Artificial Intelligence in Industry

(Please refer to the `.ipynb` files for more details about experiments)

Matteo Donati

March 3, 2023

Contents

1 Anomaly Detection via Simple Methods

1.1 Problem and Data – first thing to do is to inspect the 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

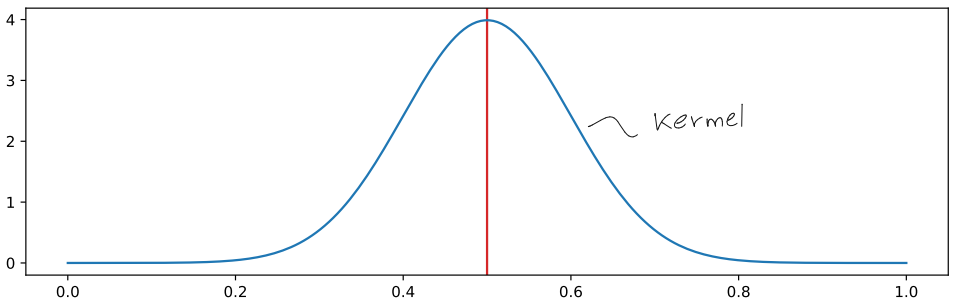
this is usually a drawback

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:

simplest approach

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

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:

probability of x_i wrt each kernel

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

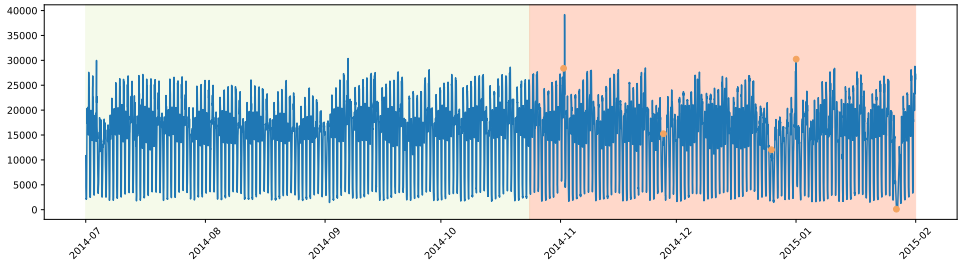
where x is the input for which to compute the estimate, $\hat{\boldsymbol{x}}$ is the matrix containing the training samples, $x - \hat{x}_i$ is the difference between x and the i-th training sample. Thus, KDE models are not trained in the usual sense: the training set is part of the model parameters.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

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

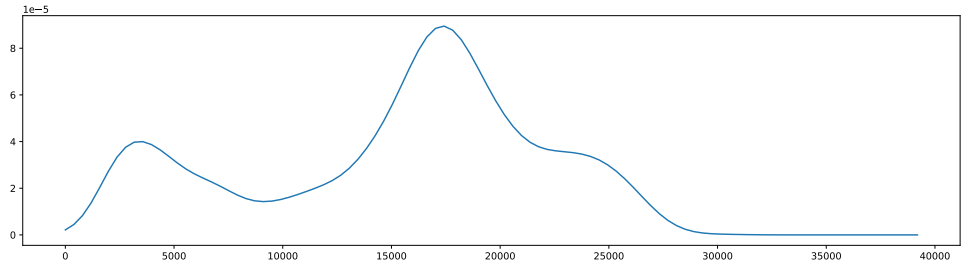
where $\hat{\sigma}$ is the standard deviation computed using the training data, and IQR is the inter-quartile range. Lastly, to avoid taking products, one can work with negated log probabilities, so that the anomaly detection condition becomes $-\log f(x, \omega) \geq \theta$.

1.3 KDE for Anomaly Detection

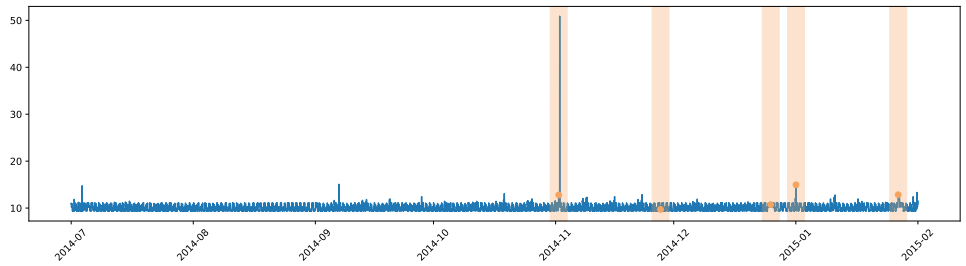
Considering a time-series, one can split it into training set and test set, where the training set only includes data about normal behavior and it will be used to fit a KDE model, while the test is used to asses how well the approach can generalize. If the training set contains some anomalies, then it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



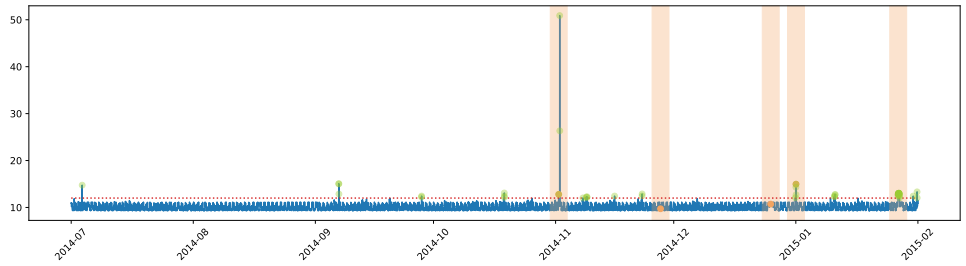
At this point, a univariate KDE is fit to the training data, obtaining the following estimated distribution:



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.

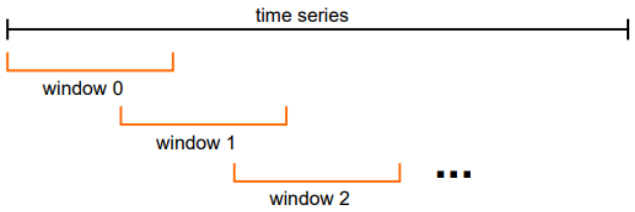
this should contain some anomalies.

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

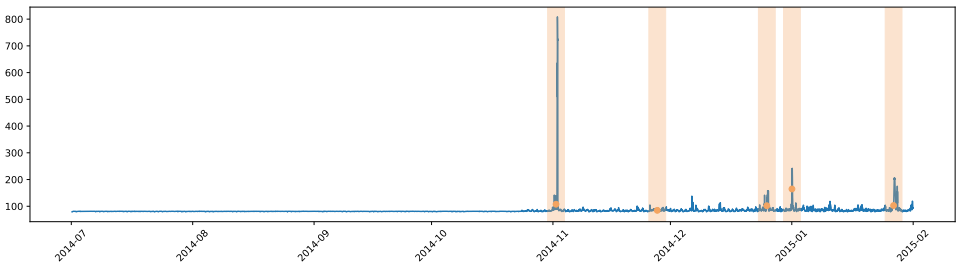
When dealing with KDE, there exists a straightforward approach to take sequences input into account. In particular, this can be done by using multivariate KDE: each sequence is treated as a vector variable, and individual sequences are treated as independent (Markov property). Then, a multivariate KDE 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:

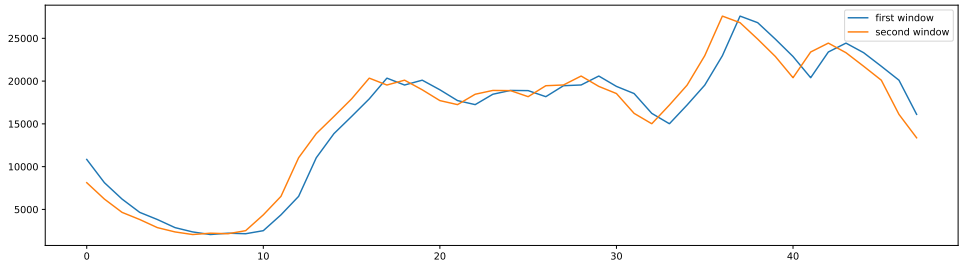


This signal shows much less noise with respect to the one computed when considering single observations.

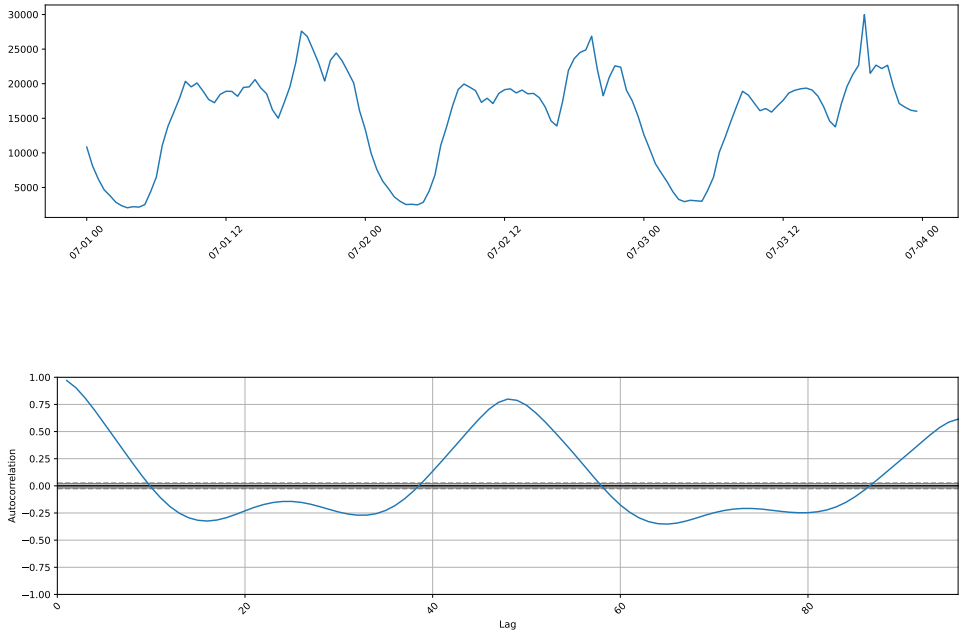
2 Anomaly Detection via Advanced Methods

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:

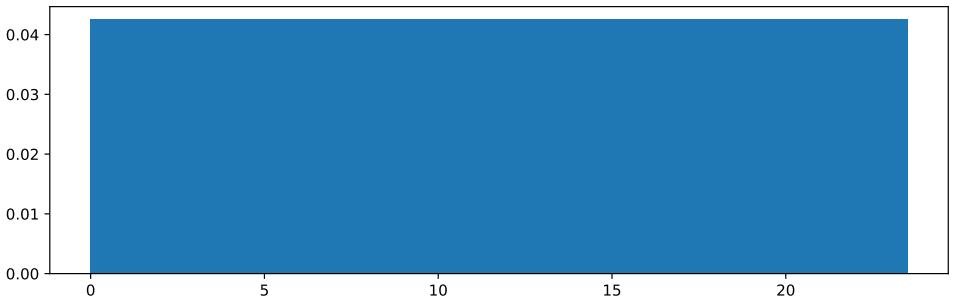


Thus, the previous sequence-based estimator is solving a uselessly complicated problem, and it is not using all the available knowledge. In order to solve these two problems, one could start by adding the

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:

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

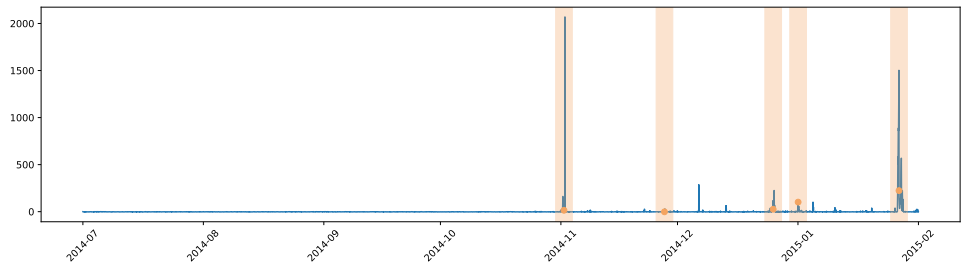
In order implement this, one needs an estimator for $f(t, x)$ (e.g. using KDE), and one estimator for $f(t)$ which one can easily obtain (e.g. using KDE again).



From equation 7:

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

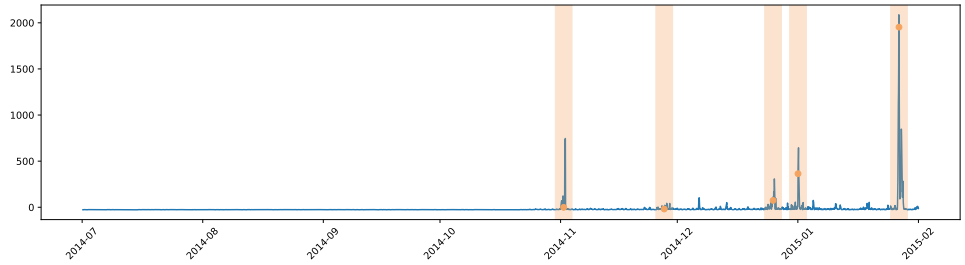
Now, having chosen a specific bandwidth through grid-search, one can produce the specific alarm signal:



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:

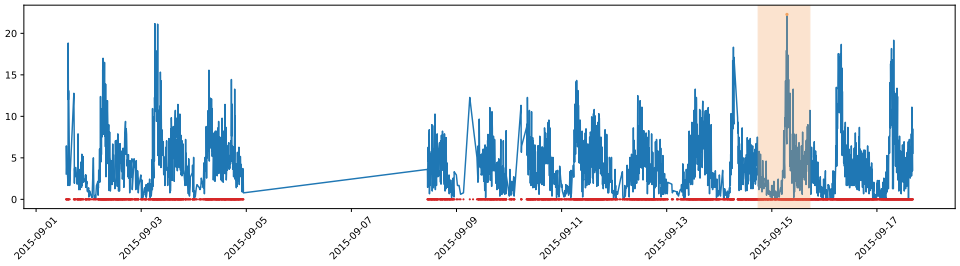
$$\text{argmin}_\theta \log \prod_{i=1}^{m} f(\hat{x}_i, h(\hat{x}_i), I) \tag{30}$$

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

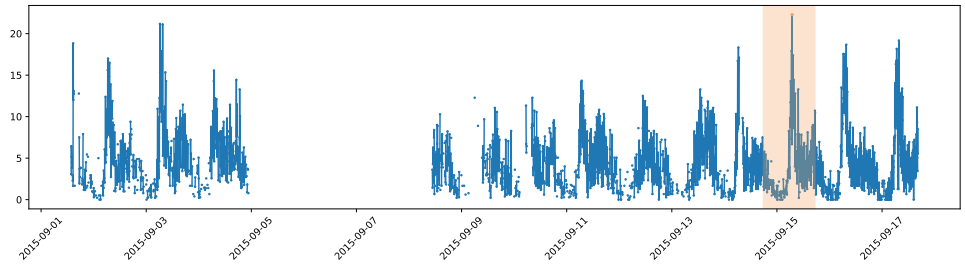
Missing values in real-world time-series are very common. These arise for a variety of reasons: malfunctioning sensors, network problems, lost data, sensor maintenance/installation/removal, and others. An example of such problem is the following:



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:

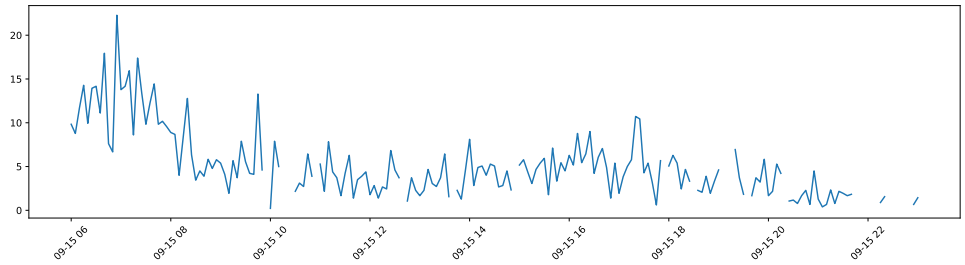


3.2 Basic Approaches for Missing Values

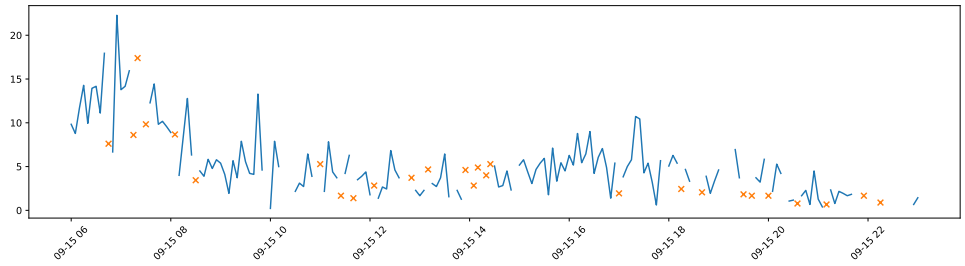
Considering specific, and mostly intact, sections of a given time-series, a way to measure the accuracy of a filling approach is to artificially remove values from such portions of the series, and to compute the root MSE

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

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:



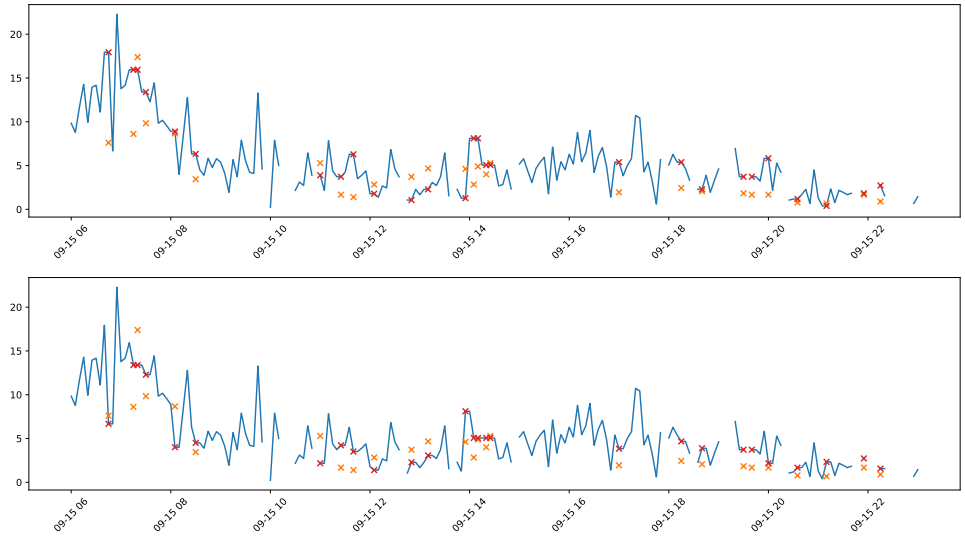
Then, some missing values are artificially introduced:



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.

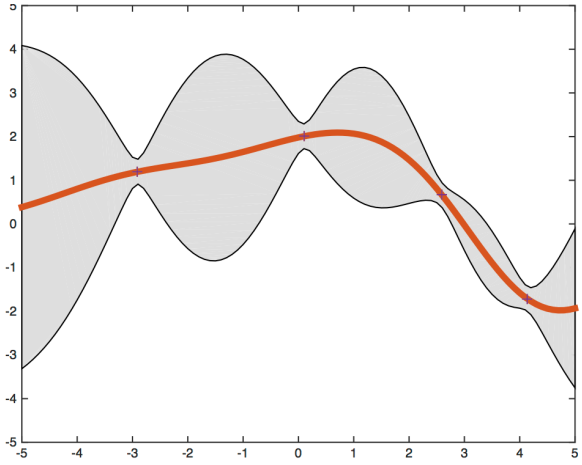
  – Nearest filling is a compromise between forward and backward filling.

  – 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:

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

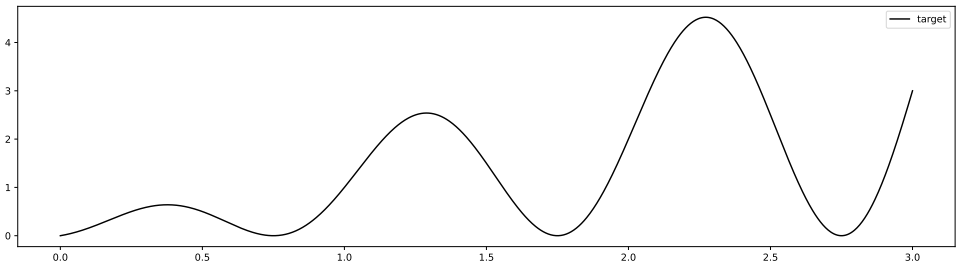
which can be computed based on the input (and the parameters) alone.
In practice, at training time:

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).

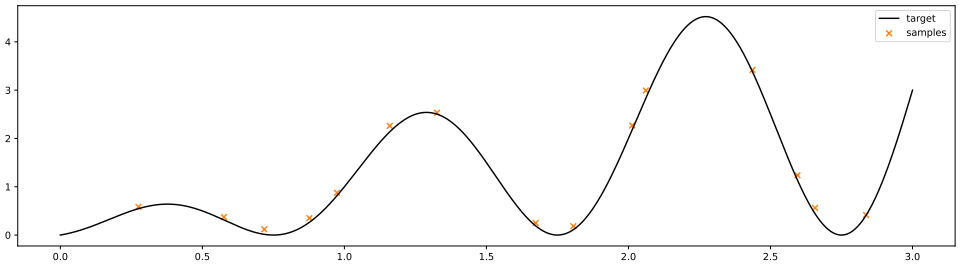
Both the parameters, θ, and the observations, $\hat{y}_{\hat{x}}$ are stored in the model. At inference time, given a new input (i.e. index) x:

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:



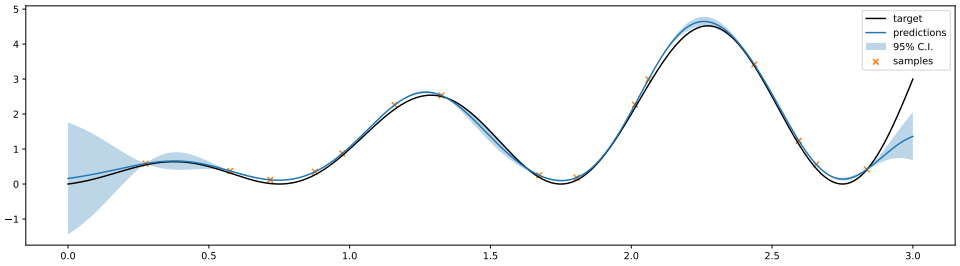
Starting from this series, a small training set of fifteen points is sampled:



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

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

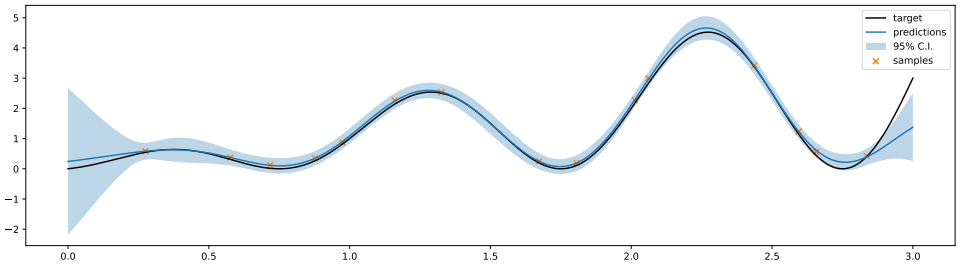
In this specific case, since the training set is composed of fifteen points, the covariance matrix will be a 15×15 matrix. In particular, the such covariance decreases with the Euclidean distance $d(x_i, x_j)$. Intuitively, the closer the points, the higher the correlation. The l parameter (namely, scale) 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:



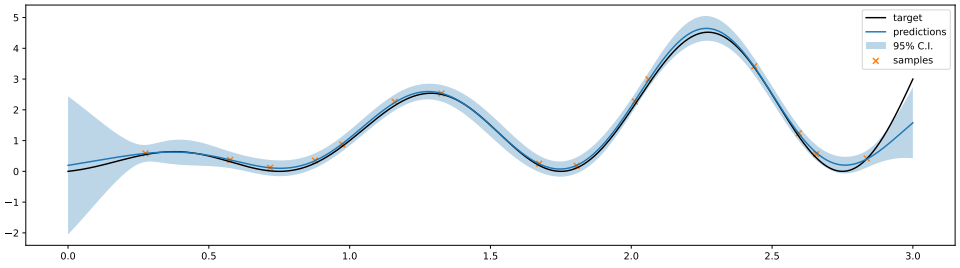
In this case, the black curve is mostly in the confidence interval, but the period and the trend have not yet been exploited. To exploit the period, an `ExpSineSquared` kernel is added to the previous kernel:

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

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:



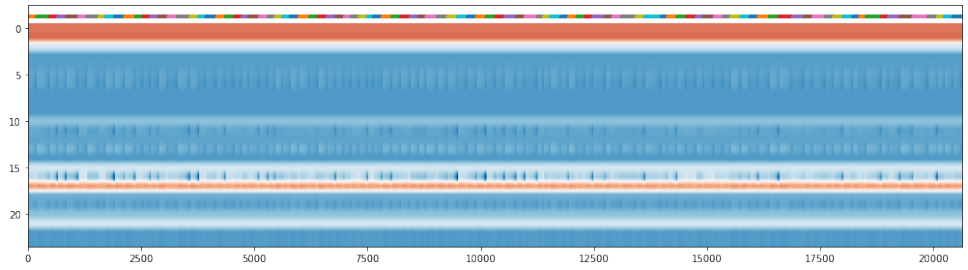
4 RUL-Based Maintenance Policies

4.1 Remaining Useful Life

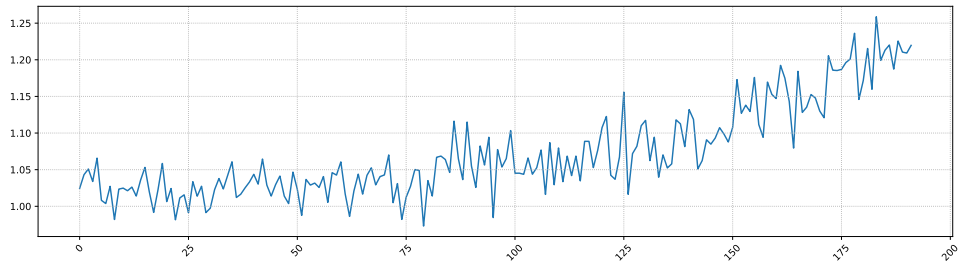
The **remaining useful life** (**RUL**) is a key concept in predictive maintenance. In particular, the RUL refers to the time until a component becomes unusable. Current best practices are based on preventive maintenance (i.e. on having a fixed maintenance schedule for each component family). RUL prediction 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:

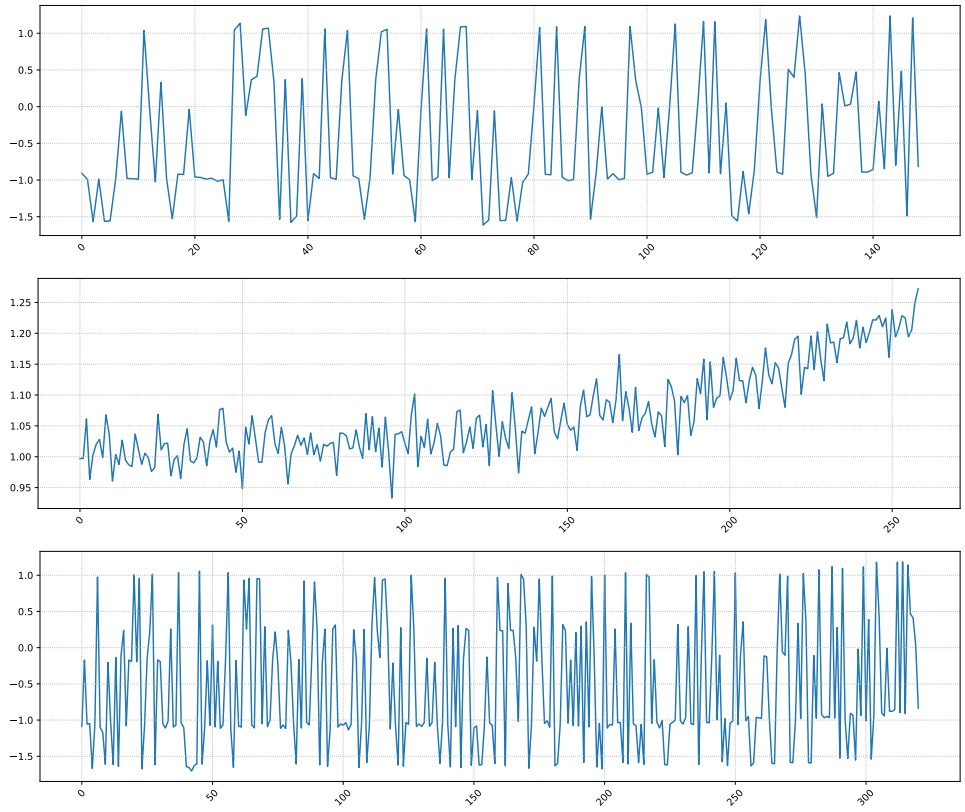
- The training set files contain multiple run-to-failure experiments.
- The test set files contain truncated experiments.

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:

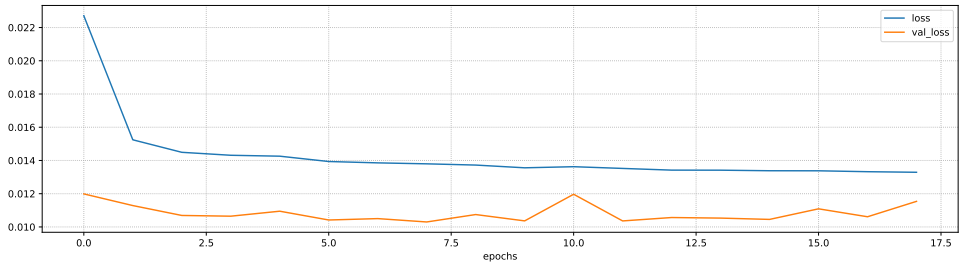




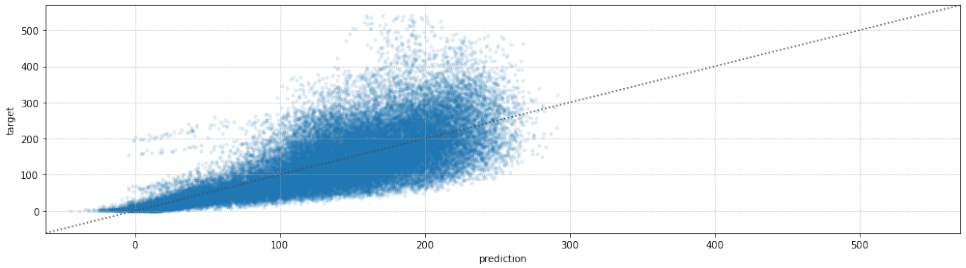
where the first and third files shows a clear trend, possibly correlated to component wear.

4.2 RUL Prediction as Regression

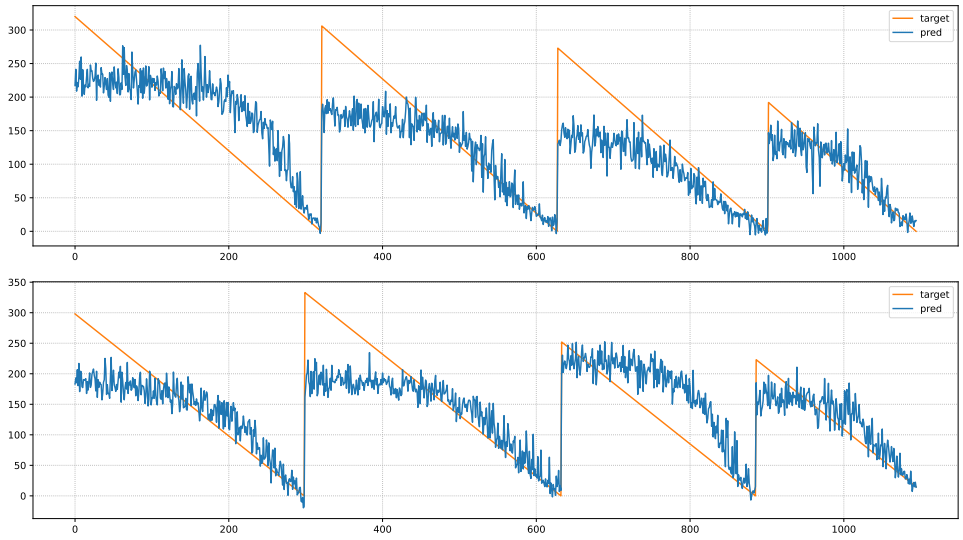
A simple formulation of a RUL-based policy is given by a regression approach. This implies triggering maintenance when the estimated RUL becomes too low, i.e. when $f(x, \lambda) < \theta$, where f is the regressor with parameter vector λ. The threshold θ must account for possible estimation errors. 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:

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

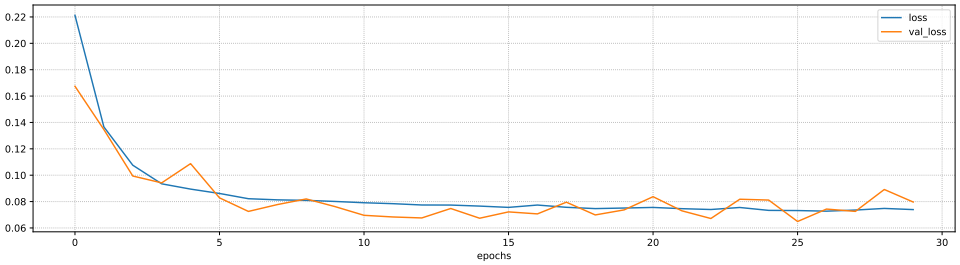
Thus, s units of machine operation are guaranteed, and profit is modeled as a negative cost. For the total cost, one can sum over all machines. Then, in practice, s is determined by the preventive maintenance schedule, and C must be determined by discussing with the customer. When considering what has 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

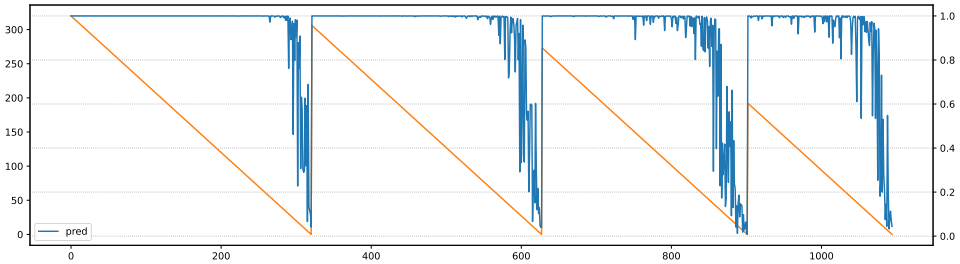
Feeding more time-steps to the neural network might improve the results. Intuitively, sequences provide information about the trend, and this may allow a better RUL estimate with respect to using only the current state. For example, one may gauge how quickly the component is deteriorating. To achieve 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

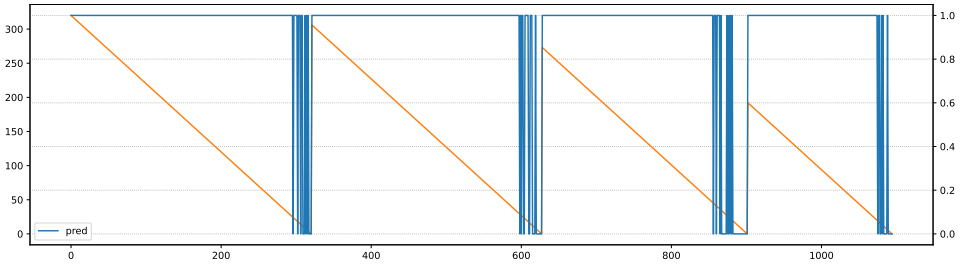
RUL-based maintenance can also be tackled using a classifier. In particular, a classifier can be built in order to determine whether a failure will occur in θ steps. As soon as the classifier outputs, e.g., a zero (i.e. $f_\theta(x, \lambda) = 0$), the inference is stopped. In a sense, this approach 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:

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

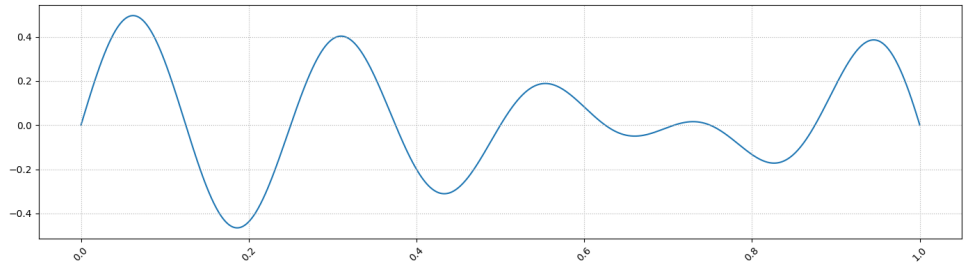
where a canonical threshold (i.e. $1/2$) is used in the cost model. L is again the loss function (i.e. binary cross entropy), and $\mathbb{1}_{y_k \geq \theta}$ is the indicator function of $y_k \geq \theta$ (i.e. class labels). Unlike the previous one, this problem cannot be decomposed, because θ appears 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

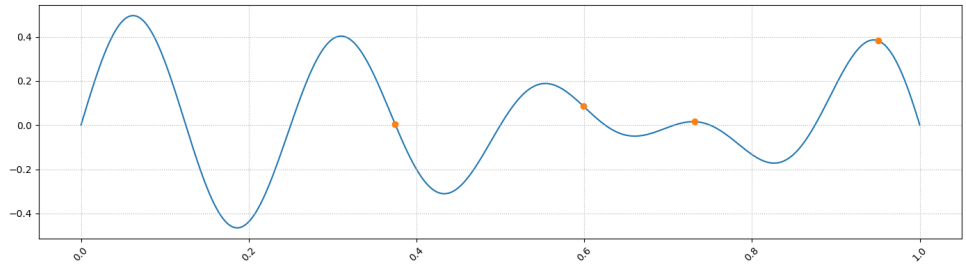
The **surrogate-based Bayesian optimization** (**SBO**) approach can be used to optimize black-box functions (i.e. functions with an unknown structure, that can only be evaluated, and their evaluation is expensive). Formally, such problems have the form

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

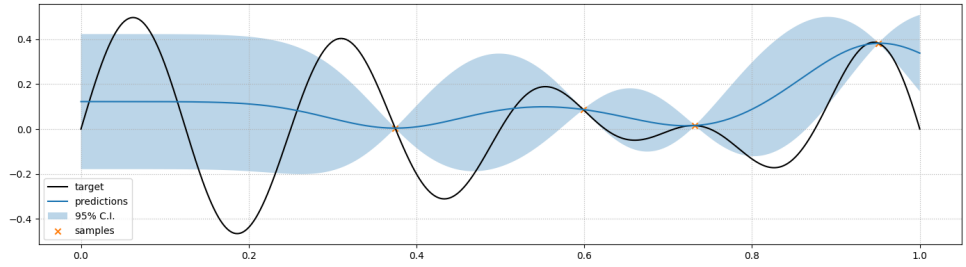
where B is a box (i.e. a specification of bounds for each component of x). The decision variables for such a problem will be x and θ, and the function to be optimized would be the cost. Since evaluating f is expensive, it should be done infrequently, and the main trick to achieve this is using a surrogate 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:



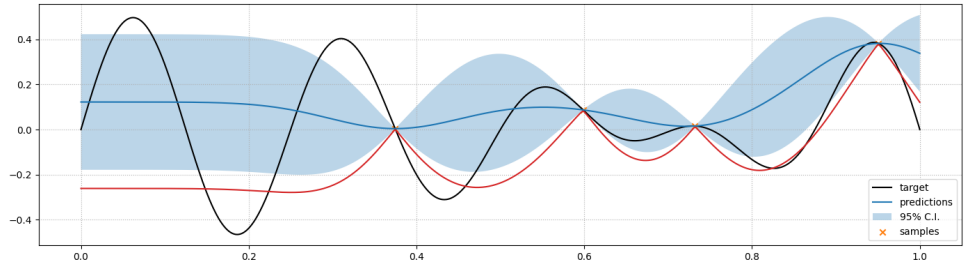
At this points, the surrogate model should approximate very accurately all evaluated points, and reflect our confidence level on unexplored regions. A Gaussian process has exactly this properties: it can interpolate very well known measurements and can provide a confidence level that decays with the 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:



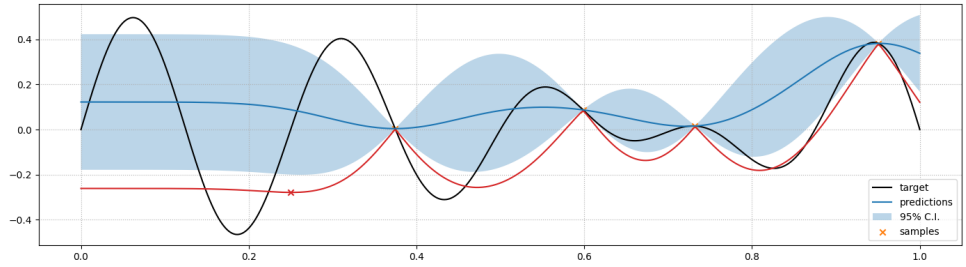
All known points are interpolated (almost) exactly and the confidence intervals behave in an intuitive fashion. Now there is the need to search over the surrogate model (i.e. which ares does it make sense to explore, and why?): this is the same as choosing which function to optimize. In particular, one 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:

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

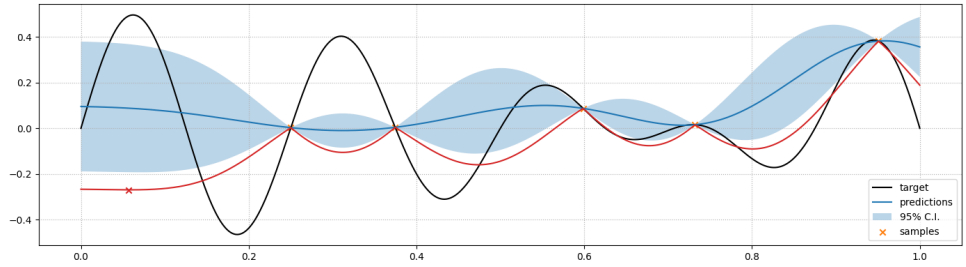
where $\mu(x)$ is the predicted mean, $\sigma(x)$ is the predicted standard deviation, and Z_α is a multiplier for a $\alpha\%$ Normal confidence interval. An example with $Z_\alpha = 2.5$ is the following:



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

5.1 Component Wear and Binning

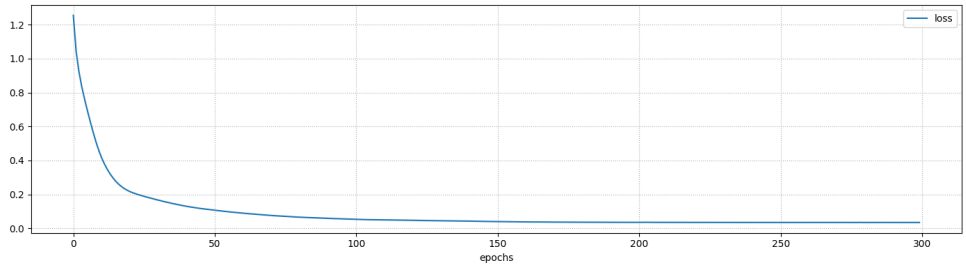
One common type of anomaly is due to component wear. Run-to-failure experiments are a typical way to investigate this type of anomalies. All problems in this class share a few properties:

- There is a critical anomaly at the end of the experiment.
- The behavior becomes more and more distant from normal over time.

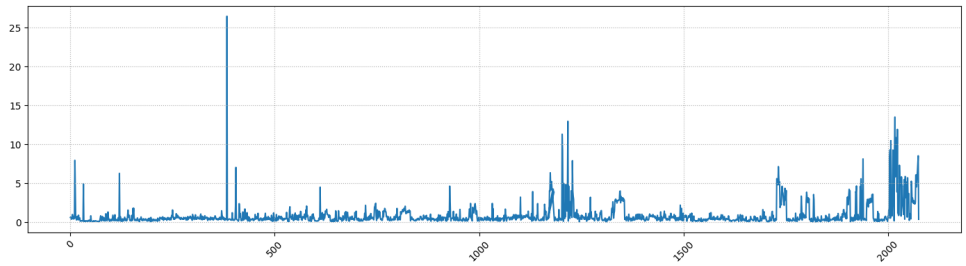
The dataset that will be considered is a dataset coming from OCME about the functioning of packaging machines. This dataset contains a single run-to-failure experiment, and refers to disjoint measurement windows. Each segment, in particular, contains data sampled every 4ms. Being this a high-frequency dataset, it could be a good idea to apply subsampling. More specifically, a sliding window, whose consecutive applications do not overlap, is applied. Each window application is called a bin, and it can be used to extract features by means of different aggregation functions. The result of this subsampling operation will be a time series that contains a smaller number of samples, but typically a larger number of features. Usually, the bin size if chosen so as to have enough data to compensate noise, and so as to capture regular patters (e.g. spiking signals).

5.1.1 Baseline Approach

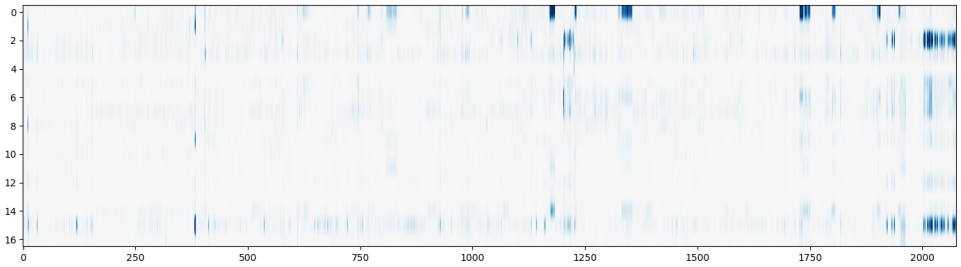
The baseline approach consists in detecting a component's state by learning an autoencoder. In particular, one can train a model on the earlier data, and then use the reconstruction error as a proxy for component wear. For example, after having preprocessed the data, one could train an autoencoder:



where the reconstruction error could look like



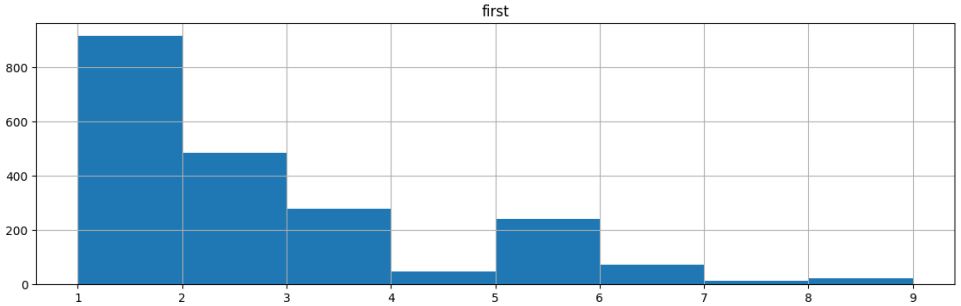
One could gain more information by checking the individual errors:



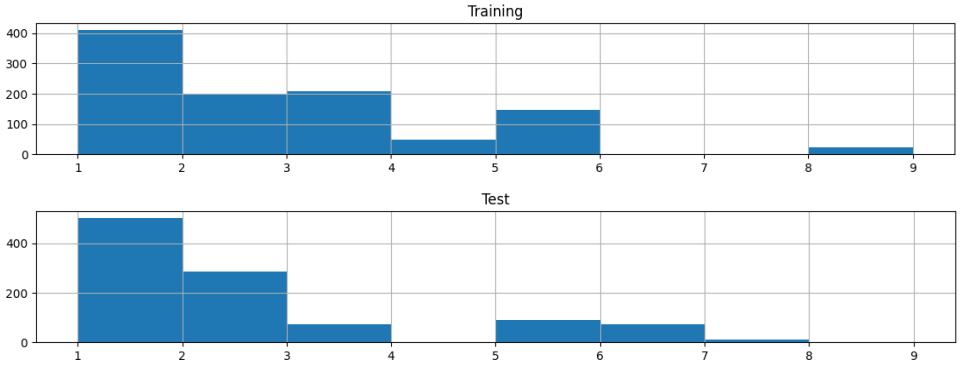
It seems that reconstruction errors are large for different features over time.

5.1.2 Altering the Training Distribution

A major problem of the given data is related to the distribution balance: the modes of operation of the machine are not used equally often:



Indeed, there is a difference between the training and test distribution:



This problem matters because one is training the model according to maximum likelihood:

$$\text{argmax}_\theta \, \mathbb{E}_{\hat{x},\hat{y}\sim P} \left[\prod_{i=1}^{m} f_\theta(\hat{y}_i|\hat{x}_i) \right] \tag{50}$$

where P represents the real (joint) distribution, $f_\theta(\cdot|\cdot)$ is the parameterized model (i.e. an estimator for a conditional distribution). In the case of an autoencoder, $\hat{x}$ and $\hat{y}$ are the same. However, in practice, one does not have access to the full distribution, so one can use the Monte-Carlo approximation:

$$\text{argmax}_\theta \prod_{i=1}^{m} f_\theta(\hat{y}_i|\hat{x}_i) \tag{51}$$

The resulting objective (i.e. the big product) is sometimes called **empirical risk**. This represents the usual training approach with most machine learning models, and it mostly works. Problems arise when the given sample is biased (e.g. one can collect data only under certain circumstances, the dataset is the result of a selection process, or perhaps due to pure sampling noise). In this case, the given data is not representative of the true distribution. A possible solution to this problem would be to alter the training distribution (e.g. by means of data augmentation or sample weights), so that it matches more closely the test distribution. Let the training set consist of $\{(\hat{x}_1, \hat{y}_1), (\hat{x}_2, \hat{y}_2)\}$. the corresponding optimization problem would be:

$$\text{argmax}_\theta \, f_\theta(\hat{y}_1|\hat{x}_1) f_\theta(\hat{y}_2|\hat{x}_2) \tag{52}$$

Let us pretend that the second sample occurs twice; that would lead to:

$$\text{argmax}_\theta \, f_\theta(\hat{y}_1|\hat{x}_1) f_\theta(\hat{y}_2|\hat{x}_2)^2 \tag{53}$$

In general, multiplicities show up as exponents in the training objective:

$$\text{argmax}_\theta \prod_{i=1}^{m} f_\theta(\hat{y}_i|\hat{x}_i)^{n_i} \tag{54}$$

One can use this insight to simulate a different distribution. In particular, assuming that $\tilde{p}_i$ is the true probability of sample i, and that $\hat{p}_i$ is its probability in the training set, than one can simulate a training distribution closer to the true one by solving:

$$\text{argmax}_\theta \prod_{i=1}^{m} f_\theta(\hat{y}_i|\hat{x}_i)^{\tilde{p}_i/\hat{p}_i} \tag{55}$$

Switching to log scale and to minimization, one ends uo with **sample weights**:

$$\text{argmin}_\theta - \sum_{i=1}^{m} \frac{\tilde{p}_i}{\hat{p}_i} \log f_\theta(\hat{y}_i|\hat{x}_i) \tag{56}$$

Making a good assumption about $\tilde{p}$ is difficult. A common approach is to set $\tilde{p}_i = 1 \; \forall i$, while $\hat{p}_i$ is based on the training data. This approach is called **inverse probability weighting**. This approach can be, in general, used to adjust the distribution of different classes by setting $\hat{p}_i =$ the frequency of the class

for example i. However, this approach is not limited to classes. For example, if some operating modes are under-represented, then one can make $\hat{p}_i =$ the frequency of the model for example i. Also, this method can be used to deal with selection bias. For example, if data points in a training set are not chosen at random, then one can estimate the selection probabilities $\hat{p}_i$ (e.g. via another classifier), and then one can mitigate the bias effect using sample weights. This means that one can cancel (sampling) bias based on any kind of attribute. Considering the MSE loss, the optimization problem becomes:

$$\text{argmin}_\theta - \sum_{i=1}^{m} \log k \exp\left(-\frac{1}{2}(\hat{y}_i - h_\theta(\hat{x}_i))^2\right) \tag{57}$$

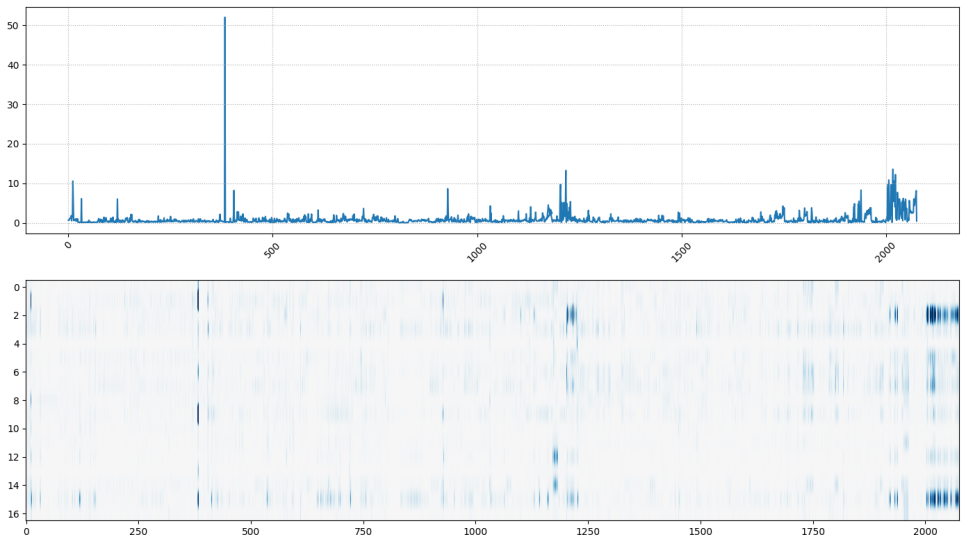
where the generic PDF has been replaced by a Normal one, and $k = 1/\sqrt{2\pi}$. Now, sample weights can be introduced as $1/\hat{\sigma}_i^2$:

$$\text{argmin}_\theta - \sum_{i=1}^{m} \frac{1}{\hat{\sigma}_i^2} \log k \exp\left(-\frac{1}{2}(\hat{y}_i - h_\theta(\hat{x}_i))^2\right) \tag{58}$$

which can be re-written as:

$$\text{argmin}_\theta - \sum_{i=1}^{m} \log k \exp\left(-\frac{1}{2}\left(\frac{\hat{y}_i - h_\theta(\hat{x}_i)}{\hat{\sigma}_i^2}\right)^2\right) \tag{59}$$

This means that sample weights with an MSE loss can be interpreted as **inverse sample variances**. This approach gives one a way to account for measurements errors. Indeed, if one knows that there is a measurement error with standard deviation $\hat{\sigma}_i$ on example i, then one can account for that by using $1/\hat{\sigma}_i^2$ as a weight.
Considering the previous dataset, one can get rid of the bias linked to operating modes by using the inverse probability weighting approach by first computing the inverse mode frequencies, and then by computing the weight for each example:

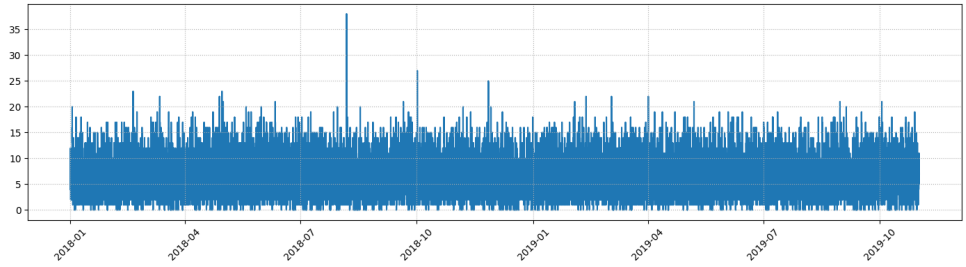


5.2 Emergency Department Arrivals

This task is about on predicting arrivals to the emergency department of a hospital. After having inspected the given dataset, one can start preprocessing. Considering the problem:

- Decisions have to be made at fixed intervals.
- What has to be computed are the expected arrivals in a given horizon.

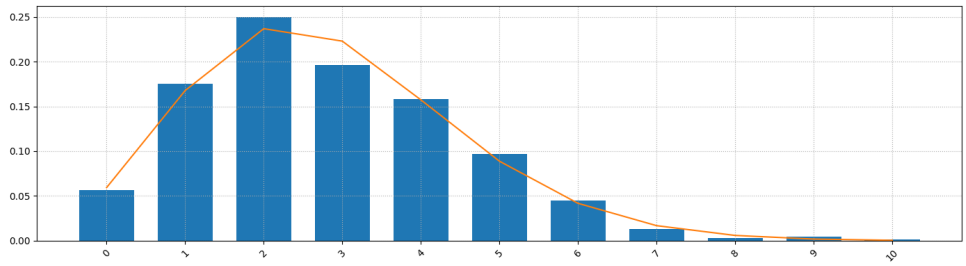
Overall, a meaningful time unit is required (i.e. some binning has to be performed). The preprocessed time-series could be something like:



Given this new data, one can focus on deploying a machine learning model that should be able to predict the total number of arrivals, where the same model can be applied to any of the individual counts (e.g. red, yellow, white, etc.). In general, this a regression problem, but in this case the MSE loss is not the best choice. Indeed, the type of distribution that should be checked in these types of problem is the Poisson distribution. This distribution models the number of occurrences of a certain event in a given interval, assuming that these events are independent and they occur at a constant rate. Such distribution is defined by a single parameter λ, namely the rate of occurrence of the events. Moreover, the distribution has a discrete support, and its probability mass function is:

$$p(k, \lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{60}$$

Both the mean and the standard deviation have the same value (i.e. λ), and the distribution skewness if $\lambda^{-\frac{1}{2}}$. Considering the emergency department problem, the target distribution seems to resemble a Poisson distribution. For example:



In order to tackle this problem, a probabilistic model of the phenomenon is built:

$$y \sim \text{Pois}(\lambda(x)) \tag{61}$$

This defines the number of arrivals in a one-hour bin. In this case, y is drawn from a Poisson distribution, but the rate of the distribution is a function of some known input x. At this points, λ can be approximated using an estimator:

$$y \sim \text{Pois}(\lambda(x, \theta)) \tag{62}$$

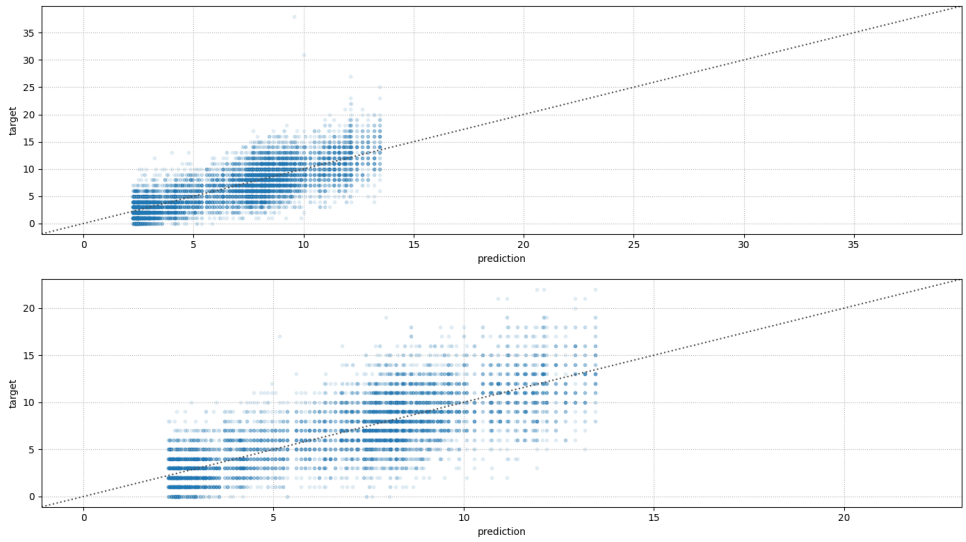
where $\lambda(x, \theta)$ can be any model. This is a hybrid approach, combining statistics and machine learning. Such model, in particular, can be trained for maximum log likelihood:

$$\text{argmin}_\theta - \sum_{i=1}^{m} \log f(\hat{y}_i, \lambda(\hat{x}_i, \theta)) \tag{63}$$

where $f(\hat{y}_i, \lambda)$ is the probability of value $\hat{y}_i$ according to the distribution, and $\lambda(\hat{x}_i, \theta)$ is the estimate rate for the input $\hat{x}_i$. In this case:

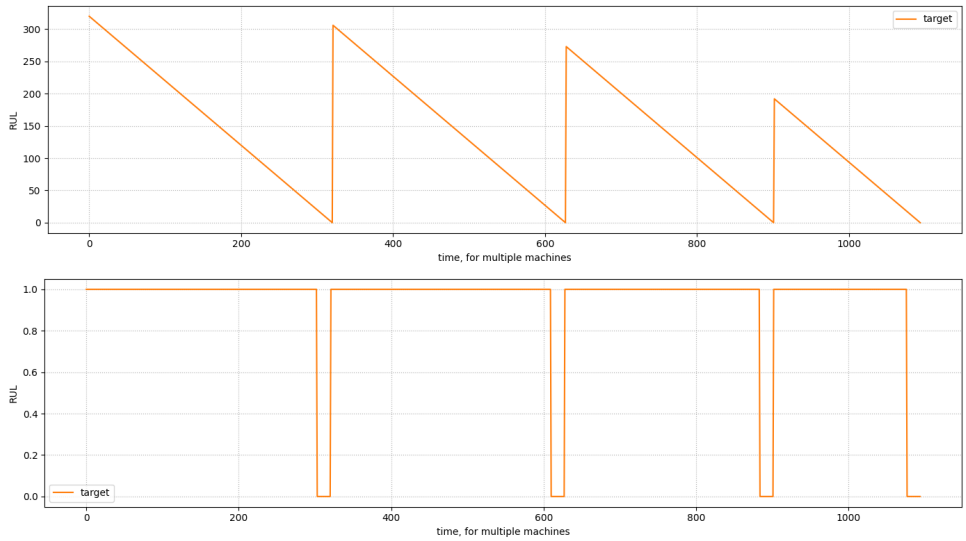
$$\text{argmin}_\theta - \sum_{i=1}^{m} \log \frac{\lambda^{\hat{y}_i} e^{-\lambda(\hat{x}_i, \theta)}}{\hat{y}_i!} \tag{64}$$

that is differentiable and can be solved via gradient descent. In order to define models that work in the aforementioned way, the `TensorFlow Probability` (`TFP`) library can be used. The following are the results on training and test set:



5.3 Survival Analysis using Neural Models

To tackle the previous RUL estimation problem, two approaches were considered: the first consisted in using a regressor, and the second consisted in using a classifier. The two target functions were the following:



By using such approaches, one was able to obtain a good maintenance policy, but no well-grounded RUL estimate. In such cases, the phenomena was treated as a deterministic one, and the estimators were built without any underlying analysis. A better way to tackle the problem would be the following:

1. Start by defining a probabilistic model.
2. Use machine learning to approximate key components of such model.
3. Use the model and the approximators to make probabilistic predictions.

This approach is called **conditional survival analysis**. In particular, the remaining survival time of an entity if modeled as a single random variable T with unknown distribution:

$$T \sim P(T|t) \tag{65}$$

where T has support in $\mathbb{R}^+$, and t is the time at which the estimation is performed. Survival is usually based on other factors: behavior in the past, $X_{\leq t}$, and behavior in the future, $X_{>t}$. Formally:

$$T \sim P(T|X_{\leq t}, t, X_{>t}) \tag{66}$$

Future behavior cannot be accessed at estimation time, and intuitively, such behavior affects the estimate as noise. Formally, one can average out its effect by means of **marginalization**:

$$T \sim \mathbb{E}_{X_{>t}}[P(T|X_{\leq t}, t)] \tag{67}$$

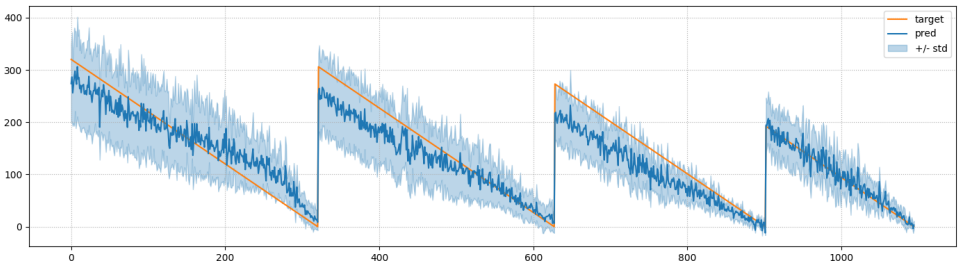
In the section about RUL, the regressor was trained using MSE (i.e. $T \sim \mathbb{E}_{X_{>t}}[\mathcal{N}(\mu(X_t), \sigma)]$). In such a case:

- One is only considering a single X_t, rather than $X_{\leq t}$.
- One is disregarding time as an input.
- One assumed a Normal distribution with fixed variance.

The second point can be solved by adding time to the input columns of the dataset. The third point can be solved by making use of a machine learning model capable of estimating variance:

$$T \sim \mathbb{E}_{X_{>t}}[\mathcal{N}(\mu(X_t), \sigma(X_t, t))] \tag{68}$$

In practice, one can use a conventional model to estimate μ and σ. In this way, the model will learn how σ depends on the input. By training such model using `TFP`, one is able to obtain the following results:



This approach works already very well. However, going for a Normal distribution could not be convincing. Moreover, the RUL could depend strongly on when a defect arises. In such a case, in the early part of each run one might be accounting too much for what happened in the future (i.e. overfitting). Another approach could be the one based on a **survival function** in order to study the distribution of T. The survival function of a variable T is defined as:

$$S(t) = P(T > t) \tag{69}$$

namely, as the probability that the entity survives at least until time t. Accounting for conditioning factors:

$$S(t, X_{\leq t}) = P(T > t|X_{\leq t}) \tag{70}$$

This means that such functions cannot account for the future, but also that it cannot overfit due to poor marginalization. If one assumes discrete time, then S can be factorized:

$$S(t, X_{\leq t}) = (1 - \lambda(t, X_t))(1 - \lambda(t-1, X_{t-1})) \cdots \tag{71}$$

where λ is called **hazard function** and it is a conditional probability. In particular, $\lambda(t, X_t)$ is the probability of not surviving at time t given that the entity has survived until time $t-1$ (i.e. $\lambda(t, X_t) = P(T > t | T \geq t-1, X_t)$). At this point, an estimator $\lambda(t, X_t, \theta)$ can be trained in order to estimate the hazard function. This requires no assumption on the distribution, it does not risk overfitting due to poor marginalization, and it makes sense even if one does not observe a "death" event. Additionally, it is not immediate to use λ to obtain a RUL estimate, but one can use it to approximate the chance of surviving n steps from the current time:

$$S(t+n)/S(t) = P(T > t+n | T \geq t-1) \tag{72}$$

The λ estimator can be trained by modeling the probability of a survival event. Supposing the k-th experiment in the given dataset ends at time e_k, then the corresponding probability according to the estimator is:

$$\lambda(e_k, \hat{x}_{e_k}, \theta) = \prod_{t=1}^{e_k - 1} (1 - \lambda(t, \hat{x}_{kt}, \theta)) \tag{73}$$

This is the probability of surviving all time steps from 1 to $e_k - 1$ and of not surviving at time e_k. The $\hat{x}_{kt}$ is the available input data for experiment k at time t. Assuming to have m experiments:

$$\text{argmax}_\theta \prod_{k=1}^{m} \lambda(e_k, \hat{x}_{e_k}, \theta) = \prod_{t=1}^{e_k - 1} (1 - \lambda(t, \hat{x}_{kt}, \theta)) \tag{74}$$

Let

$$\hat{d}_{kt} = \begin{cases} 1 & \text{if } t = e_k \\ 0 & \text{otherwise} \end{cases} \tag{75}$$

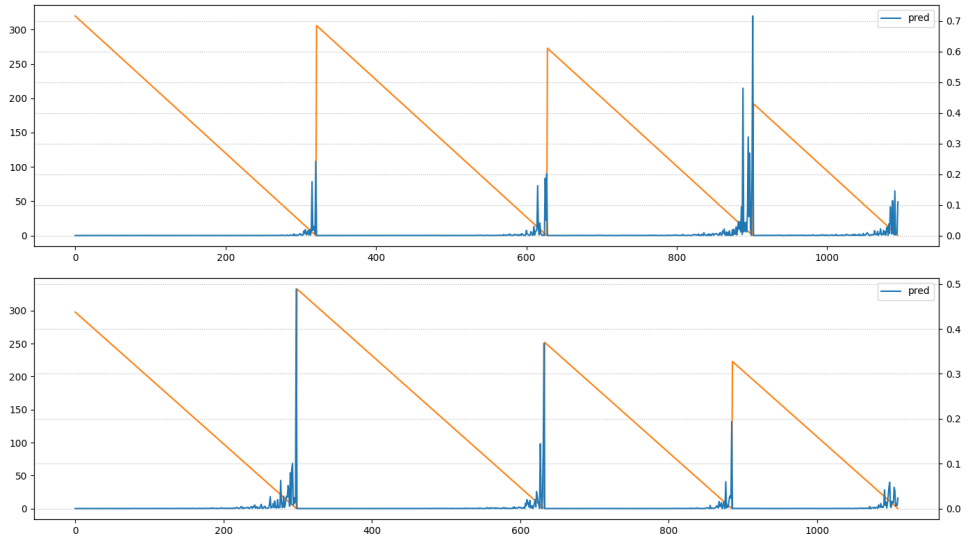
then:

$$\text{argmax}_\theta \prod_{k=1}^{m} \prod_{t=1}^{e_k} \hat{d}_{kt} \lambda(e_k, \hat{x}_{e_k}, \theta) + (1 - \hat{d}_{kt})(1 - \lambda(t, \hat{x}_{kt}, \theta)) \tag{76}$$

which is equivalent to:

$$\text{argmin}_\theta \sum_{k=1}^{m} \sum_{t=1}^{e_k} \hat{d}_{kt} \log \lambda(e_k, \hat{x}_{e_k}, \theta) + (1 - \hat{d}_{kt}) \log(1 - \lambda(t, \hat{x}_{kt}, \theta)) \tag{77}$$

that corresponds to a binary cross-entropy minimization problem. What one needs to do is to consider all samples in the dataset individually, and to attach to them a class given by $\hat{d}_{kt}$. Once the estimator has been trained, the hazards for training and test sets can be inspected:



At this point, the probability of being still up in n steps is:

$$S(t+n)/S(t) = \prod_{h=0}^{n}(1 - \lambda(t+h, X_{t+h})) \tag{78}$$

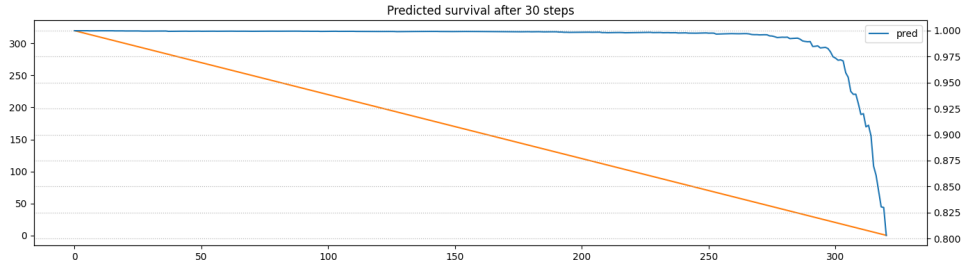
which can be approximated (by using the estimator for the k-th run) with:

$$S(t+n)/S(t) \simeq \prod_{h=0}^{n}(1 - \lambda(t+h, \hat{x}_{k,t+h}, \theta)) \tag{79}$$

However, such formula requires access to future values of the X_t variable, so one can approximate with:

$$S(t+n)/S(t) \simeq \prod_{h=0}^{n}(1 - \lambda(t+h, \hat{x}_{kt}, \theta)) \tag{80}$$

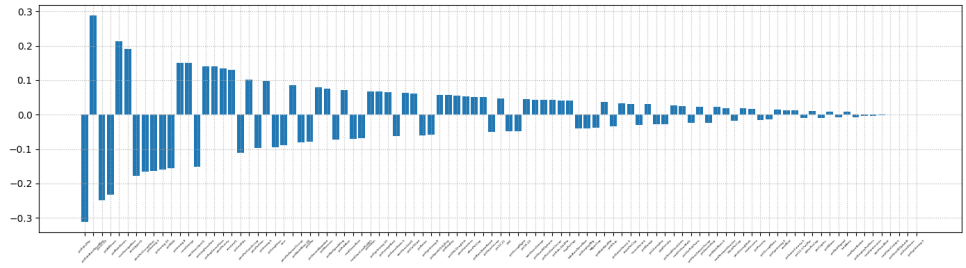
where the sample values $\hat{x}_{kt}$ are kept fixed. On a short horizon (i.e. small n), the error is typically limited.



6 Constraints in Machine Learning Models

6.1 Fairness in Machine Learning

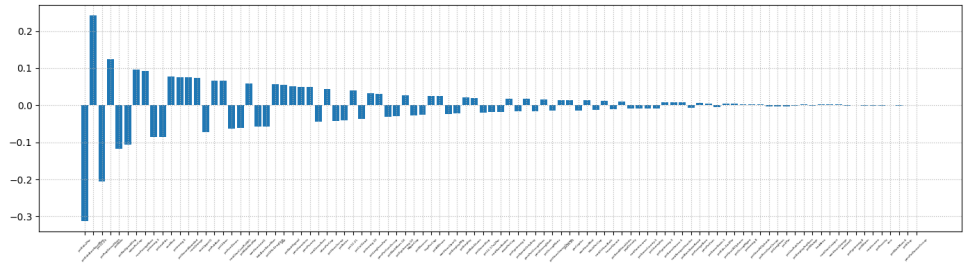
To discuss fairness in data-driven methods, the "crime" UCI dataset will be used. As data-driven systems become more pervasive, they have the potential to significantly affect social groups. The given dataset has one categorical input which is called `race`, that will be the attribute used to check for discrimination. After having preprocessed the dataset, a linear regressor can be used to solve the problem of predicting the number of violent offenders per 100k people. Having trained the model, one can evaluate the importance of each input attribute by inspecting the weights of the model:



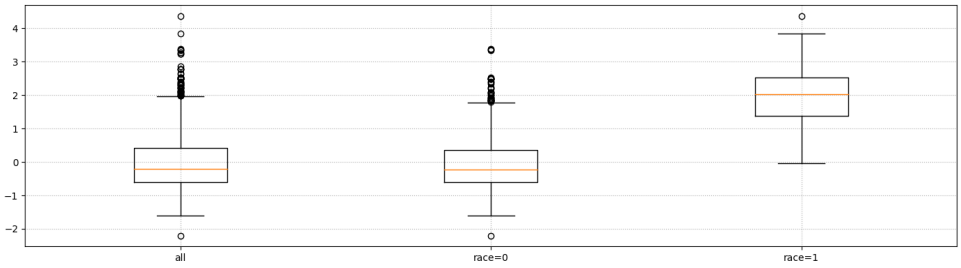
If all attributes are standardized/normalized, then the larger the weight, the larger the impact. From the plot above once can notice that there are many large-ish weights. This can be solved by adding an L1 regularizer to obtain **LASSO** (**regression**):

$$L(\hat{y}, f(x, \theta)) = \|\theta^T x - \hat{y}\|_2^2 + \alpha\|\theta\|_1 \tag{81}$$

Lasso weights are sparse (i.e. only a few attributes will have a significant impact):



Unfortunately, the attribute `race` is not within the most important attributes. Measuring fairness, in general, is complicated, and several fairness metrics have been proposed. The idea is to check whether different groups, as defined by the value of a protected attribute (e.g. `race`), are associated to different predictions. It turns out that the model treats the groups differently, even is `race` is not an important attribute:



Therefore, checking the important attributes is not enough. Given a set of categorical protected attribute (indexes) J_p, the **distance impact discrimination index** (for regression) is give by:

$$\text{DIDI}_r = \sum_{j \in J_p} \sum_{v \in D_j} \left| \frac{1}{m} \sum_{i=1}^{m} y_i - \frac{1}{|I_{j,v}|} \sum_{i \in I_{j,v}} y_i \right| \tag{82}$$

where D_j is the domain of attribute j, and $I_{j,v}$ is the set of example such that attribute j has value v. The idea is to penalize the group predictions for deviating from the global average.

6.2 Constrained ML via Lagrangians

The idea is to train an accurate regressor ($L = \text{MSE}$):

$$\text{argmin}_\theta \, \mathbb{E}[L(\hat{y}, f(\hat{x}, \theta))] \tag{83}$$

and to measure fairness via the DIDI metric ($\text{DIDI}(y)$), in such a way that $\text{DIDI}(y) \leq \varepsilon$. This last information can be used to re-state the training problem:

$$\text{argmin}_\theta \{\mathbb{E}[L(\hat{y}, f(\hat{x}, \theta))] | \, \text{DIDI}(f(\hat{x}, \theta)) \leq \varepsilon\} \tag{84}$$

Training is now a constrained optimization problem, and after training the constraint will be distilled in the model parameters. Given a problem of the form:

$$\text{argmin}_\theta \{L(y) | g(y) \leq 0\} \quad \text{with } y = f(\hat{x}, \theta) \tag{85}$$

where L is the loss, $\hat{x}$ is the training input, y is the model's output, g is a constraint function, one way to deal with such problem is to rely on a **Lagrangian relaxation**. The main idea is to turn the constraints into penalty terms: from the original constrained problem one can obtain al unconstrained problem:

$$\text{argmin}_\theta \, L(y) + \lambda^T \max(0, g(y)) \quad \text{with } y = f(\hat{x}, \theta) \tag{86}$$

The new loss function is known as a Lagrangian. $\max(0, g(y))$ is sometimes known as **penalizer** (or Lagrangian term), and λ is a vector of multipliers. In particular, when the constraint is satisfied (i.e. $g(y) \leq 0$), then the penalizer is 0, while when the constraint is violated, the penalizer is > 0. Hence, in the feasible area, one still has the original loss, while in the infeasible area, one incurs a penalty that

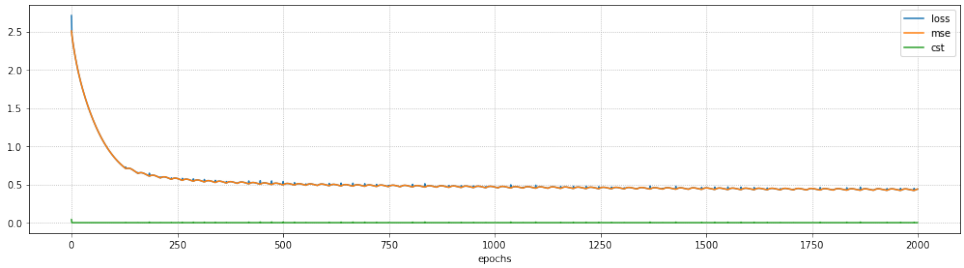
can be controlled using λ. Thus, assuming that $L(y)$ stays finite, if one chooses λ large enough, then one can guarantee that a feasible solution is found. This is the basis of the classical penalty method. For example, considering the following problem:

$$\text{argmin}_\theta\{\mathbb{E}[L(\hat{y}, f(\hat{x}, \theta))] | \, \text{DIDI}(f(\hat{x}, \theta)) \leq \varepsilon\} \tag{87}$$

the Lagrangian term for the constraint is:

$$\lambda \max(0, \text{DIDI}(f(\hat{x}, \theta)) - \varepsilon) \tag{88}$$

In this case, having only one constrain, λ is a scalar. In principle one can implement the approach with a custom loss function (or with a custom model).



The constraint is satisfied (and the accuracy reduced, as expected).

6.2.1 Lagrangian Dual Framework

Based on what has done so far, one is trying to solve the aforementioned optimization problem by using gradient descent. However, a large λ may cause the gradient to be unstable. Therefore, with a convex model, one should still reach convergence (slowly), and with a non-convex model, one may end up in a poor local optimum. To solve such problem, one can think of increasing λ gradually, and this leads to the classical penalty method:

1. $\lambda^{(0)} = 1$.
2. $\theta^{(0)} = \text{argmin}_\theta\{L(y) + \lambda^{(0)T} \max(0, g(y))\}$ with $y = f(\hat{x}, \theta)$.
3. For $k = 1, \ldots, n$:
   (a) If $g(y) \leq 0$ then stop, otherwise $\lambda^{(k)} = r\lambda^{(k)}$, with $r \in (1, \infty)$.
   (b) $\theta^{(k)} = \text{argmin}_\theta\{L(y) + \lambda^{(k)T} \max(0, g(y))\}$ with $y = f(\hat{x}, \theta)$.

However, λ grows quickly and may still become problematically large, and early and late stages of gradient descent may call different values of λ. A gentler approach consists in using gradient ascent for the multipliers. Let us consider the modified loss:

$$\mathcal{L}(\theta, \lambda) = L(\hat{y}, f(\hat{x}, \theta)) + \lambda^T \max(0, g(f(\hat{x}, \theta))) \tag{89}$$

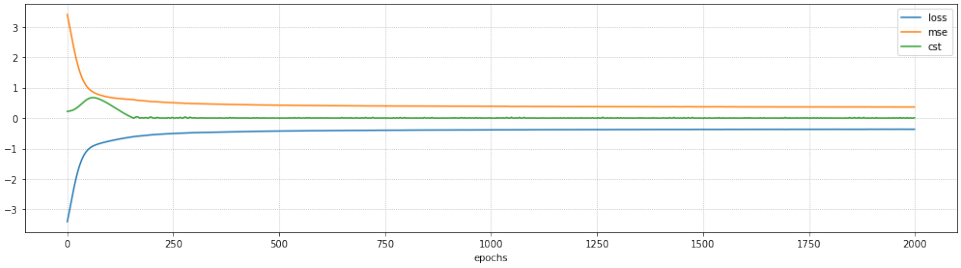
The gradient of such function is:

$$\nabla_\lambda \mathcal{L}(\theta, \lambda) = \max(0, g(f(\hat{x}, \theta))) \tag{90}$$

Thus, for satisfied constrains the partial derivative is 0, and for violated constraints it is equal to the violation. Finally, the problem can be solved by alternating gradient descent and ascent:

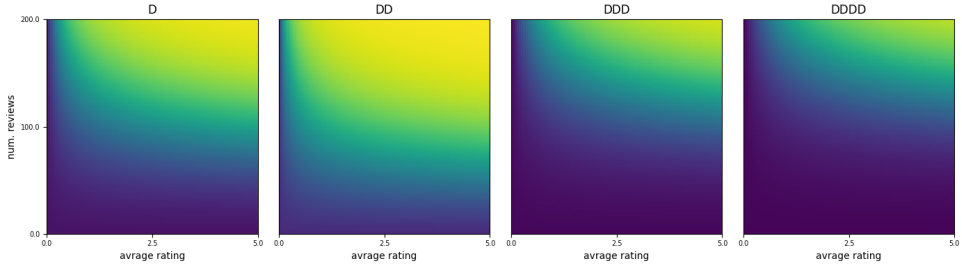
1. $\lambda^{(0)} = 0$.
2. For $k = 1, \ldots, n$ (or until convergence):
   (a) Obtain $\lambda^{(k)}$ via an ascent step with $\nabla_\lambda \mathcal{L}(\lambda, \theta^{(k-1)})$.
   (b) Obtain $\theta^{(k)}$ va a descent step with $\nabla_\theta \mathcal{L}(\lambda^{(k)}, \theta)$.

The new approach leads to fewer oscillations at training time:



6.3 Click-Through Rate Prediction

Let us consider an automatic recommendation problem: given a set of restaurant indexed on a web platform, one would like to estimate how likely a user is to actually open the restaurant card. Each row of the particular dataset represents one visualization event, and the click rate can be inferred by the number of clicks for each restaurant. In particular, it can be noticed that the click rate grows with the average rating and the number of reviews, and that average priced restaurants are the most clicked:



By checking the distributions of training and test sets, one can immediately notice that the two distributions are very different, mainly due to app usage data: users seldom scroll through all search

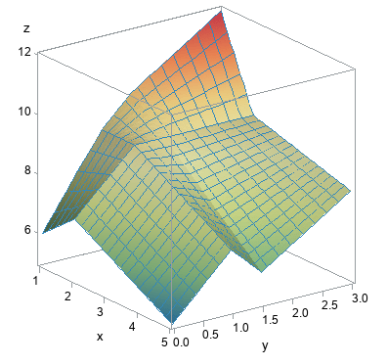
results, so their clicks will be biased toward high ranked restaurant. Any training set obtained in this fashion will be strongly biased. However, click rate prediction is typically used for ranking search results, meaning that there is the need to evaluate also less viewed restaurants. The baseline approach is based on:

- Data preprocessing.
- Model (MLP) definition and training.

This is not a classification problem, so accuracy is not a good metric. The output of the system is means to be interpreted as a probability, so rounding to obtain a deterministic prediction may be too restrictive. Instead of accuracy, one could make evaluate the model using a ROC curve. In general, given a ROC curve, the larger the Area Under Curve (AUC), the better the performance. It turns out that the performances of the model are not bad, however the AUC testing score is lower than the training one. The main problem of the given data is that in some areas of the input space, increasing an attribute has the opposite of the expected effect.

6.4 Lattice Models

Lattice models are a form of piecewise linear interpolated model:



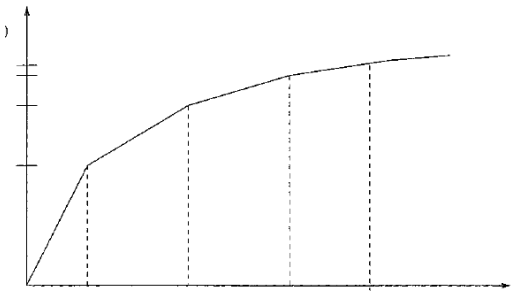
Such models are defined via a grid over their input variables, their parameters are the output values at each grid point, and the output values for the input vectors not corresponding to a point of the grid is the linear interpolation of neighboring grid points. In general, these models can represent non-linear multivariate functions and can be trained using gradient descent. Moreover, the grid is defined by splitting each input domain into intervals: the domain of variable x_i is split by choosing a fixed set of n_i "knots". TensorFlow provides a useful API that can be used to define and train lattice models: the `tensorflow-lattice` API. Once defined, the lattice model is trained and tested on the aforementioned restaurant dataset: the final results are very similar (i.e. sample problems) to ones obtained using a MLP.

6.4.1 Calibration

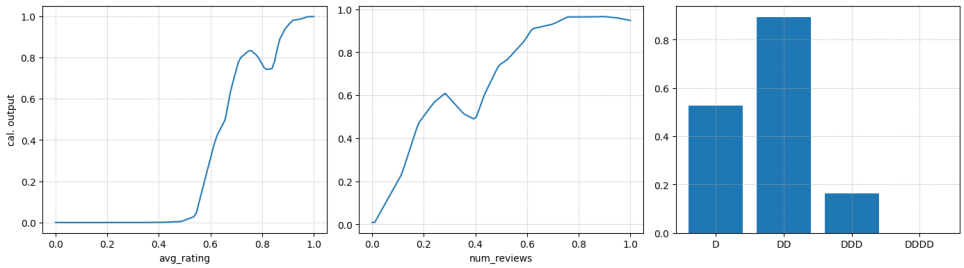
In a lattice model, the number of grid points is given by:

$$n = \prod_{i=1}^{m} n_i \tag{91}$$

hence the parameter number scales exponentially with the number of inputs, so that modeling complex non-linear function seems to come at a steep cost. This problem can be dealt with by applying a calibration step to each input variable individually. In particular, calibration for number attributes consists in applying a piecewise linear transformation to each input:



which is essentially a 1D lattice. On the other hand, calibration for categorical inputs consists in applying a map, where categorical inputs must be encoded as integers. In general, calibration allows one to use fewer knots in the lattice, and to get more regular results (i.e. higher bias and lower variance), which might be an advantage for out-of-distribution generalization. Indeed, by using this new lattice model, the AUC score for the test set is higher. However, the learned calibration functions still violate the expected monotonicities:



6.4.2 Shape Constraints

Lattice models are well suited to deal with shape constraints, where shape constrains are restrictions on the input-output function, such as monotonicity and convexity/concavity. Thus, these constrains can be used to fix the aforementioned calibration issues. In particular,shape constraints translate into constraints on the lattice parameters. Let $\theta_{i,k,\bar{i},\bar{k}}$ be the parameter for the k-th knot of input i, while

all the remaining attributes and knots (i.e. $\bar{i}$ and $\bar{k}$) are fixed, then increasing monotonicity translates to:

$$\theta_{i,k,\bar{i},\bar{k}} \leq \theta_{i,k+1,\bar{i},\bar{k}} \tag{92}$$

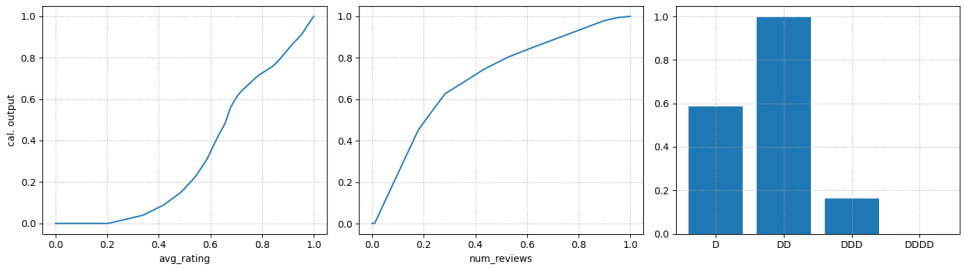
While convexity translate to:

$$(\theta_{i,k+1,\bar{i},\bar{k}} - \theta_{i,k,\bar{i},\bar{k}}) \leq (\theta_{i,k+2,\bar{i},\bar{k}} - \theta_{i,k+1,\bar{i},\bar{k}}) \tag{93}$$

At this point, one can expect:

- A monotonic effect of the average rating (i.e. restaurants with a high rating will be clicked more often).
- Diminishing returns from the the number of reviews (i.e. n reviews will be linked to much more clicks than m reviews, where $n > m$).
- Partial orders on categories (i.e. more clicks for reasonably priced restaurants). On categorical attributes one can enforce partial order constraints: each pair (i, j) translates into an inequality $\theta_i \leq \theta_j$.

At this point the calibration functions are the following:



where all monotonicities are respected.

6.5 Ordinary Differential Equations

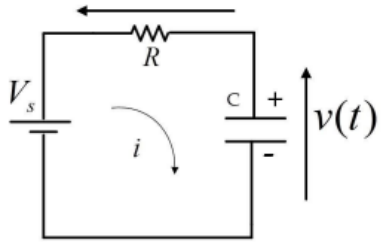
An **ordinary differential equation** (**ODE**) is any equation of the form:

$$\dot{y} = f(y, t) \tag{94}$$

where y is the state variable, and f is a function, providing the gradient of the state variable. The t variable, typically, represents time. An **initial value problem** consists of an ODE and a initial condition, $y(0) = y_0$. Such problems can be solved (i.e. integrated) exactly, using symbolic approaches, e.g.:

$$\dot{y} = a, y(0) = b \quad \Rightarrow \quad y(t) = ay + b \tag{95}$$

or they can be solved numerically. For example, considering a simple RC circuit:



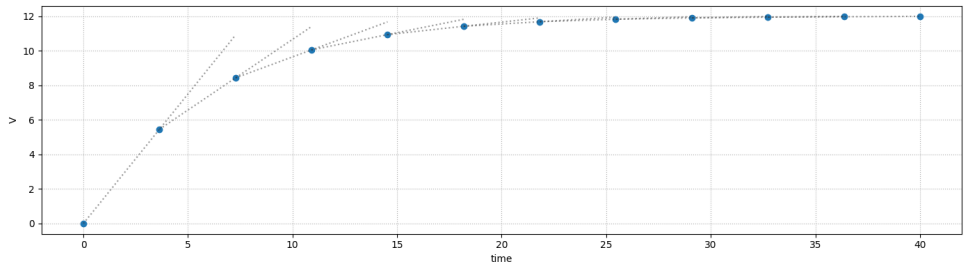
it's dynamic behavior is described by the ODE:

$$\dot{V} = \frac{1}{\tau}(V_s - V) \tag{96}$$

where $\tau = RC$. The simplest numerical approach for ODEs is called **Euler method**, and it is based on the following points:

- It considers a fixed sequence of evaluation points $\{t_k\}_{k=0}^{n}$.
- It uses a linear approximation for $y(t)$ within each interval $[t_k, t_{k+1}]$.
- It approximates the slope with the gradient f at time t_k.

Having defined the $f(y,t)$, the initial state y_0 and the evaluation points $\{t_i\}_{i=0}^{n}$, the Euler method will yield:



where the dots represent evaluated states, and the slope of the lines corresponds to the gradient at each step. However, the Euler method is one of the worst integration approaches in terms of accuracy.

6.5.1 Learning ODEs

The parameters of an ODE can be estimated from data. Formally, this training problem amounts to solving:

$$\text{argmin}_\theta\{L(y(\hat{t}), \hat{y}) | \dot{y} = (y, t, \theta), y(0) = \hat{y}_0\} \tag{97}$$

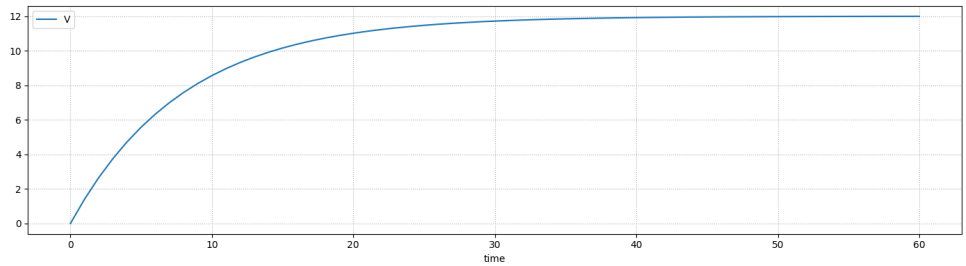
where $\{\hat{t}_k\}_{k=0}^n$ is a sequence of points for which measurements are available, $\{\hat{y}_k\}_{k=0}^n$ are the corresponding state measurements, f is a parameterized gradient function, and L is a loss function (e.g. MSE). The goal is to choose the parameters (e.g. τ and V_s) so as to be close to the real integrated ODE. Once can start from observing that every step in the Euler method is differentiable (if f is differentiable):

$$y_k = y_{k-1} + (t_k - t_{k-1}) f(y_{k-1}, t_{k-1}) \tag{98}$$

Thus, a viable approach is to discretize, and then optimize:

1. Use an automatic differentiation engine to solve the initial value problem using a numerical method.
2. Compute the loss L.
3. Update the parameters using gradient descent.

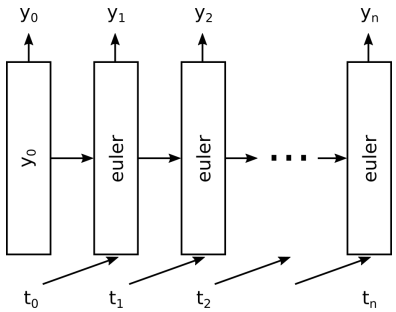
Considering the aforementioned RC circuit, let's start by building a high-quality ground truth sequence using `odeint` from `scikit learn`.



Once the values $\{V_k\}_{k=0}^n$ are produced, the gradient function $f(y, t, \theta)$ can be interpreted as a layer type, and a `keras.Model` object can be used to encode the Euler methods:

$$y(\hat{t}_k) = y(\hat{t}_{k-1}) + (\hat{t}_k - \hat{t}_{k-1}) f(y(\hat{t}_{k-1}), \hat{t}_{k-1}, \theta) \tag{99}$$

In particular, each step of the method can be seen as a layer instance, and all instances share the same weights:



This network resembles a recurrent neural network. The input includes the initial state y_0 and the evaluation points $\{\hat{t}_k\}_{k=0}^n$. The output consists of the sequence of state evaluations $\{y_k\}_{k=0}^n$. In the RC circuit case, one has:

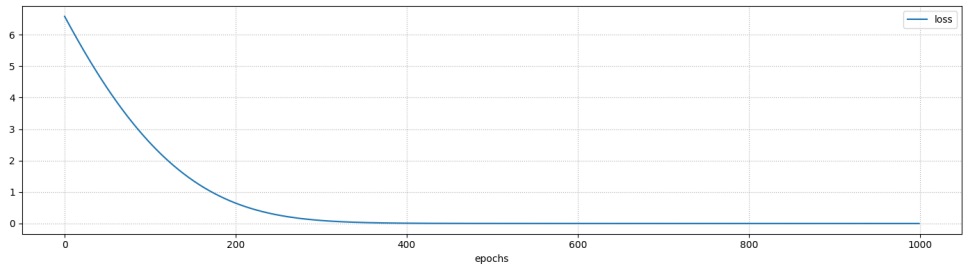
$$\text{argmin}_{\tau,V_s} L(y(\hat{t}), \hat{y}) \quad \text{subject to } \dot{y} = \frac{1}{\tau}(V_s - y),\ y(0) = y_0 \tag{100}$$

where the parameters to be learned are τ and V_s. For both of these parameters, negative values make no sense. Moreover, the initial guesses should be reasonable. To meet these conditions, the following re-formulation is adopted:

$$\tau = \sigma_\tau e^{\theta_\tau} \tag{101}$$

$$V_s = \sigma_{V_s} e^{\theta_{V_s}} \tag{102}$$

where the parameters to be learned are now θ_τ and θ_{V_s}. In particular, using an exponential ensures non-negative values, and the scaling factors σ_τ and σ_{V_s} are user-provided. by modeling all this in `keras`, one can obtain the following behavior:



This approach seems to be working, but there are a few issues:

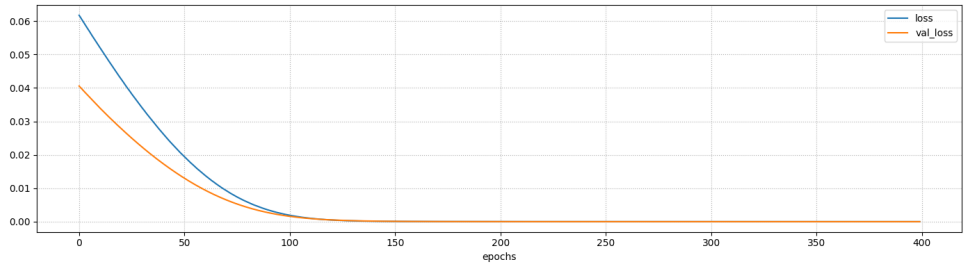
1. The convergence is slow.
2. There is no possibility of using a validation set.
3. The predicted parameters are still not correct.

6.5.2 Better Learning ODEs

The first two issues can be addressed using a re-formulation. In particular, the sequence of measurements $\{\hat{y}_k\}_{k=0}^n$ can be viewed as a sequence of pairs $\{(\hat{y}_{k-1}, \hat{y}_k)\}_{k=1}^n$, each referring to a distinct ODE (i.e. $\dot{y} = f(y_k, t, \omega)$), with all ODEs sharing the same parameter vector ω. With this approach, the training problem can be formulated as:

$$\text{argmin}_\omega \sum_{k=1}^{n} L(y_k(\hat{t}_k), \hat{y}_k) \quad \text{subject to } \dot{y} = f(y_k, t, \omega) \text{ and } y_k(\hat{t}_{k-1}) = \hat{y}_{k-1}\ \forall k \in [1, n] \tag{103}$$

In practice, one assumes to deal with multiple initial value problems. The new training problem, however, is not exactly equivalent to the old one, since by re-starting at each step, the compound errors are disregarded. By applying such method, the results are the following:



This approach has significant computational advantages, since here one can use a shallow network rather than a deep one (i.e. lower training time, no vanishing/exploding gradient). Lastly, the third problem is mainly due to the fact that the Euler method is inaccurate. The solution is to use a more accurate numerical method.

6.6 Universal Ordinary Differential Equations

In general, there several advantages in using a neural engine to train ODEs. For example, high-dimensionality is not a problem, one can train ODEs with multiple parameters, and one can approximate ODEs with weaker methods (e.g. the Euler method). However, the real advantage comes from the ability to incorporate black-box functions. This is sometimes called a **universal ordinary differential equation** (**UDE**):

$$\dot{y} = f(y, t, U(y, t)) \tag{104}$$

where U is a trainable universal approximator (e.g. a neural network) which produces some parameters. This, in particular, is an example of **physics informed neural networks**, where f encodes knowledge about the system behavior and U can be trained to learn implicit knowledge from data. Let's consider a SIR model (i.e. an epidemic model)

$$\dot{S} = -\beta \frac{1}{N} SI \tag{105}$$

$$\dot{I} = +\beta \frac{1}{N} SI - \gamma I \tag{106}$$

$$\dot{R} = +\gamma I \tag{107}$$

and say that one's goal is to control the epidemic via non-pharmaceutical interventions (NPI) (e.g. using masks, social distancing, etc.). These interventions have an effect of β, and they change over time. The effect of NPIs can be modeled via a UDE model:

$$\dot{S} = -\beta(t)\frac{1}{N}SI \tag{108}$$

$$\dot{I} = +\beta(t)\frac{1}{N}SI - \gamma I \tag{109}$$

$$\dot{R} = +\gamma I \tag{110}$$

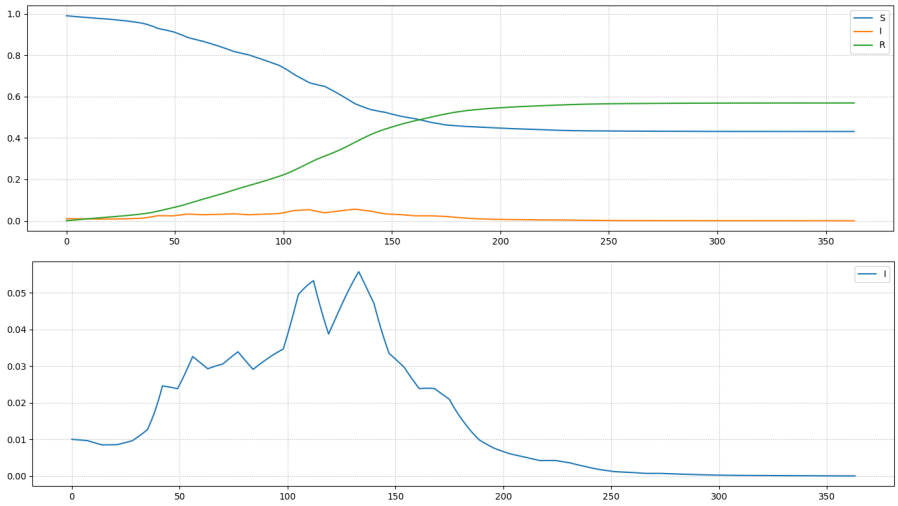
where $U(y,t)$ corresponds to $\beta(t)$. Assuming that 1% of the population is infected (i.e. $S_0 = 0.99$, $I_0 = 0.01$, $R_0 = 0$), a recovery time of 10 days (i.e. $\gamma = 1/10$), and a natural value of $\beta = 0.23$ one can make up a synthetic use case. In this use case, let's also assume that active NPIs, defined by the I set, are such that:

$$\hat{\beta}(t) = \beta \prod_{i \in I} e_i \tag{111}$$

By randomly sampling NPIs values, one can build a 52-week dataset having the following form:

	S	I	R	week	masks-indoor	masks-outdoor	dad	bar-rest	transport	beta
0.0	0.990000	0.010000	0.000000	0	0	0	1	1	0	0.0966
1.0	0.989046	0.009956	0.000998	0	0	0	1	1	0	0.0966
2.0	0.988098	0.009911	0.001991	0	0	0	1	1	0	0.0966
3.0	0.987154	0.009866	0.002980	0	0	0	1	1	0	0.0966
4.0	0.986216	0.009820	0.003964	0	0	0	1	1	0	0.0966
5.0	0.985283	0.009773	0.004944	0	0	0	1	1	0	0.0966
6.0	0.984356	0.009725	0.005919	0	0	0	1	1	0	0.0966
7.0	0.983434	0.009677	0.006889	1	0	0	0	1	1	0.0828

The S, I, R curves can be inspected:



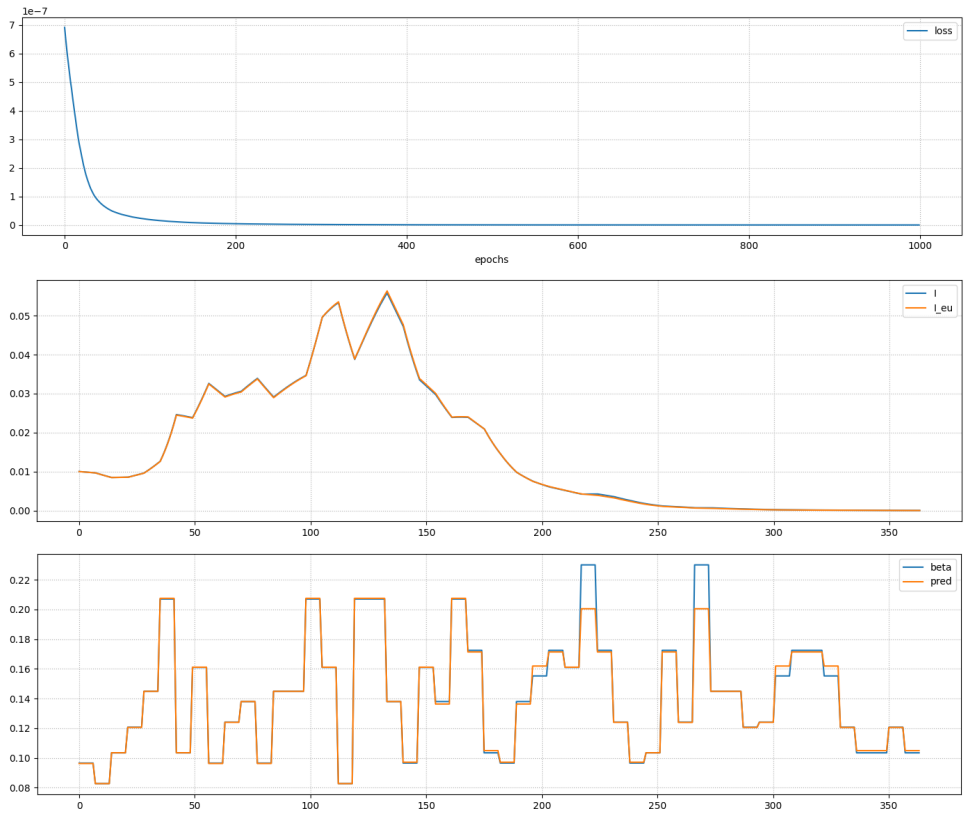
When $\hat{\beta}(t)/\gamma > 1$ one has a true epidemic behavior, while when $\hat{\beta}(t)/\gamma \leq 1$ the number of new cases always drops.
In principle, one could be build a custom layer for the aforementioned UDE, where the input for $\beta(t)$ is time. In practice, however, the input one cares about are the active NPIs. Thus, a more accurate formulation would be:

$$\dot{S} = -\beta(NPI(t))\frac{1}{N}SI \tag{112}$$

$$\dot{I} = +\beta(NPI(t))\frac{1}{N}SI - \gamma I \tag{113}$$

$$\dot{R} = +\gamma I \tag{114}$$

The custom layer should take as input S, I, R and t, use t to retrieve $NPI(t)$, and then compute the gradient. In practice, it is easier to supply the NPIs as additional inputs. Having done this in `keras`, the model can be trained and evaluated:



The estimates are good, except for the later part of the sequence. This is mostly due to the plain MSE as the loss function.

7 Machine Learning and Combinatorial Optimization

7.1 Epidemic Control

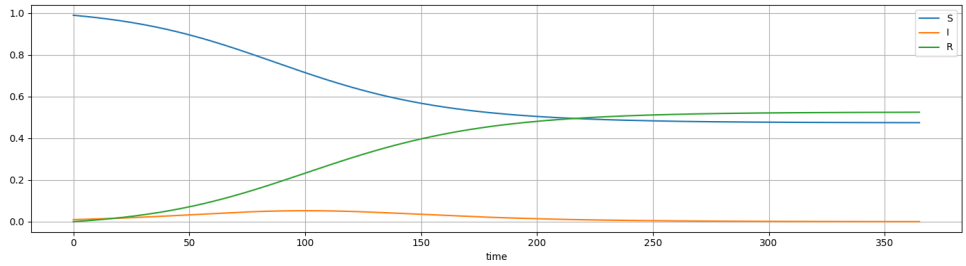
Here, an epidemic control problem is considered. This problem can be formulated as an optimization problem that can be tackled by combining machine learning and combinatorial optimization. For this case, a SIR model will be used as a simulator, where the population is divided into three groups (i.e. susceptibles, infected and recovered). The classical SIR model is a dynamic system, where the size of the three groups evolved over time according to an ODE:

$$\dot{S} = -\beta \frac{1}{N} SI \tag{115}$$

$$\dot{I} = +\beta \frac{1}{N} SI - \gamma I \tag{116}$$

$$\dot{R} = \gamma I \tag{117}$$

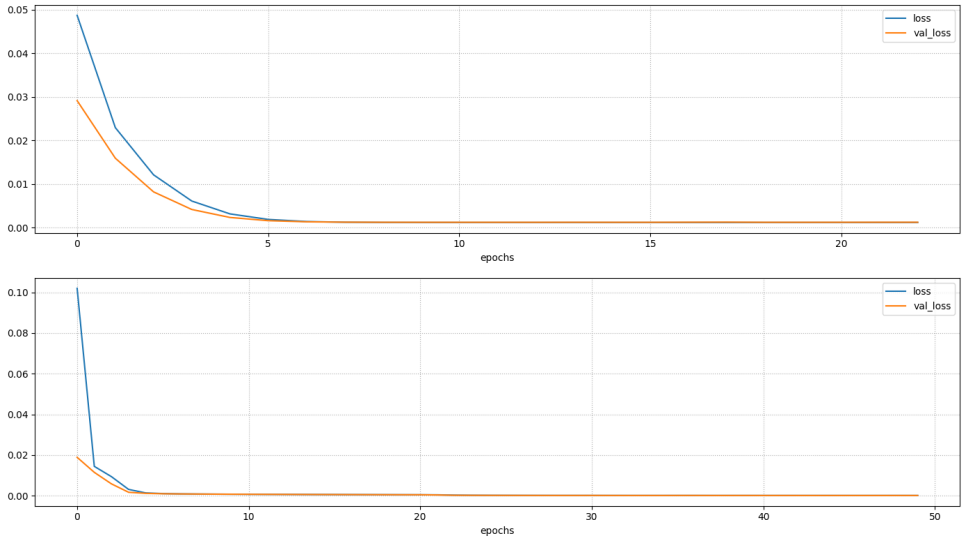
where S, I, R refer to the size of each component, N is the population size, β is the infection rate and γ is the recovery rate. Moreover, the ratio $R_0 = \beta/\gamma$ is called basic reproductive number and determines whether one has a proper epidemic behavior: if $R_0 > 1$ infections grow before falling, otherwise they only decrease. An example of true epidemic behavior could be given by $R_0 = 1.4$. By initializing $S_0 = 0.99$, $I_0 = 0.01$, $R_0 = 0$, $\beta = 0.1$, and $\gamma = 1/14$, one can plot the dynamics for one year:



Here, the S compartment monotonically decreases, the R compartment monotonically increases, and the I compartment initially grows, before decreasing again.

7.1.1 Machine Learning for Epidemic Control

The idea is to learn a machine learning model that can then be embedded in a larger optimization problem to obtain a set of actions to control the epidemics. To do this, a dataset has to be built. This dataset will focus on Non-Therapeutic Interventions (NPIs), which usually affect β, and it will also focus making predictions at weekly intervals. Thus, the dataset input will contain the state (S_0, I_0, R_0) at the beginning of the week and the value of β, while the dataset output will contain the state (S, I, R) after one week. To build this, one can run a simulation with inputs of the form (S, I, R, β), making sure to sample points from all over the input space. After having generated the dataset, one can try learning a linear regressor and a shallow neural network:



7.1.2 Encoding Machine Learning Models

The basic approach to embed the machine learning model into an optimization model is based on two observations:

- A neural network is a collection of connected neurons.
- Each neuron has to be encoded using a given optimization method.

A generic ReLU neuron is such that:

$$y = \max(0, w^T x + b) \tag{118}$$

where w is the vector of weights and b is the bias. This can be encoded by introducing a variable for each input, introducing a variable for each output, and modeling the sum and max operators. By adopting a mixed-integer linear programming encoding:

$$y - s = wx + b \tag{119}$$
$$z = 1 \Rightarrow s \leq 0 \tag{120}$$
$$z = 0 \Rightarrow y \leq 0 \tag{121}$$
$$y, s \geq 0, x \in \mathbb{R}^n, z \in \{0, 1\} \tag{122}$$

where s is an auxiliary slack variable and z is an auxiliary binary variable. The implications are called indicator constraints. In general, if $z = 1$, it means that the neuron is active, in such a case, s is force to 0. Thus, $y = wx + b$ and $wx + b$ is non-negative. In case $z = 0$, it means that the neuron is inactive, and in this case $y = 0$. Thus, $s = wx + b$ and $wx + b$ is negative. The aforementioned linear regressor and shallow neural network can be encoded, using the specific encoding, by exploiting the

`EMLib` library. The domains for all variables will be set to $[0, 1]$. At this point, one can consider a planning problem over *eoh* weeks:

$$\beta_t \in [0, 1] \; \forall t \in [0, eoh - 1] \tag{123}$$
$$S_t, I_t, R_t \in [0, 1] \forall t \in [0, eoh] \tag{124}$$

where the objective is to maximize S_{eoh}. The encoding of an instance of the netowrk for each week is:

$$(S_{t+1}, I_{t+1}, R_{t+1}) = \text{NN}(S_t, I_t, R_t, \beta_t) \; \forall t \in [0, eoh - 1] \tag{125}$$
$$\beta_t \in [0, 1] \; \forall t \in [0, eoh - 1] \tag{126}$$
$$S_t, I_t, R_t \in [0, 1] \forall t \in [0, eoh] \tag{127}$$

where $\text{NN}(\cdot)$ represents the network encoding. At this point, a number of NPIs are applied at each week. One can assume each NPI i has a cost c_i and can reduce the current β value by a factor r_i. This part of the problem can be formalized as:

$$x_{it} \in \{0, 1\} \forall i \in [1, n_{NPI}], \forall t \in [0, eoh - 1] \tag{128}$$

where $x_{it} = 1$ if and only if NPI i is applied at week t. The total cost should not exceed a given budget:

$$\sum_{t=0}^{eoh-1} \sum_{i=1}^{n_{NPI}} c_i x_{it} \leq C \tag{129}$$

Moreover, the effect this approach has on β is non-linear and trickier to handle. One can linearize it by introducing multiple variables for β at each week: β_{0t} represents the base β value, β_{it} represents β as affected by the i-th NPI. Thus, $\beta_{n_{NPI},t}$ is the same as the variable connected to the neural network for week t. For each intermediate variable:

$$\beta_{it} \geq r_i \beta_{i-1,t} - 1 + x_{it} \; \forall i \in [1, n_{NPI}], \forall t \in [0, eoh - 1] \tag{130}$$
$$\beta_{it} \geq \beta_{i-1,t} - x_{it} \; \forall i \in [1, n_{NPI}], \forall t \in [0, eoh - 1] \tag{131}$$

In this case, if $x_{it} = 1$, the first constraint is active and the second is trivialized, and vice-versa. Lastly, an analogous set of constraints handles the upper bounds:

$$\beta_{it} \leq r_i \beta_{i-1,t} + 1 + x_{it} \; \forall i \in [1, n_{NPI}], \forall t \in [0, eoh - 1] \tag{132}$$
$$\beta_{it} \leq \beta_{i-1,t} + x_{it} \; \forall i \in [1, n_{NPI}], \forall t \in [0, eoh - 1] \tag{133}$$

By setting up such constraints and by defining some initial values for the variables, one is then capable of using the neural models to obtain values for the different variables, and for each possible week. Having done this, one can compare such results with the ones obtained by running a simulation of the evolution of the SIR system.

7.2 Motivation for Decision-Focused Learning

Real world problems typically rely on estimated parameters, and sometimes one has access only to a bit more information. Supposing to have to predict traffic-dependent travel times, then one can operate as follows:

- Train an estimator for the problem parameters:

$$\text{argmin}_\theta\{L(y, \hat{y})|y = f(\hat{x}, \theta)\} \tag{134}$$

- Solve the optimization problem with the estimated parameters:

$$z^*(y) = \text{argmin}_z\{c(z, y)|z \in F\} \tag{135}$$

  where z is the vector of variables of the optimization problem, c is the cost function which can depend on the parameters y, and F is the feasible space which one can assume to be fixed.

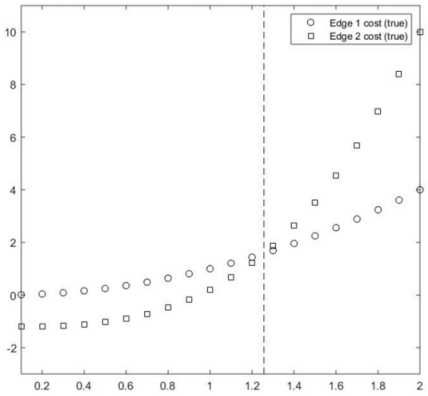
Following this procedure, for any reasonable loss function, better training leads to better predictions:

$$y \xrightarrow{L(y,\hat{y})\to L^*(\hat{y})} \hat{y} \tag{136}$$

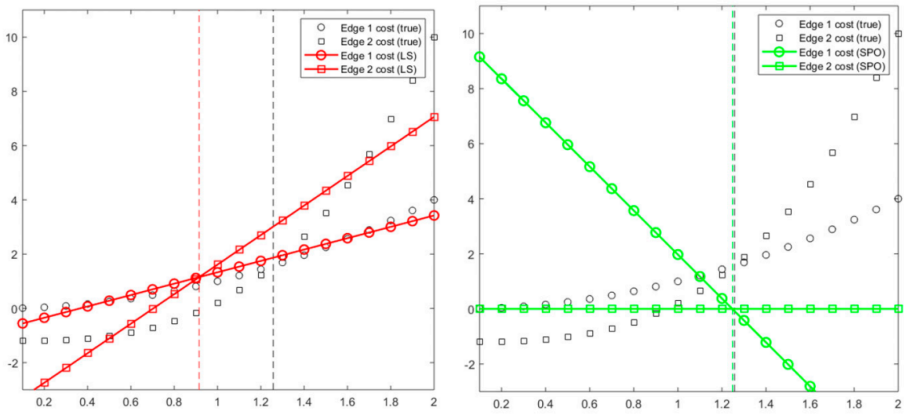
which then lead to the best solution:

$$z^*(y) \xrightarrow{L(y,\hat{y})\to L^*(\hat{y})} z^*(\hat{y}) \tag{137}$$

where $L^*(\hat{y})$ refers to the best possible loss value. However, the second relation holds only asymptotically. In practice, a generic model may not be capable of reaching minimum loss. In such a situations, it is unclear how imperfect predictions impact the cost. Let's suppose to move from location A to B, using one of two routes based on the time of the day (y axis):



In this case, one would like to pick the best route. In the plot above, the dashed line shows the input value that causes the optimal choice to switch. By training a linear regressor one could end up with two different outputs:



where the first estimator is the most accurate, but does not get the switching point right, while the second estimator is less accurate but get the switching point correctly. The aim of **decision focused learning** (**DFL**) is to address such problems. The general idea is to account for the optimization problem during training. In general, DFL formalizes problems in the following way:

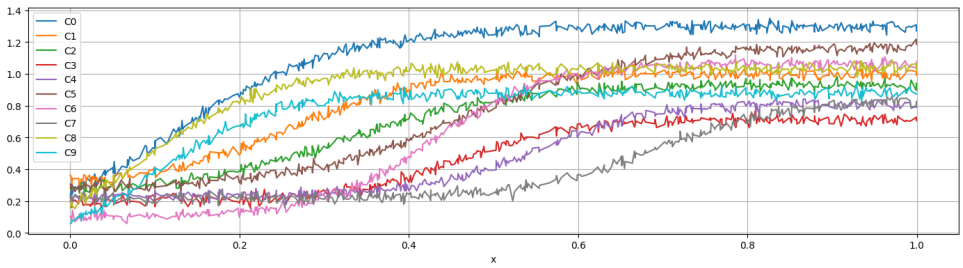
$$\text{argmin}_\theta \left\{ \sum_{i=1}^{m} c(z^*(y_i), \hat{y}_i) | y = f(\hat{x}, \theta) \right\} \tag{138}$$

7.2.1 A Baseline Approach

An optimal purchase problem is considered. Given a set of objects with values v_i and cost y_i, the goal is to buy items for a value of at least v_{min}, while minimizing the purchase cost. In particular, the costs depend on some kind of market state:

$$y = f(x) \tag{139}$$

In this case, it is assumed that the dependency on x is captured by sigmoid curves:



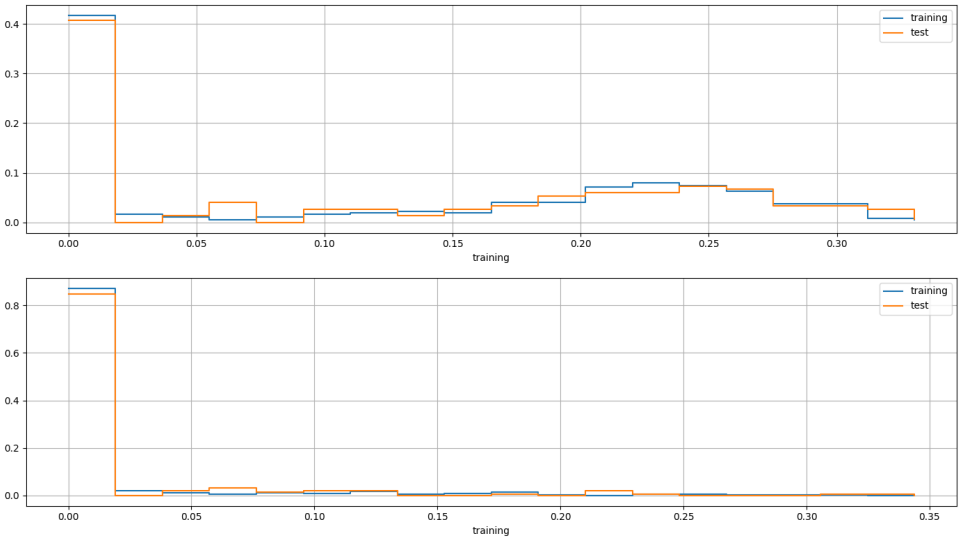
Once such data has been generated, one can split it into a training and test set. At this point, a linear regressor can be trained. Here, the predictions $y = f(x)$ are used to solve the following optimization problem:

$$\text{argmin}_z\{y^T z | v^T z \geq v_{min}, z \in \{0,1\}^n\} \tag{140}$$

which represents an integer linear programming problem. By using a ILP solver, one can solve this problem and supervise the trained linear regressor so as to measure its performance in terms of **regret**. Regret is defined as the cost difference with respect to the true solution. For the i-th example, this is given by:

$$\hat{y}_i^T z(y_i) - \hat{y}_i^T z(\hat{y}_i) \quad \text{with } y_i = f(x_i) \tag{141}$$

where $\hat{y}_i$ is the true cost vector, $z(\hat{y}_i)$ is the true optimal solution and $z(y_i)$ is the optimal solution for the predicted costs. Considering the trained linear regressor, one can plot the regret with respect to the training and test set, and for boht early and late convergence:



7.2.2 Decision-Focused Learning

When considering the original DFL problem:

$$\text{argmin}_\theta \left\{ \sum_{i=1}^{m} c(z^*(y_i), \hat{y}_i) | y = f(\hat{x}, \theta) \right\} \tag{142}$$

one can notice how the argmin used to defined $z^*(y_i)$ is non-differentiable. A small change in the prediction vector y_i may cause a large/discrete change in the optimal solution z_i. A possible solution consists in using a surrogate loss (e.g. a contrastive loss):

$$c(z^*(\hat{y}_i), y_i) - c(z^*(y_i), y_i) \tag{143}$$

This loss contains a constant, $z^*(\hat{y}_i)$, a non-differentiable term, $z^*(y_i)$, and a naturally differentiable term, $c(\cdot, y_i)$. One can pretend that $z^*(y_i)$ is fixed, even if it depends on y_i. It can be proved that,

under some assumptions, this yields a valid subgradient. Considering the market problem for the i-th example:

$$\text{argmin}_z\{y^T z | v^T z \geq v_{min}, z \in \{0,1\}^n\} \tag{144}$$

one can get a valid subgradient by first computing the optimal solution $z_i = z^*(y_i)$, and then by computing $\nabla_y(y^T z_i^*(\hat{y}_i) - y^T z_i) = z_i^*(\hat{y}_i) - z^*(y_i)$ (i.e. the difference between optima with respect to the true and the predicted costs). Thus:

- When evaluating the machine learning model, there is the need to solve the market problem, so as to compute $z^*(y_i)$ for each example in the mini-batch.
- Then, the loss can be computed:

$$L_C(y, \hat{y}) = \sum_{i=1}^{m} y_i^T(z^*(\hat{y}_i) - z^*(y_i)) \tag{145}$$

- Automatic differentiation is used to get the subgradient.

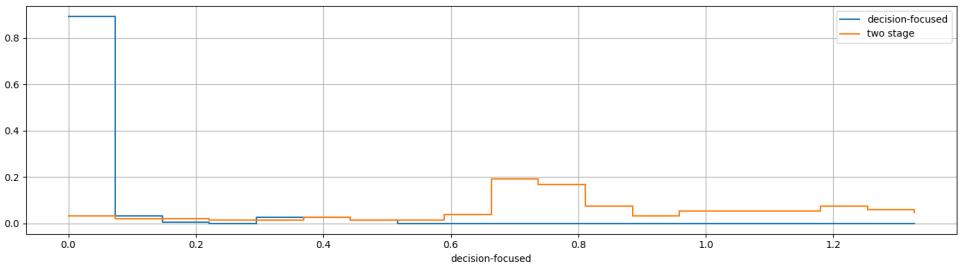
However, all contrastive terms are non-negative by definition and they can be made null by just predicting $y_i = 0$ for all examples. A possible fix consists in using:

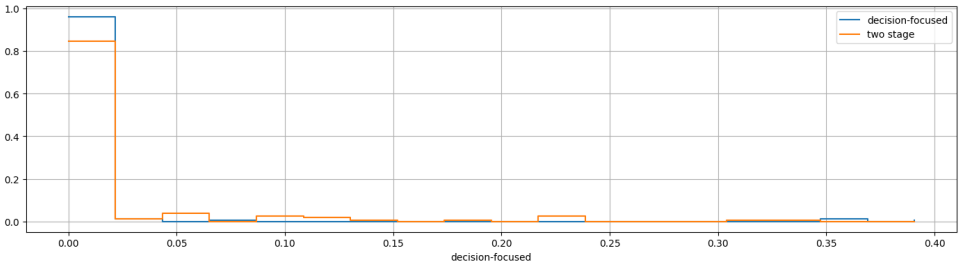
$$L_{CR}(y, \hat{y}) = \sum_{i=1}^{m} \underbrace{y_i^T(z^*(\hat{y}_i) - z^*(y_i))}_{\text{contrastive}} + \underbrace{\hat{y}_i^T(z^*(y_i) - z^*(\hat{y}_i))}_{\text{regret}} \tag{146}$$

where both terms are guaranteed to be non-negative, thus $L_{CR}(y, \hat{y}) \geq 0$. The loss can also be re-written as:

$$L_{CR}(y, \hat{y}) = \sum_{i=1}^{m} (y_i - \hat{y}_i)^T(z^*(\hat{y}_i) - z^*(y_i)) \tag{147}$$

which can be optimized by setting either $z^*(\hat{y}_i) \simeq z^*(y_i)$ or $y_i \simeq \hat{y}_i$. By implementing this in code, and by training the decision-focused model, one is able to beat the previous regret values:




decision-focused
two stage
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
decision-focused