed, the size of a KDE model grows directly with the training set size. To solve such problem, another density estimation technique is introduced: the **Gaussian Mixture Models** (**GMMs**). A GMM describes a distribution via a weighted sum of Gaussian components, where the model size depends on the dimensionality and on the number of components, and the number of components can be chosen. Formally, one can assume data is generated by a probabilistic model

$$X_Z \tag{9}$$

where Z and X_k are both variables. In particular, Z represents the index of the component that generates the sample, and X_k follows a multivariate Gaussian distribution. In other words, a GMM is a selection-based ensemble. The probability density function (PDF) of a GMM is given by:

$$g(x, \mu, \Sigma, \tau) = \sum_{k=1}^{n} \tau_k f(x, \mu_k, \Sigma_k) \tag{10}$$

where f is the PDF of a multivariate Normal distribution, μ_k is the vector mean and Σ_k is the covariance matrix for the k-th component, and τ_k corresponds to $P(Z = k)$. When one wants to sample from a GMM, first one needs to sample the Z variable, then one can sample from the corresponding multivariate distribution. Hence, one does not get to know just the sample value, but also which of the Gaussian components it was generated by. Training a GMM to approximate other distributions can be done in terms of likelihood maximization:

$$\text{argmax}_{\mu,\Sigma,\tau}\, \mathbb{E}_{\hat{x}\sim X}[L(\hat{x}, \mu, \Sigma, \tau)] \quad \text{s.t.} \quad \sum_{k=1}^{n} \tau_k = 1 \tag{11}$$

The likelihood function L measures how likely it is that the training sample $\hat{x}$ is generated by a GMM with parameters μ, Σ, τ. This expectation can be approximated by using the training set:

$$\mathbb{E}_{\hat{x}\sim X}[L(\hat{x}, \mu, \Sigma, \tau)] \simeq \prod_{i=1}^{m} g(\hat{x}_i, \mu, \Sigma, \tau) \tag{12}$$

This leads to:

$$\text{argmax}_{\mu,\Sigma,\tau} \prod_{i=1}^{m} \sum_{k=1}^{n} \tau_k f(x, \mu_k, \Sigma_k) \quad \text{s.t.} \quad \sum_{k=1}^{n} \tau_k = 1 \tag{13}$$

From an optimization point of view, this is an annoying problem, mainly due to the presence of a constant, a product and a sum, and the fact that the product cannot be decomposed. So, in order to simplify such formulation, a random variable Z_i is introduced for each example. In particular, $Z_i = k$ if and only if the i-th example was drawn from the k-th component, and the Z_i variables are latent (i.e. their value is unknown). With the new variables, the PDF becomes:

$$\tilde{g}_i(x_i, z_i, \mu, \Sigma, \tau) = \tau_{z_i} f(x, \mu_k, \Sigma_k) \tag{14}$$

The PDF is now specific for each example and does not contain a sum. Moreover, the value z_i is now an input to $\tilde{g}_i$ can be used as an index to retrieve the correct τ_k. The likelihood expectation over both X and Z, $\mathbb{E}_{\hat{x}\sim X,\hat{z}\sim Z}[L(\hat{x}, \hat{z}, \mu, \Sigma, \tau)]$ can be computed by using the training set as a single sample so to obtain:

$$\mathbb{E}_{\hat{x}\sim X,\hat{z}\sim Z}[L(\hat{x}, \hat{z}, \mu, \Sigma, \tau)] \simeq \mathbb{E}_{\hat{z}\sim Z}\left[\prod_{i=1}^{m} \tilde{g}_i(x_i, z_i, \mu, \Sigma, \tau)\right] \tag{15}$$

The same technique cannot be used for Z, since the values of the Z_i variables are unknown. To deal with the expectation on Z, another set of variables is added. These variables represent the unknown distribution of the latent Z_i variables. In particular, $\tilde{\tau}_{i,k}$ corresponds to $P(Z_i = k)$. With the new variable, the expectation can be computed in closed form:

$$\mathbb{E}_{\hat{x}\sim X,\hat{z}\sim Z}[L(\hat{x}, \hat{z}, \mu, \Sigma, \tau)] \simeq \prod_{i=1}^{m} \prod_{k=1}^{n} \tilde{g}_i(x_i, z_i, \mu, \Sigma, \tau)^{\tilde{\tau}_{i,k}} \tag{16}$$

Intuitively, $\tilde{\tau}_{i,k}$ samples are generated for each example i and component k. Then, their densities are multiplied as usual. Thus, the training problem is:

$$\begin{aligned} &\text{argmax}_{\mu,\Sigma,\tau,\tilde{\tau}} \prod_{i=1}^{m}\prod_{k=1}^{n} \tilde{g}_i(x_i, z_i, \mu, \Sigma, \tau)^{\tilde{\tau}_{i,k}} \\ &\text{s.t.} \sum_{k=1}^{n} \tau_k = 1 \\ &\text{s.t.} \sum_{k=1}^{n} \tilde{\tau}_{i,k} = 1 \quad \forall i \in \{1, \ldots, m\} \end{aligned} \tag{17}$$

The expectation-maximization algorithm can be now used. This algorithm is an optimization method based on alternating steps:

- In the expectation step:
  - μ, Σ, τ are considered as fixed and one can optimize over $\tilde{\tau}$.
  - The expectation over Z is computed in a symbolic form.
- In the maximization step:
  - $\tilde{\tau}$ is considered as fixed and one can optimize over μ, Σ, τ.

Such method stops when the likelihood improvement become too small. When considering the aforementioned optimization problem, these two steps are defined as follows:

- In the expectation step, the μ, Σ, τ are fixed, so that one needs to solve:

$$\text{argmax}_{\tilde{\tau}} \prod_{i=1}^{m}\prod_{k=1}^{n} \tilde{g}_i(x_i, z_i, \mu, \Sigma, \tau)^{\tilde{\tau}_{i,k}} \quad \text{s.t.} \quad \sum_{k=1}^{n} \tilde{\tau}_{i,k} = 1 \quad \forall i \in \{1, \ldots, m\} \tag{18}$$

  Such optimization problem can be easily decomposed, so one can optimize over each example individually. By substituting $\tilde{g}_i$ for a single example i one has:

$$\text{argmax}_{\tilde{\tau}} \prod_{k=1}^{n} (\tau_k f(x, \mu_k, \Sigma_k))^{\tilde{\tau}_{i,k}} \quad \text{s.t.} \quad \sum_{k=1}^{n} \tilde{\tau}_{i,k} = 1 \tag{19}$$

  Which (since μ, Σ, τ are fixed) is solved by choosing:

$$\tau_{i,k} = \frac{\tau_k f(\hat{x}_i, \mu_k, \Sigma_k)}{\sum_{h=1}^{n} \tau_h f(\hat{x}_i, \mu_k, \Sigma_k)} \tag{20}$$

- For the maximization step the math is a bit more difficult. Each τ_k is optimized by computing relative sum of the corresponding $\tilde{\tau}_{i,k}$ variables:

$$\tau_k = \frac{1}{m} \sum_{i=1}^{m} \tilde{\tau}_{i,k} \tag{21}$$

  In fact, the latent variables represent samples drawn from the Z_k variables. Lastly, the μ_k and Σ_k parameters can be estimated based on classical methods. In particular, each example is given a weight equal to $\tilde{\tau}_{i,k}$, then μ and Σ are estimated via a least square approach.

2.4 Autoencoders for Anomaly Detection

An **autoencoder** is a type of neural network which is usually designed to reconstruct its input vector by first learning an internal representation of the input (encoder), and then by learning how to reconstruct the input starting from such representation (decoder). Formally, an encoder $e(x, \theta_e)$ maps x into a vector of latent variables z, and a decoder $d(z, \theta_d)$ maps z into the reconstructed input tensor. In general, autoencoders are trained for minimum MSE:

$$\text{argmin}_{\theta_e, \theta_d} \| d(e(\hat{x}_i, \theta_e), \theta_d) \|_2^2 \tag{22}$$

Usually, there is a risk that an autoencoder learns a trivial transformation (i.e. $x' = x$), which can be avoided by choosing a small-dimensional latent space, and by encouraging sparse encodings with an L1 regularizer. Moreover, autoencoders can be used for anomaly detection by using the reconstruction as an anomaly signal, e.g.:

$$\| x - d(e(\hat{x}_i, \theta_e), \theta_d) \|_2^2 \geq \theta \tag{23}$$

This approach has some pros and cons compared to KDE. Indeed, the size of a neural network does not depend on the size of the training set, and neural networks have good support for high dimensional data. On top of this, there is a limited possibility of overfitting and time needed for prediction/detection is low. On the other hand, error reconstruction can be harder than density estimation. Moreover, the results obtained by using an autoencoder are similar to ones obtained when using KDE. Indeed, given an autoencoder h, one tries to solve the following problem:

$$\text{argmin}_{\theta} \| h(\hat{x}_i, \theta) - \hat{x}_i \|_2^2 \tag{24}$$

By expanding the L2 norm:

$$\text{argmin}_{\theta} \sum_{i=1}^{m} \sum_{j=1}^{n} (h_j(\hat{x}_i, \theta) - \hat{x}_{i,j})^2 \tag{25}$$

By introducing a log and exp transformation:

$$\text{argmin}_{\theta} \log \exp \left(\sum_{i=1}^{m} \sum_{j=1}^{n} (h_j(\hat{x}_i, \theta) - \hat{x}_{i,j})^2 \right) \tag{26}$$

Rewriting the outer sum using properties of exponentials:

$$\text{argmin}_\theta \log \prod_{i=1}^{m} \exp\left(\sum_{j=1}^{n} (h_j(\hat{x}_i, \theta) - \hat{x}_{i,j})^2\right) \tag{27}$$

Rewriting the inner sum in matrix form:

$$\text{argmin}_\theta \log \prod_{i=1}^{m} \exp\left((h(\hat{x}_i, \theta) - \hat{x}_{i,j})^T I (h(\hat{x}_i, \theta) - \hat{x}_{i,j})\right) \tag{28}$$

Negating the argument of exp and swapping the argmin for an argmax, multiplying the exponential argument by $1/2$, and multiplying the exponential by $1/\sqrt{2\pi}$:

$$\text{argmax}_\theta \log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(h(\hat{x}_i, \theta) - \hat{x}_{i,j})^T I (h(\hat{x}_i, \theta) - \hat{x}_{i,j})\right) \tag{29}$$

The term inside the product if the PDF of a multivariate Normal distribution:

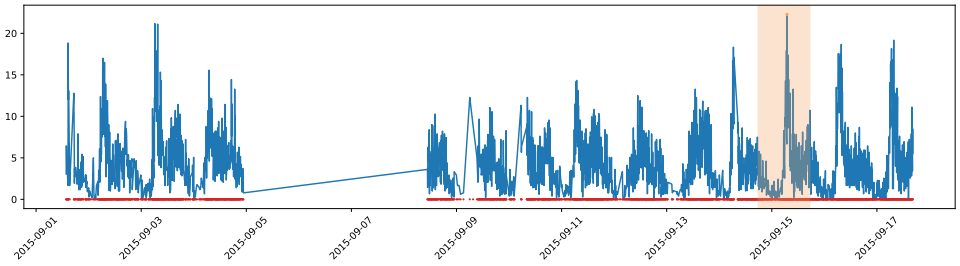
$$\text{argmin}_\theta \log \prod_{i=1}^{m} f(\hat{x}_i, h(\hat{x}_i), I) \tag{30}$$

In particular, such distribution is centered on $h(\hat{x}_i)$, and has independent Normal components all having unit variance. Lastly, when the MSE loss is used for training a neural network, such network is trained for maximum likelihood (just like density estimators).

3 Filling Missing Values in Time Series

3.1 Missing Data in Time Series

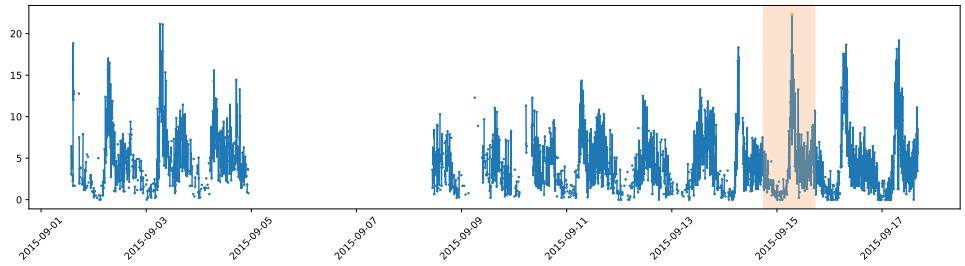
Missing values in real-world time-series are very common. These arise for a variety of reasons: malfunctioning sensors, network problems, lost data, sensor maintenance/installation/removal, and others. An example of such problem is the following:



Before dealing with missing values, the time-series' sparse indexes need to be turned into dense temporal indexes, so to properly detect where, in the signal, missing values are. Once a dense index is computed, missing values can be represented as NaN (i.e. not a number), and can be filled by replacing NaN with a meaningful value. In order to compute such index, the distance between consecutive index values in computed. One could end-up with values that are **out of alignment** (i.e. with values that are not multiples of some specific quantity). For example, using the `value_counts()` function of `pandas` one could produce the following list:

```
0 days 00:05:00 1754
0 days 00:10:00 340
0 days 00:15:00 106
0 days 00:20:00 37
0 days 00:04:00 26
0 days 00:25:00 22
0 days 00:06:00 18
0 days 00:30:00 9
0 days 00:35:00 8
0 days 00:11:00 7
Name: timestamp, dtype: int64
```

where `00:04:00`, `00:06:00` and `00:11:00` are not multiples of `00:05:00`. Thus, there is the need to realign the original index. This procedure is also called **resampling** (or **binning**), and can be done in `pandas` with the `resample(rule = None, ...)` function, where `rule` specifies the length of each individual interval (or bin). In this case, `rule = "5min"`. Now, one can inspect the new dense time-series:

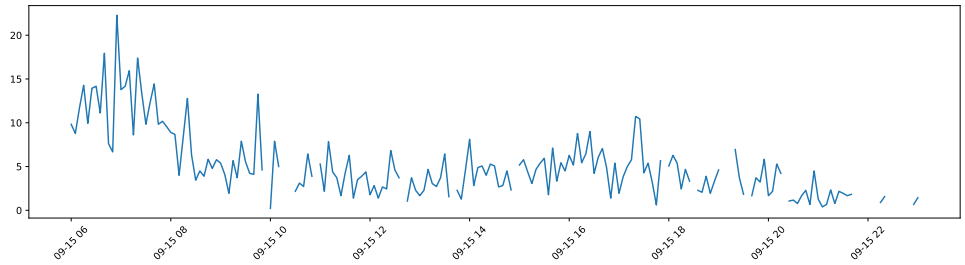


3.2 Basic Approaches for Missing Values

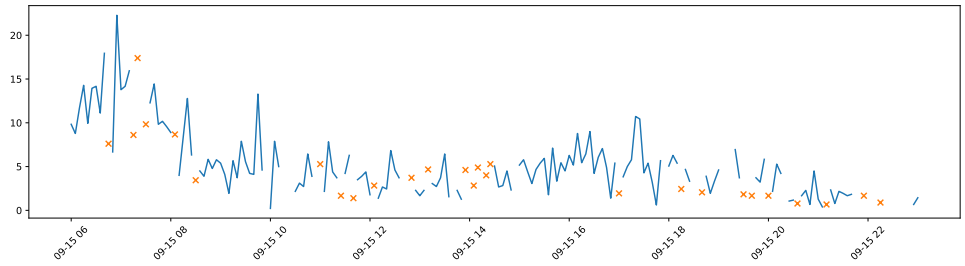
Considering specific, and mostly intact, sections of a given time-series, a way to measure the accuracy of a filling approach is to artificially remove values from such portions of the series, and to compute the root MSE

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(x_i - \hat{x}_i)^2} \tag{31}$$

where x_i is a value from the filled series, and $\hat{x}_i$ is the ground truth. In general, $x_i = \hat{x}_i$ if no value is missing, hence any MSE difference is entirely due to missing values. The portion of series which is here considered is the following:



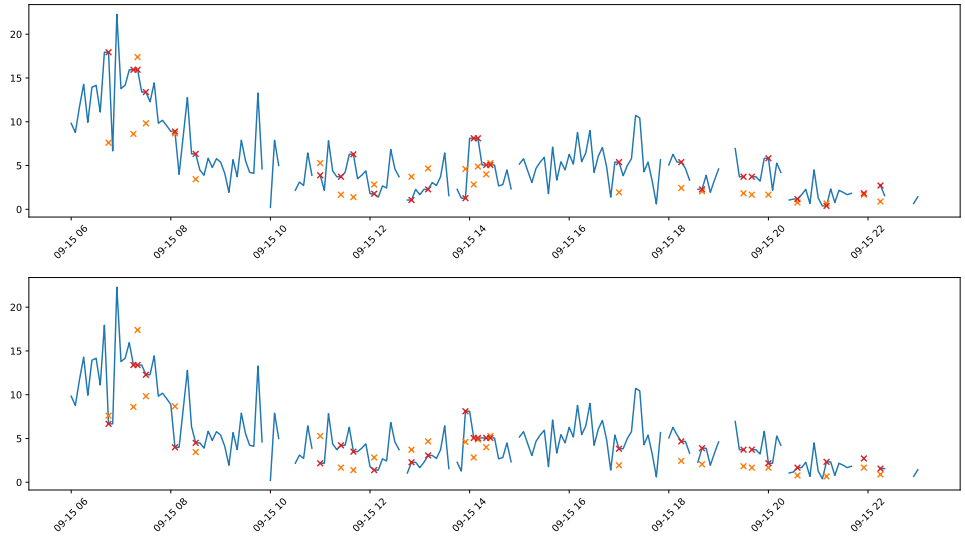
Then, some missing values are artificially introduced:



Now, some of the most useful filling techniques are listed below:

- **Forward/Backward filling**. The easiest approach for missing values consists in replicating nearby observations. In particular, forward filling propagates forward the last valid observation, while backward filling propagates backward the next valid observation. This approach works quite well, do to the presence of local correlation in the considered series.

  For example, the results of forward filling ($RMSE = 1.33$) and backward filling ($RMSE = 0.87$) would be, respectively:



  In general, forward and backward filling tend to work well for low variance sections.

- **Geometric interpolation**. This technique works by filling missing values using linear, time, nearest, polynomial, spline, etc. interpolation. In general:

  – Linear filling works well for series with slower dynamics.

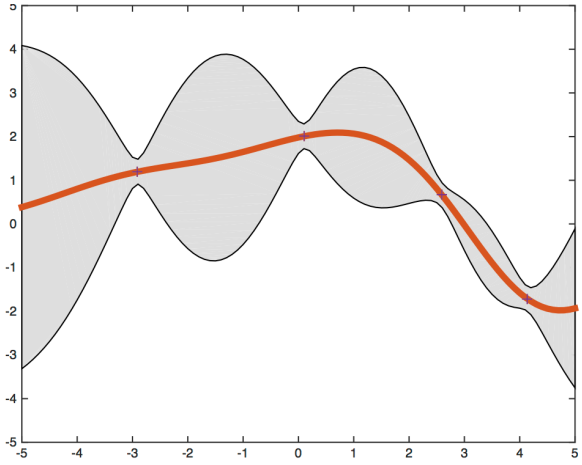
  – Nearest filling is a compromise between forward and backward filling.

  – Polynomial filling relies in nearby value to fit a polynomial. High-order polynomials often vary too much and work less well.

  – Spline filling relies on piecewise polynomial curves and it is often more robust than polynomial interpolation.

3.3 Gaussian Processes

In general, given a gap (i.e. one or more contiguous missing values), the model should be able to make a prediction about missing values, and should take into account the values that are before the hole as well as the values that comes after the hole (i.e. it should be able to interpolate all the available data). An example of a machine learning model that can do what has been stated above are is given by **Gaussian processes** (**GP**).



Gaussian processes define a probability distribution over an index (i.e. input) variable, where this distribution is based on the available observation and a few assumptions. These assumptions are:

- For every value of the index variable the distribution is Gaussian. Therefore, it can be described by a mean and standard deviation.
- The standard deviation depends on the distance between a point and the observations. Thus, it will be low when one is close to the observations, and high when one is far away from the observations.

Formally, a GP is a stochastic process (i.e. a collection of indexed random variables) where:

- The index variable x represents an input.
- Each variable y_x represents the output for input x. This output can though of as the value of a stochastic function for input x.
- The index is continuous and the collection is therefore infinite.

In particular, each y_x follows a normal distribution, but the variables are correlated. Therefore, every finite subset of y_x variables follows a multivariate normal distribution. In general, multivariate normal distributions work for many real world phenomena, and have a relatively simple closed form density function. The PDF for a multivariate normal distribution is defined via: a vector mean, μ, a covariance matrix, Σ. However, by recentering, one can assume $\mu = 0$, meaning that knowing Σ is enough. Thus, if Σ is known, one can easily compute: the joint density, $f(\hat{y}_{\hat{x}})$, for a set of observations, and the conditional density $f(y_x|\hat{y}_{\hat{x}})$ of an observation y_x given $\hat{y}_{\hat{x}}$. In particular, considering figure above, the line and grey areas represent the conditional density $f(y_x|\hat{y}_{\hat{x}})$ of y_x, based on the available observations, i.e. $\hat{y}_{\hat{x}}$.
In practice, one does not know Σ and can assume that Σ is a parameterized function $\Sigma(\theta)$ and can optimize the parameters θ for maximum likelihood. Formally, given a set of training observations $\hat{y}_{\hat{x}}$, the parameters the can be calibrated by solving a problem of the form:

$$\text{argmax}_\theta \, f(\hat{y}_{\hat{x}}) \tag{32}$$

where $f(\hat{y}_{\hat{x}})$ is the joint probability density function. Now, supposing to have a covariance matrix Σ for a set of observations $\hat{y}_{\hat{x}}$ and wanting to perform inference for an input value x (i.e. to compute $f(y_x|\hat{y}_{\hat{x}})$), then the following formula could be applied:

$$f(y_x|\hat{y}_{\hat{x}}) = \frac{f(y_x, \hat{y}_{\hat{x}})}{f(\hat{y}_{\hat{x}})} \tag{33}$$

$f(\hat{y}_{\hat{x}})$ can be easily computed by using Σ. Indeed, Σ refers to the set of observed variables $\hat{y}_{\hat{x}}$. If such set is composed of n variables, then Σ will be $n \times n$:

$$\Sigma_{\hat{x}} = \begin{bmatrix} \sigma_{\hat{x}_1,\hat{x}_1} & \sigma_{\hat{x}_1,\hat{x}_2} & \cdots & \sigma_{\hat{x}_1,\hat{x}_n} \\ \sigma_{\hat{x}_2,\hat{x}_1} & \sigma_{\hat{x}_2,\hat{x}_2} & \cdots & \sigma_{\hat{x}_2,\hat{x}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{\hat{x}_n,\hat{x}_1} & \sigma_{\hat{x}_n,\hat{x}_2} & \cdots & \sigma_{\hat{x}_n,\hat{x}_n} \end{bmatrix} \tag{34}$$

However, $f(y_x, \hat{y}_{\hat{x}})$ refers to one more variable, meaning that it will be specified via an $(n+1) \times (n+1)$ matrix:

$$\Sigma_{x,\hat{x}} = \begin{bmatrix} \sigma_{x,x} & \sigma_{x,\hat{x}_1} & \sigma_{x,\hat{x}_2} & \cdots & \sigma_{x,\hat{x}_n} \\ \sigma_{\hat{x}_1,x} & \sigma_{\hat{x}_1,\hat{x}_1} & \sigma_{\hat{x}_1,\hat{x}_2} & \cdots & \sigma_{\hat{x}_1,\hat{x}_n} \\ \sigma_{\hat{x}_2,x} & \sigma_{\hat{x}_2,\hat{x}_1} & \sigma_{\hat{x}_2,\hat{x}_2} & \cdots & \sigma_{\hat{x}_2,\hat{x}_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sigma_{\hat{x}_n,x} & \sigma_{\hat{x}_n,\hat{x}_1} & \sigma_{\hat{x}_n,\hat{x}_2} & \cdots & \sigma_{\hat{x}_n,\hat{x}_n} \end{bmatrix} \tag{35}$$

Assuming that $\hat{y}_{\hat{x}}$ are the training observations, $\sigma_{\hat{x}_i,\hat{x}_j}$ could be defined at training time. However, in order to define the new covariances (i.e. those related to y_x), one has to introduce a specific parameterized function. Given two variables y_{x_i} and y_{x_j}, one can specify their covariance via a parameterized kernel function $K_\theta(x_i, x_j)$, where K typically depends on the distance between input values. Given any finite set of variables $\{y_{x_1}, \ldots, y_{x_n}\}$, the covariance matrix is:

$$\Sigma = \begin{bmatrix} K_\theta(x_1, x_1) & K_\theta(x_1, x_2) & \cdots & K_\theta(x_1, x_n) \\ K_\theta(x_2, x_1) & K_\theta(x_2, x_2) & \cdots & K_\theta(x_2, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ K_\theta(x_n, x_1) & K_\theta(x_n, x_2) & \cdots & K_\theta(x_n, x_n) \end{bmatrix} \tag{36}$$

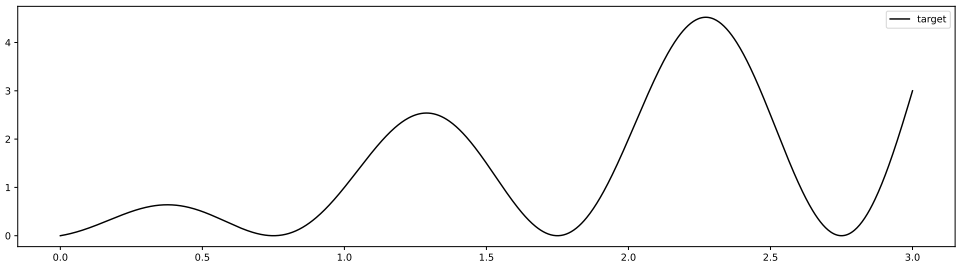
which can be computed based on the input (and the parameters) alone.
In practice, at training time:

1. A parameterized kernel function $K_\theta(x_i, x_j)$ is picked.
2. Training observations, $\hat{y}_X$, are collected.
3. The kernel is optimized for maximum likelihood (e.g.via gradient descent).

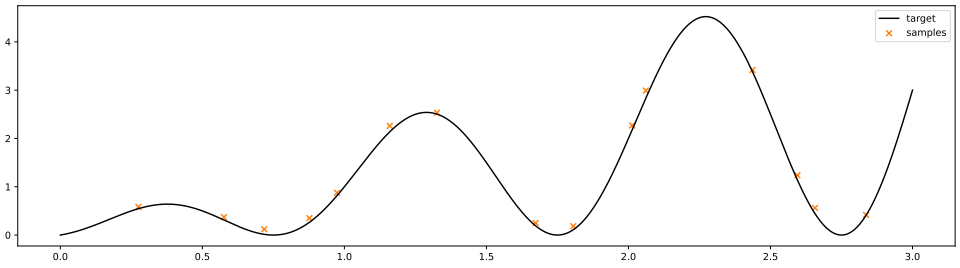
Both the parameters, θ, and the observations, $\hat{y}_{\hat{x}}$ are stored in the model. At inference time, given a new input (i.e. index) x:

1. The covariance matrix $\Sigma_{\hat{x}}$ is obtained.
2. The covariance matrix $\Sigma_{x,\hat{x}}$ is obtained.

Hence, one can completely characterize $f(y_x|\hat{y}_{\hat{x}})$. For example: let $f(x) = x\sin(2\pi x) + x$ generate the ground truth data:



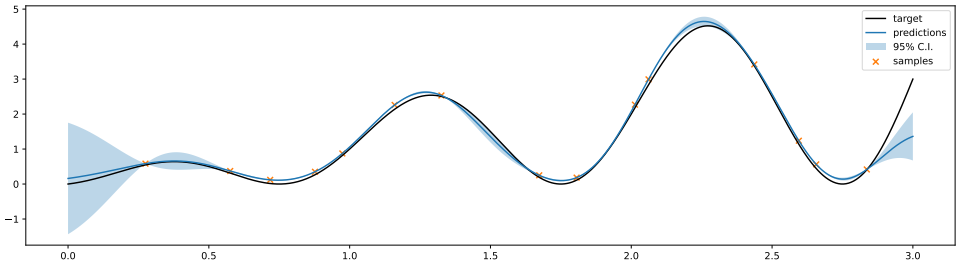
Starting from this series, a small training set of fifteen points is sampled:



Now, a specific kernel need to be chosen. In this case, a radial basis function (`RBF`) (i.e. Gaussian) kernel is picked:

$$K(x_i, x_j) = e^{-\frac{d(x_i,x_j)^2}{2l}} \tag{37}$$

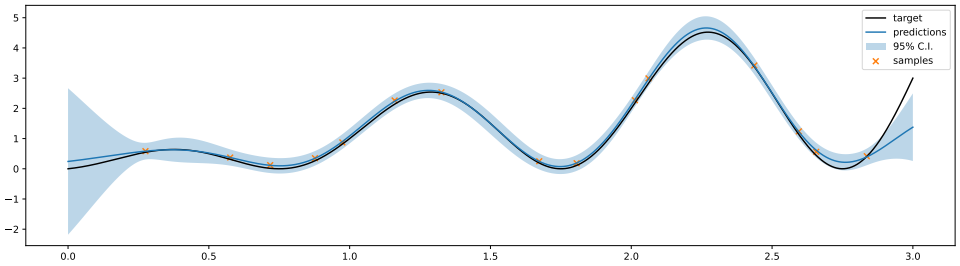
In this specific case, since the training set is composed of fifteen points, the covariance matrix will be a 15×15 matrix. In particular, the such covariance decreases with the Euclidean distance $d(x_i, x_j)$. Intuitively, the closer the points, the higher the correlation. The l parameter (namely, scale) controls the rate of the reduction. Having defined the the parameter l and the overall kernel, a Gaussian process can be trained. Training uses gradient descent to maximize the likelihood of the training data. Once trained, the predictions are not point estimates, but parameters (i.e. mean and standard deviation) of Gaussian distributions. Indeed, the model output is a fully characterized conditional distribution. Plotting the predictions:



The model can be improved by looking at the training data, and by searching for a more appropriate kernel. Indeed, one can notice how the training data have some noise, a period, and a trend. a `WhiteKernel` captures the presence of noise in the data:

$$K(x_i, x_j) = \begin{cases} \sigma^2 & \text{if } x_i = x_j \\ 0 & \text{otherwise} \end{cases} \tag{38}$$

This kernel is added to the `RBF` kernel. The only parameter of a white kernel represents the noise level σ^2. Moreover, it is often a good idea to have magnitude parameters in the kernel, which can be added by using a `ConstantKernel` (i.e. a constant factor that is multiplied to the `RBF` kernel. This product is then added to the white kernel) that allows the optimizer to tune the magnitude of the `RBF` kernel. Having considered all this, the entire procedure is repeated and the output model is the following:



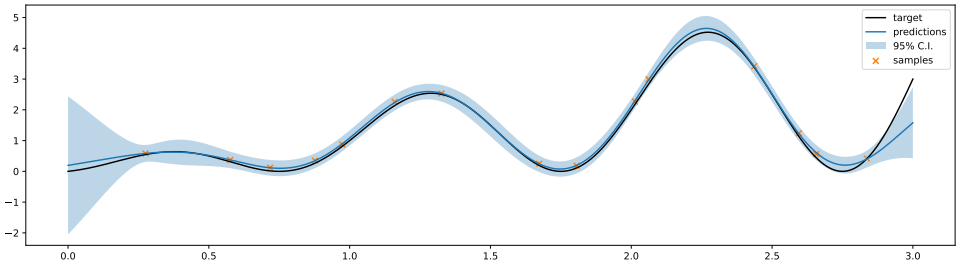
In this case, the black curve is mostly in the confidence interval, but the period and the trend have not yet been exploited. To exploit the period, an `ExpSineSquared` kernel is added to the previous kernel:

$$K(x_i, x_j) = e^{-2\frac{\sin^2\left(\pi\frac{d(x_i,x_j)}{p}\right)}{l^2}} \tag{39}$$

in particular, the correlation grows is the distance is close to a multiple of the period p, and the scale l controls the rate of decrease/increase. Moreover, to exploit the trend, a `DotProduct` kernel is added to the previous kernel:

$$K(x_i, x_j) = \sigma^2 + x_i x_j \tag{40}$$

In particular, the larger the x values, the larger the correlation, and σ controls the base level of correlation. The final output model is the following:



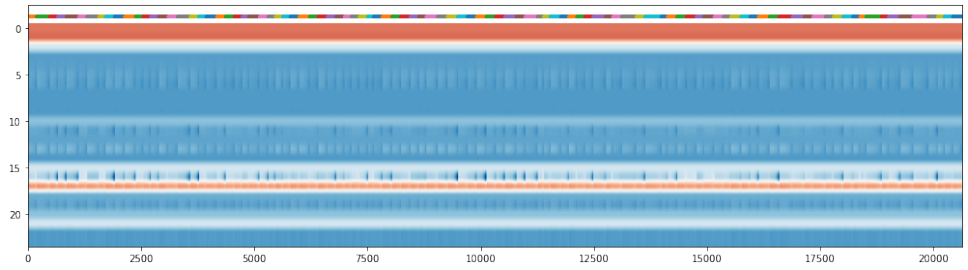
4 RUL-Based Maintenance Policies

4.1 Remaining Useful Life

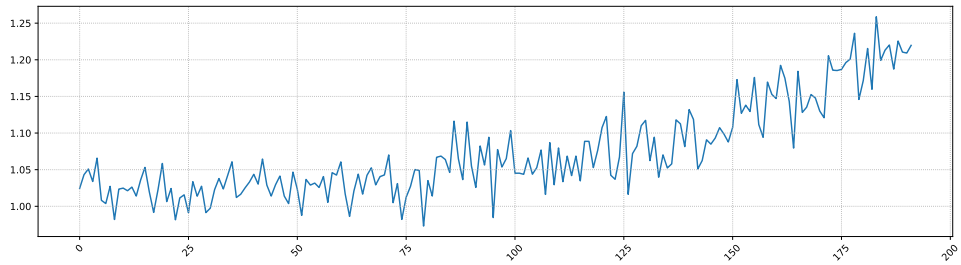
The **remaining useful life** (**RUL**) is a key concept in predictive maintenance. In particular, the RUL refers to the time until a component becomes unusable. Current best practices are based on preventive maintenance (i.e. on having a fixed maintenance schedule for each component family). RUL prediction can lead to significant savings, e.g., by delaying maintenance operations with respect to the schedule. To better study this concept, the NASA commercial modular aero-propulsion system simulation (C-MAPSS) dataset is used. MAPSS is a simulator for turbofan engines developed by NASA. The dataset consists of four training set files and four test files:

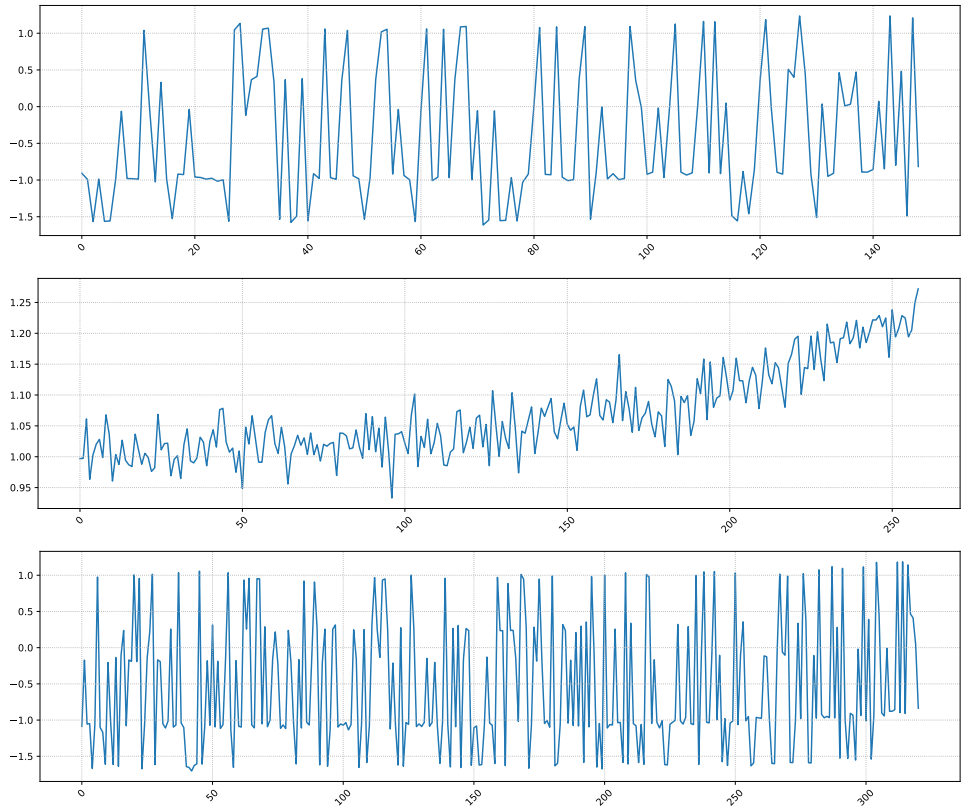
- The training set files contain multiple run-to-failure experiments.
- The test set files contain truncated experiments.

The PHM08 conference hosted a competition based on such dataset, in which the goal was to predict thr RUL at the end of each truncated experiment. This is fine as long as the focus is on pure prediction, however the whole predictive maintenance problem is to be considered here. As a consequence, the focus will be only on the training data. In particular, each training file refers to different faults and operating conditions. Each sample inside each training file contains columns related to controlled parameters, labeled as p_1, p_2, etc., and columns related to sensor reading, labeled as s_1, s_2, etc.. After having standardize each column, one can split the data based on source file and plot all parameters and sensors for each file. For example, considering the first file:



One can also inspect columns of each training file. For example, by analyzing the s_4 column of the four training files:

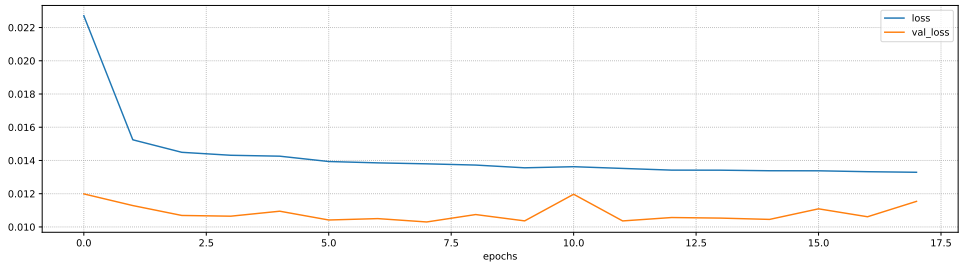




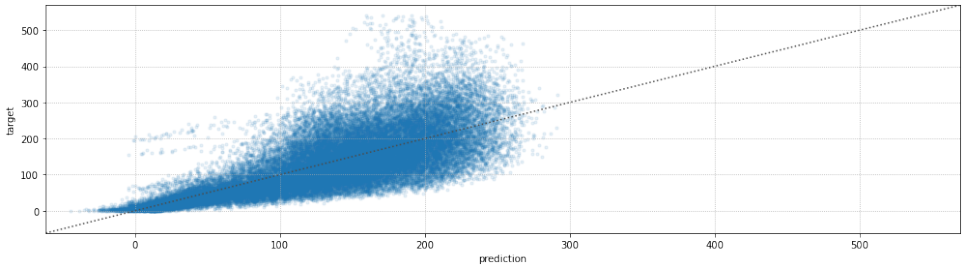
where the first and third files shows a clear trend, possibly correlated to component wear.

4.2 RUL Prediction as Regression

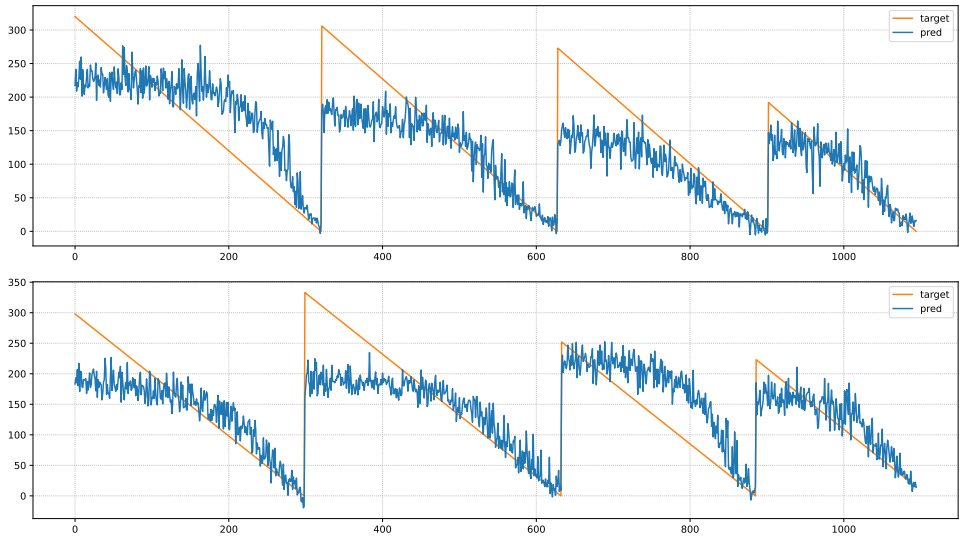
A simple formulation of a RUL-based policy is given by a regression approach. This implies triggering maintenance when the estimated RUL becomes too low, i.e. when $f(x, \lambda) < \theta$, where f is the regressor with parameter vector λ. The threshold θ must account for possible estimation errors. In the remaining of this subsection, only the fourth dataset of C-MAPSS is used.
To define the training and test datasets, a number run-to-failure experiments are run. Some of these will form the training set, and others will form the test set. Each run-to-failure experiment is associated to a machine. In order to define the two aforementioned sets, a specific percentage of machines can be selected so to populate the training set (e.g. 75%). Given such sets, all parameters and sensor inputs are standardized, and all RUL values (i.e. the regression targets) are normalized.
In this context, a multi-layer perceptron (MLP) is used to solve the given regression problem. The computed results are the following:



The quality of the predictions are then evaluated:



Lastly, the RUL on the training set and on the test set are computed:



It can be noticed how the accuracy is quite poor, especially for large RUL values. However, the goal of this approach is not to reach a high accuracy, but to stop at the right time. For a proper evaluation, a cost model is then needed. To define such model, few assumptions are made:

- One step of operation is considered as a value unit, so that the failure cost can be expressed in terms of operating steps. Namely, one step of operation brings one unit of profit.
- Every run ends with either failure of maintenance. Assuming that the failure cost is higher than maintenance cost, the maintenance cost can be disregarded. Namely, a failure costs C units more than maintenance.
- A traditional preventive maintenance policy is available. Maintenance is not triggered earlier that such policy. Namely, only what happens after s steps is counted.

Formally, let x_k be the series for machine k, and I_k its set of steps. The time-step at which said policy triggers maintenance is given by:

$$\min\{i \in I_k | f(x_{ki}) < \theta\} \tag{41}$$

A failure occurs if:

$$f(x_{ki} \geq \theta) \quad \forall i \in I_k \tag{42}$$

The whole cost formula for a single machine will be:

$$cost(f, x_k, \theta) = op\_profit(f(x_k), \theta) + fail\_cost(f(x_k), \theta) \tag{43}$$

where:

$$op\_profit(f(x_k), \theta) = -\max\{0, \min\{i \in I_k | f(x_{ki}) < \theta\} - s\} \tag{44}$$

and:

$$fail\_cost(f(x_k), \theta) = \begin{cases} C & \text{if } f(x_{ki}) \geq \theta \ \forall i \in I_k \\ 0 & \text{otherwise} \end{cases} \tag{45}$$

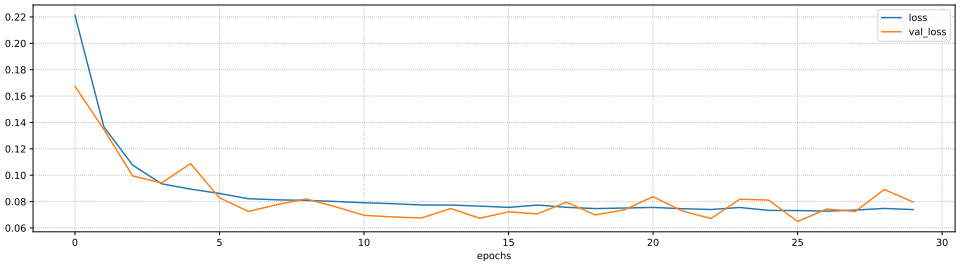
Thus, s units of machine operation are guaranteed, and profit is modeled as a negative cost. For the total cost, one can sum over all machines. Then, in practice, s is determined by the preventive maintenance schedule, and C must be determined by discussing with the customer. When considering what has been achieved using the MLP model and said cost model, one can notice how the computed results are actually quite good.

4.2.1 Sequence Input in Neural Models

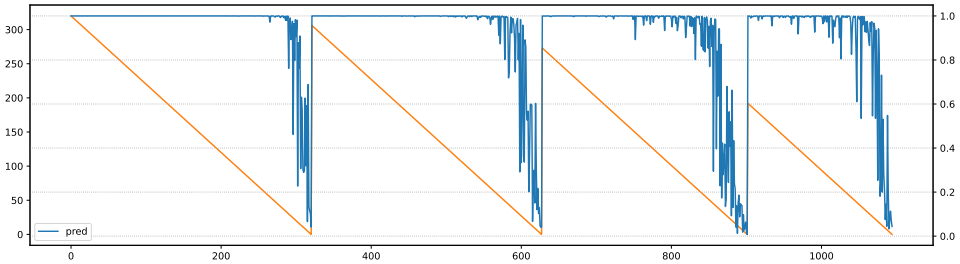
Feeding more time-steps to the neural network might improve the results. Intuitively, sequences provide information about the trend, and this may allow a better RUL estimate with respect to using only the current state. For example, one may gauge how quickly the component is deteriorating. To achieve this, a sliding window is necessary. Then, a convolutional neural network, that uses 1D convolutions, can be used as a regressor. However, not always an input sequence is useful. Indeed, sequences matter only if the input is correlated with patterns that involve multiple time-steps.

4.3 RUL Prediction as Classification

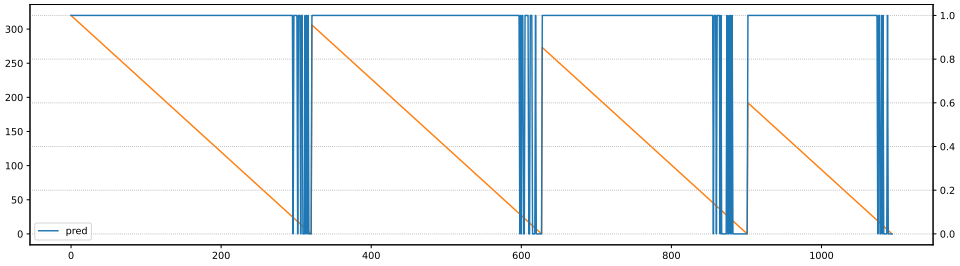
RUL-based maintenance can also be tackled using a classifier. In particular, a classifier can be built in order to determine whether a failure will occur in θ steps. As soon as the classifier outputs, e.g., a zero (i.e. $f_\theta(x, \lambda) = 0$), the inference is stopped. In a sense, this approach tries to directly learn a maintenance policy, where the policy is of the form "stop θ units before a failure". Before training such classifier, it is necessary to define the target classes (namely, to define the detection horizon θ). For example, the class "1" can be used if a failure is more than θ steps away, while the class "0" if a failure is less than θ steps away.
Considering the same problem as before, a specific MLP is used to solve the discussed classification task. The computed results are the following:



The model prediction can be interpreted as the probability of not stopping:



Such probability falls closer to failures. In practice, the predictions are converted into integers via rounding:



The classifier can be evaluated directly, because it defines the whole policy, with no need for additional calibration. For example, one could compute the cost of the model, the average number of fails, and the average slack on the training and test sets. However, it could turn out that the performances of the classifier are worse than the ones of the best regression model. This could be due to the fact that, like in the regression case, θ is used as a threshold, but in this case it is employed to define the classes. This approach has both pros and cons. Ideally, one can choose how close to the failure one should stop, but early signs of failure might not be evident in the chosen interval. Moreover, one does not calibrate θ. Indeed, in the regression case, one is formally solving:

$$\text{argmin}_\theta \sum_{k \in K} cost(f(\hat{x}_k \omega^*), \theta) \quad \text{s.t. } \omega^* = \text{argmin}_\omega \, L(f(\hat{x}_k \omega), \hat{y}_k) \tag{46}$$

where ω^* is the optimal parameter vector (i.e. the network weights), L is the loss function (i.e. the MSE), and *cost* is the cost model. The threshold θ is chosen so as to minimize such cost. This represents a bi-level optimization problem. However, since θ appears neither in L nor in f, it can be decomposed into two sequential sub-problems. In the classification case, on the other hand, one is formally solving:

$$\text{argmin}_\theta \sum_{k \in K} cost(f(\hat{x}_k \omega^*), 1/2) \quad \text{s.t. } \omega^* = \text{argmin}_\omega \, L(f(\hat{x}_k \lambda), \mathbb{1}_{y_k \geq \theta}) \tag{47}$$

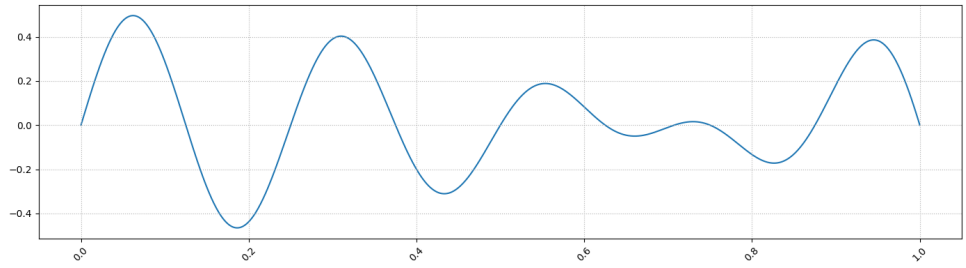
where a canonical threshold (i.e. $1/2$) is used in the cost model. L is again the loss function (i.e. binary cross entropy), and $\mathbb{1}_{y_k \geq \theta}$ is the indicator function of $y_k \geq \theta$ (i.e. class labels). Unlike the previous one, this problem cannot be decomposed, because θ appears in the loss function. This means that one needs to optimize θ and ω at the same time. This problem can be solved applying grid-search, but this approach is computationally expensive.

4.4 Bayesian (Surrogate-Based) Optimization

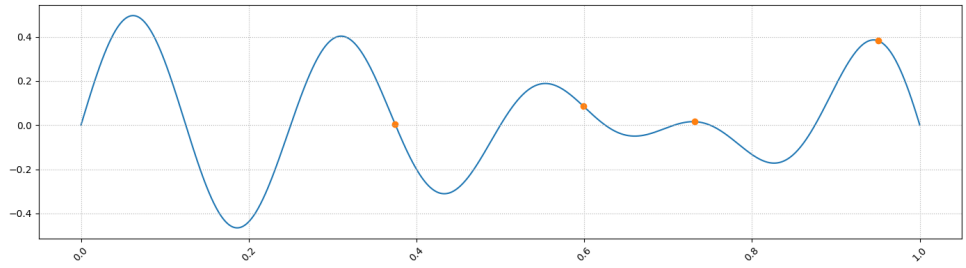
The **surrogate-based Bayesian optimization** (**SBO**) approach can be used to optimize black-box functions (i.e. functions with an unknown structure, that can only be evaluated, and their evaluation is expensive). Formally, such problems have the form

$$\text{min}_{x \in B} \, f(x) \tag{48}$$

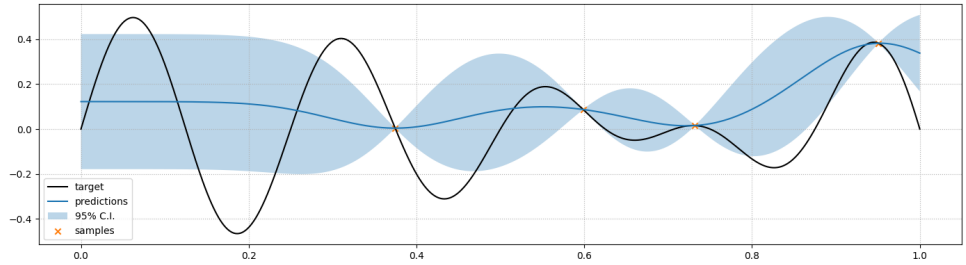
where B is a box (i.e. a specification of bounds for each component of x). The decision variables for such a problem will be x and θ, and the function to be optimized would be the cost. Since evaluating f is expensive, it should be done infrequently, and the main trick to achieve this is using a surrogate model (i.e. a machine learning model). The idea is to use a neural model instead of f. Since optimizing such model is equivalent to optimize over some prior information (i.e. the current model), and since such model is refined based on its evaluation (i.e. posterior information), this approach is called a Bayesian optimization.
For example, let's assume one is trying to minimize the following function over $[0, 1]$, where this function shows multiple local minima, and where the global minimum is $\simeq 0.19$:



Let's sample a few points at random:



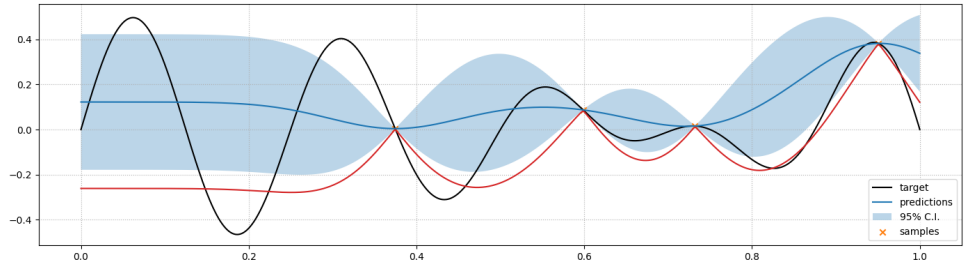
At this points, the surrogate model should approximate very accurately all evaluated points, and reflect our confidence level on unexplored regions. A Gaussian process has exactly this properties: it can interpolate very well known measurements and can provide a confidence level that decays with the distance from the observations. To deal with the case shown above, a GP that uses a kernel composed of a RBF and a white noise kernel is used:



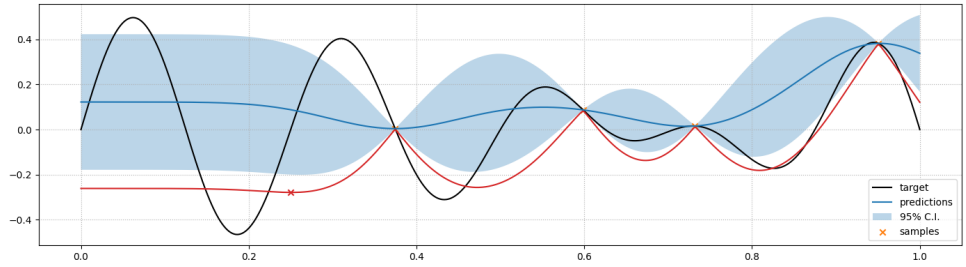
All known points are interpolated (almost) exactly and the confidence intervals behave in an intuitive fashion. Now there is the need to search over the surrogate model (i.e. which ares does it make sense to explore, and why?): this is the same as choosing which function to optimize. In particular, one needs to account for both the predictions and their confidence: areas with low predictions are promising, but so are areas with high confidence. This issue can be solved in SBO by optimizing an **acquisition function** which should balance exploration and exploitation. In this case, the Lower Confidence Bound (LCB) function is used:

$$LCB(x) = \mu(x) - Z_\alpha \sigma(x) \tag{49}$$

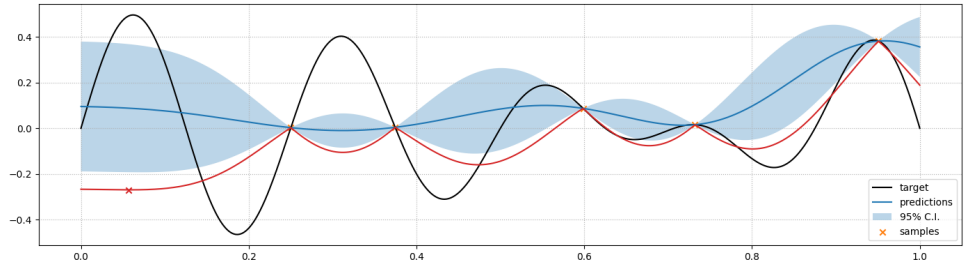
where $\mu(x)$ is the predicted mean, $\sigma(x)$ is the predicted standard deviation, and Z_α is a multiplier for a $\alpha\%$ Normal confidence interval. An example with $Z_\alpha = 2.5$ is the following:



At this points, one can optimize via any method applicable to the surrogate. In the plot above, the x value with the best acquisition function is highlighted:



Now, the surrogate model can be updated. First, f is evaluated for the new point and the training set is grown. Then, the GP can be retrained. Lastly, the acquisition function if optimized again:



The general idea behind SBO is the following: given a collection $\{\hat{x}_i, \hat{y}_i\}_i$ of evaluated points, train a surrogate model $\tilde{f}$ for f and proceed as follows:

1. Optimize an acquisition function $a_{\tilde{f}}(x)$ to find a value x'.

2. Evaluate $y' = f(x')$.

3. If y' is better than the current optimum $f(x*)$, then replace $x*$ with x'.

4. Expand the collection of measurements to include (x', y').

5. Re-train $\tilde{f}$.

6. Repeat until a termination condition is reached.

4.4.1 SBO for Threshold Calibration

SBO can be used to tackle the policy definition problem:

$$\text{argmin}_\theta \sum_{k \in K} cost(f(\hat{x}_k \omega^*), 1/2) \quad \text{s.t. } \omega^* = \text{argmin}_\omega \, L(f(\hat{x}_k \lambda), \mathbb{1}_{y_k \geq \theta})$$

Here there is the need to optimize over θ, and the goal is to minimize the cost. Computing the cost requires to re-define the classes, and therefore to repeat training. SBO can be used by exploiting the existing `scikit-optimize`. One one needs to do is to define the black-box function and run the optimization process.

5 Probabilistic Models