AI in Industry

Francesco Farinola

March 2021

1 Anomaly Detection

Goal: Analyze and anticipate anomalies.

In univariate time series we track only quantity. It's convenient to use `datetime` index which allows us to manipulate dates/hours directly.

NAB module to plot series and `windows` data structure to plot areas where anomalies happen.

1.1 Density Estimation

Density Estimation is an unsupervised learning problem and tipically we do not have access to the true density f^*. It can be solved via a number of techniques:

- simple histograms
- Kernel Density Estimation
- Gaussian Mixture Models
- Normalizing Flows
- Non Volume Preserving transformations

Formalize the problem: if we can estimate the probability of every occurring observation `x` then we can spot anomalies based on their low probabilities. $f(x) \leq \theta$ where f is a Probability Density Function and θ is a threshold.

Given the true density function $f^*(x) : \mathbf{R}^n \rightarrow \mathbf{R}^+$ and a second function $f(x, w)$ with the same input and parameters w and we want to make the two as similar as possible.

1.2 Kernel Density Estimation

Idea: whenever there is a sample it's likely that there are more. Each training sample is the center for a density kernel. Formally a kernel $K(x, h)$ is a valid PDF where x is the input variable and h is the *bandwidth*. E.g. kernels: Gaussian, Tophat, exponential, cosine, linear.

All kernels in KDE are by default zero-centered (mode is 0) and can be relocated via an affine transformation $\rightarrow$K(x-μ, h). Changing the kernel function allows to adjust the properties of the distribution (smoothness) by exploiting our prior knowledge.

Tune bandwidth: minimize the Mean Integrated Squared Error $MISE(h) = \int (f(x, \hat{x}, h) - f^*(x))^2 dx$. In practice, we don't have f^* but we can use an approximation (validation set).

Rule of thumb for univariate case $h = 0.9min(\hat{\sigma}, \frac{IQR}{1.34})m^{-\frac{1}{5}}$ where IQR is the inter-quartile range.

A few tweaks: work with negated log probabilities since it simplifies some operations, are always positive and can be interpreted as alarm signals $\rightarrow$-log f(x,w) $\geq \theta$; Normalize data (compute maximum); estimate bandwidth with rule

of thumb, then fit the univariate KDE estimator; to detect anomalies just apply a filter $signal \geq thr$, use the function of our module `get_pred(signal, thr)`.

Evaluate model: devise a cost model. Define:

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

Call from module `tp,fp,fn,adv = nab.get_metrics(pred,labels,windows)`. We introduce the following costs for choosing our threshold:

- *c_alarm* for loosing time in analyzing false positives = 1 `*len(fp)`
- *c_missed* for missing an anomaly = 10 `*len(fn)`
- *c_late* for a late detection = 5 `*len([a for a in adv if a.totalseconds()<=0])`

Plot the response surface or cost function landscape to find the best threshold (bottom peak) for an ideal optimization problem. Practically we define a validation set, define a range of 'sampled' thresholds and use a function from our module to pick the best one. First approach!

1.3 Combine multiple observations

We can **combine multiple observations** if we notice that data is noise or we are interested in persistent anomalies through: $\frac{1}{n}\sum_{i=1}^{n} logP(x_i) < \theta$.

Combining multiple observations with an Independent and Identically Distributed i.i.d. which is the same as smoothing our signal using a moving average. Notice that the assumptions may not be valid. To implement it we build a filter and then apply the convolution. The convolution needs n observations before it can be first applied and need to update the index for our smoothed signal. Then, measure the effect of changing the threshold and optimize it. We should introduce an extra parameter (window length) and optimize it as well.

Determine the period: with autocorrelation plots. Consider a range of possible lags. For each lag value l make a copy of the series and shift it by l time steps. Compute the Pearson Correlation Coefficient with the original series. Plot the correlation coefficients over the lag values. If there are peaks, positive peaks denote strong correlations and negative ones denote strong negative correlations.

Multivariate-Distribution: our x variable has two components: the first component x_1 is the time of the day and the second x_2 is the value. We can plot it with a 2D histogram.

1.4 Time-Dependent Estimator

Ttime component is completely predictable while value component may be anomalous. $P(x_2|x_1) \leq \theta$. KDE cannot learn natively conditional probabilities, two strategies:

- using the Bayes theorem, we will need a second estimator $P(x_1)$
- if x_1 is discrete, we can learn a KDE estimator for each x_1 value

We need to choose the bandwidth, let $\tilde{x}$ be a validation set of m examples, we use `GridSearchCV` from scikit-learn, define the `params` dictionary with a range to be explored and specify the number of folds. Fit the GridSearchCV, access the best params with `opt.best_params_` and fit the estimator with those. Plot the alarm signal, explore the response surface w.r.t. the threshold and optimize it too with `nab.opt_thr` like before.

Always exploit time-dependencies since time is free information.

2 Sequence Data

2.1 Sliding windows

If we want to take advantage of periods, we need to feed sequences to out estimators. Previous approach, assumed i.i.d. observations but we want to exploit dependencies. Choose a window length w and place the window at the beginning of the series, extract observations and move forward by a certain stride. The result is a table. Pandas provide a sliding window iterator `DataFrame.rolling(window,..)`. Discard the first wlen-1 applications and store each window in a list, then wrap everything in a DataFrame. Doing this by row is kinda slow, so we it by columns with a function in our module `sliding_window_1D(data, wlen)` for univariate data.

2.2 Sequence input in KDE

Use **multivariate KDE**. Treat each sequence as a vector variable and learn estimator as usual. Individual sequences are treated as independent.

Separate the training set, choose bandwidth and train multivariate estimeator. Then generate the alarm signal, look a the surface response for a varying threshold and optimize it. By looking at alarm signal it seems nice but is very unstable (frequent and wide oscillations).

We can improve the situation via time information about our period. First approach consisted in making time an additional variable and adding one dimension. This makes the problem more complex.

Second approach consists in *learning many KDE estimators*, each specialized for a given time (e.g. 00:00, 00:30, 01:00...). It becomes an ensemble. Each KDE estimator works with a smaller amount of data but the individual density estimation problems are easier. We learn 48 specialized estimators for each unique time value and separate a subset of the training data. Then build and learn a KDE estimator by storing everything in a list/dictionary. Doing this with log probabilities makes it a lot faster.

Suggestion: always concatenate lists in a single step with pandas. Then do the usual steps: plot signal, examine response surface, optimize threshold and see cost on whole dataset.

NB: Using raw, absolute, time as feature may lead to overfitting!

2.3 Anomaly Detection via Autoregression

Training with KDE was pretty fast, but predictions were slow. E.g. speed control in an industrial pump or graceful thermal throttling of a multi-core CPU need faster predictions.

Alternative way is to build an **Autoregressor** $f(x)$ (a predictor for the next step in the time series). We can use the predictor error as an alarm signal $|f(x)-y| \geq \theta$. We can use any regression approach (Linear Regression, Decision Trees, Neural Networks).

Linear regressor: is a supervised learning approach whose goal is to fit a linear function: $f(x, w) = w_0 + \sum_{j=1}^{n} w_i x_i$ where w is the vector of n+1 weights, w_0 is used as a constant and is called intercept. If the signal to be predicted has 0-mean, w_0 can be omitted. Training is done by minimizing via the **Least Squares method** $argmin_w ||f(\hat{x}, w) - \hat{y}||_2^2$. Least Square method can be optimally solved in polynomial time even by simple gradient descent. It's a convex, unconstrained, numerical optimization problem. It assumes the predictions is *Normally distributed* with *fixed variance* which is an important limitations.

First, we need to build the target vector. In linear regression, all columns should have the same scale. Then, separate the training set and train our predictor. Finally, we obtain predictions very fast and look at quality with R2 score and plotting them. We just need to compute the errors to obtain out signal.

We can also optimize the threshold and check cost over all the dataset.

2.4 Handling seasonability

Periodic-like behaviour in time series is referred to as seasonability. Linear regression can deal with periodic data as long as the period is shorter than observed window.

Identify periods with autocorrelation plot, but it may contain multiple periods that disturb each other. Trends cause even more noise. A better method consist in using frequence analysis → compute Fast Fourier Transform FFT (decomposition of a discrete signal into a discrete set of sine/cosine components). We can compute it with `numpy.fft` and get the corresponding frequencies with `numpy.fftfreq`. Frequencies are in $cycles/d$ and a different sampling frequency may be specified. For a real-valued signal, the FFT is always symmetric. For a real-values signal, the FFT is always simmetric. We care only about the positive frequencies (the first half). 0 frequency is also not relevant for us.

Select all frequencies with amplitude greater than 100 and invert them to obtain the periods (in time steps). Discard the duplicates and consider the longest periods (most important becuas eit conveys information about its submultiples).

Simple approach consist in **differencing**. It is the heart of the so-called 'integrated' linear approaches for time series. it's the I in ARIMA models. Our predictor becomes the sum of two predictor: the linear regression model and a predictor $g(x, d)$ that outputs the target exactly d steps before. The idea is to let a dedicated predictor handle the period. Once we know d by training $g(x, d)$ we obtain $f(x_i, d) \simeq \hat{y_i} - g(x_i, d)$. We obtain the linear regressor targets by subtraction and we also add the '1' period since repeating the last value has a strong prediction value (persistence predictor).

```
data_d,deltas = nab.apply_differencing(data,periods)
```

Then, choose a sliding window length with autocorrelation plot and apply it. Finally train again the regressor and obtain predictions. Getting the actual predictions requires to reverse the differencing operation: by summing the rightmost part of the residual vectors we stored.

`deapply_differencing(pred,deltas,lags,extra_wlen)`.

As usual, check prediction quality with scatter plot and R2 score, plot signal, optimize threshold over all dataset.

Another method consist in augmenting the input data with a frequency component int he from $w_1 cos\pi wt + w_2 sin\pi wt$ where w corresponds to the frequency we want to capture and w_1 and w_2 are weights to be tuned in Linear Regression. In practice, we add 1 column for $cos\pi wt$ and one for $sin\pi wt$

In differencing, we model the period effect and remove it. In time-indexed models we learn models that take time as input. They are not tied to Linear Regression and KDE. Removing seasonality is related to gradient boosting

2.5 Time Indexed KDE

Start by getting all unique hour-of-the-week values. Define a bandwidth based on part of the data and learnt he ensemble. Run all estimators and concatenate the results. Plot signal. Optimize the threshold and see the cost on whole dataset. We have two drawbacks:

- Each KDE model is trained on fewer data points n/(48*7) and has a relatively high-dimension input. The more the dimensions, the larger the required datasets.
- We are looking at multiple observations. This makes easier to detect anomalies but more observations are needed before the anomaly is considered significant.

If we have bad overfitting problems check the size of each training set. It may be smaller than the window size.

3 Missing values

Missing values in time series are called artefacts. They are common due to malfunctioning sensors, network problems, lost data, sensor maintenance/installation/removal. They prevent application of sliding windows and complicate detecting periods.

3.1 Preparing the ground

Series have sparse index but we need dense one with missing values represented as NaN.We start looking at the sampling frequency by computing the distance between consecutive available index values. If there are more frequencies we need to realign the original index with round method and remove duplicates with `.duplicated()` method (keep the first value). A more complex strategy could be to replace duplicates with averages with groupBy operation.

3.2 Basic approaches

Initially use partially synthetic data. Start with a time series without any missing values and remove values artificially. Then measure the accuracy with RMSE.

Easiest approach consists in *replicating nearby observations*:

- **Forward filling**: propagate forward the last valid observation
- **Backward filling**: propagate backward the next valid observation

These methods have a strong level of local correlation and last observation is a good predictor for the next one. Both pre-implemented in pandas through `fillna` method: it can take 'pad' or 'ffill' for forward filling and 'backfill' or 'bfill' for backward filling.

Forward filling works reasonably well for small gaps, but not larger ones.

More options are available via `interpolate` method which determines how NaN values are filled. We can use linear/time/nearest/polynomial interpolation. On real data, we cannot compare interpolation methods since we have no ground truth.

Usually, better methods just degrade a bit more slowly. MSE comparisons can be very effective but require access to ground truth (by artificially removing values)

3.3 Gaussing Processes

Most ML models cannot be used for filling. Density estimation does not natively provide prediction. Predictions can be extracted from a density estimator by computing the Maximum a Posteriori MAP $argmax_x f(x, \theta)$ but it can be very expensive. Autoregressors makes use of past observations, not future ones. They are designed for extrapolation (predict beyond the boundaries) and not interpolation.

The most viable ML model is **Gaussian Processes**: a stochastic process in which each variable y_x is indexed via a tuple x. The index is continuous and the collection infinite. It's a bunch of correlated, named, Normally Distributed variables.

The multivariate normal distribution had a closed-form density function that is easy to compute and is uniquely defined by its mean μ and its **covariance matrix** $\sum$. If we assume centered data $\mu = 0$ the covariance matrix is enough. We can compute the the *probability density* $P(\hat{y_x}$, the *conditional density* $P(y_x|\hat{y}_x$ of an additional variable y_x, the *conditional mode* $argmax_{y_x} P(y_x|\hat{y}_x$ (a prediction) and we can even *sample the distribution*.

Covariance depends on the indexes. Given two variables y_{x_i} and y_{x_j}, their covariance is given by the kernel function $K(x_i, x_j)$. The covariance matrix is defined entirely by the kernel. We hand-pick a parametrized kernel function and choose the parameter θ to maximize the likelihood of the training data. The training problem is a numerical optimization problem which can be solved to local optimality.

In scikit-learn we choose a target function, build a small training set and choose a kernel (Radial Basis Function RBF or Gaussian). The correlation decreases exponentially with the Euclidean distance. The closer the points, the higher the correlation. RBF parameter is l which is the *scale* that control the rate of reduction. Practically, the values within the boundaries that will be considered. Finally we train the model (fit) and adjust the kernel parameters (observations are then stores) and obtain predictions (predict) where covariance matrix is built. The model returns prediction (estimated mean/mode of the distribution) but also their standard deviation.

We can view a GP as a function f that outputs a probability distribution. This is enables by two components: the kernel (defines how much correlated all the points are) and a set of observations (used to obtain conditional distribution)

Choosing the appropriate kernel:

- Deal with **noise**: `WhiteKernel` has the only parameter σ^2 (noise level). Small noise level prevent overfitting data while too much leads to useless predictions.

  `ConstantKernel` is a constant factor that allows the optimizer to tune the magnitude of the RBF kernel.

- Deal with **period**: `ExpSineSquared` where correlation grows if the distance is close to a multiple of the period p. The scale parameter l control the rate of decrease/increase.

- Deal with **trend**: `DotProduct` where the larger the x values, the larger the correlation. The parameter σ control the base level of correlation. This kernel is not translation-invariant.

Choosing a kernel requires some practice, automating the process is possible but complex (grid search not enough). GPs tend to perform better when interpolating (predicting values between points in the training set) rather than

extrapolating (making predictions far from the training set). Also when there are only a few input dimensions.

3.4 Use case

If in our data we have multiple frequencies (high and low frequency) we may apply a **low-pass filter** to reduce the amplitude of high-frequency components and keeping the low-frequency mostly unaltered.

Exponentially Weighted Moving Average is a discrete non-ideal filter. We can use it in pandas using the `ewm` iterator, plus the `mean` aggregation function. The `com` parameter corresponds to the period τ. After choosing the right com parameter we apply the FFT and see the smoothed signal. If the series comes from human activities it is likely to have daily (288 steps) or weekly (2016) periods. After detecting the period we add time information to our dataframe. Cannot use `datetime` with scikit so we turn time into a number. Then remove, artificially remove values assuming half of the data points are missing and make sure that start and end are not missing. Then choose the kernel, train the model and obtain predictions.

If the model has a tendency to predict the value zero, it's most probable that GP has zero-mean so we simply add a ConstantKernel to make all points correlated and match the mean of the training set.

If our signal is not smooth, we apply a kernel for less smooth processes, `Matern`, the lower the nu parameter, the less smooth the prediction.

Exploit periods by using `ExpSineSquared` kernel and repeat training.

NB: use GP only if we have low number of missing values and small gaps.

MSE as quality metric would be good only with uniform variance. If confidence intervals are available, we can compute the likelihood of the validation set (estimated probability).

With GP there is no need to train again the kernel every time new observations arrive. We can build a new $\sum$ matrix using the new observations and the old kernel. Passing `optimizer=None` will disable optimization at training time, so calling `fit` will just take into account new observations

If confidence intervals are still large even at night hours, there are two main reasons. Either there are fewer samples at night time and no traditional GP kernel can represent point-wise, input-dependent variance (all kernel are about covariance, not variance except WhiteKernel).

We can deal them in a separate model and build an ensemble. Variance can be scaled via multiplication so we will produce a product of two models $g(x, \lambda)f(x, \theta)$. Since we hav ediscrete time and a natural period we could choose a single map (time of week $\rightarrow$ standard deviation. Add hour of the week information to our data and compute stdev via pandas `groupby` operation. The resulting table ha a hierarchical column index.

If there are too many gaps to compute σ, we can fill the gaps via standard deviation data or use a coarser time unit or choose a shorter period. We can try to use hour/two-hour intervals instead of 5 minutes and check that no std values is missing and the counts are large enough (around 30). Once we managed

to have reasonable std values, our table has a very coarse time unit. We can mitigate it by oversampling, i.e. switching to a finer grain time unit through a linear interpolation. Next, apply EWMA by choosing τ = number of time steps in one hour (12).

Learn the GP for our Ensemble. First, transform the original series using standard deviation, augment dataset with hour of week information, associate each data point to predicted standard deviation (join method). Select a sub-sequence of data for learning the kernel, separate training and validation, learn kernel parameters and obtain predictions on training data. Then, obtain predictions for missing values in the transformed series, reuse kernel and add observation to obtain predictions on whole series.

Obtain finally predictions for the original series by injecting back the variance, just multiply also the standard deviations. We can fill the missing values using the predictions using the Maximum A Posteriori but also sample from the distribution using `sample_y` which return a matrix and using `ravel` to have a single dimension. In both cases, we clip values at zero.

4 Anomaly Detection via Neural Models

If we face an HPC problem, we first try a KDE approach and do as usual. It may work well but has problems with high-dimensional data → prediction time grows and more data is needed to obtain reliable results. KDE gives nothing more than an anomaly signal, determining the cause of the anomaly is up to a domain expert.

4.1 Autoencoders

Display multivariate series:

- Showing individual columns
- Obtaining statistics (.describe())
- Standardize, then use heatmap. White = mean, red = below mean, blue = above mean

Information like 'mode' = normal/power saving/performance can be served as ground truth.

An autoencoder is a type of neural network designed to reconstruct its input vector. The input is some tensor x and the output should be the same tensor x. Autoencoders can be broken into two halves:

- Encoder: $encode(x, \theta_e)$ mapping x into a vector of latent variables z.
- Decoder: $decode(z, \theta_d)$ mapping z into reconstructed input tensor.

Autoencoders are trained so as to satisfy: $decode(encode(x, \theta_e), \theta_d) \simeq \hat{x}_i$.

We tipically employ an MSE loss. Two approached to aboid learning a trivial mapping:

- Using informaiton bottleneck, i.e. making sure that z has fewer dimensions than x
- Use a regularizer to enfore sparse encodings

Autoencoders can be used for anomaly detection by using the reconstruction error as an anomaly signal: $||x - decode(encode(x, \theta_e), \theta_d)||_2^2 > \theta$.

This approach compared to KDE have *good support for high-dimensional data*, plus *limited overfitting* and *fast prediction/detection time*. However, error reconstruction can be harder than density estimation. Compared to autoregressors, reconstructing an input is easier than predicting the future, so we tend to get *higher reliability*.

Autoencoders can do more than just anomaly detection! Instead of having a single signal we have many, so we can look at the individual reconstruction errors. They are often concentrated on a few signals and often they are useful clues about the *nature of the anomaly*.

If we focus on power saving and performance mode, the largest errors are mostly related to perfomances, 'ips' Instruction per Second and then 'util' utilization. Density estimation, however, is a bit better at pure anomaly detection. In autoencoders, we assume that the prediction for each x_i is independent and normally distributed, with mean equal tot he predictions $g_j(x, \lambda)$ and fixed standard deviation σ.

4.2 Flow models

- Normalizing flows
- Real Non-Volume Preserving transformations (Real NVP)
- Generative Flow with 1x1 convolutions (Glow)

The main idea is transforming a simple and known probability distribution into a complex one that matches that of the available data. They are trained for maximum likelihood to maximize the estimated probability of the available data.

The challenge is defining a transformation f (i.e. the mapping): it must be invertible, non-linear, allow for an easy computation of the determinant.

Real NVP are a type of neural network which take as input a vector x representing an example and output a vector z of values for the latent variable. Training goal is that z should have a chosen probability distribution, typically a standard Normal distribution for each z_i: $z \sim \mathcal{N}(0, I)$. In other words, z follows a multivariate distribution but the covariance matrix is diagonal, i.e. each component is independent.

A real NVP consists of a stack of affine coupling layers. Each layer treats its input x as split into two components. One component is passed forward and the other is processed via an affine transformation (scaled element-wise * and translated +) $\rightarrow$ the transformation is **non-linear**. Each affine coupling layer is **easy to invert** (element-wise / and -). Both s and t are usually implemented as Multilayer Perceptron. The determinant of each layer is easy to compute by the product of the diagonal of a Jacobian. The most important thing is that the matrix is triangular.

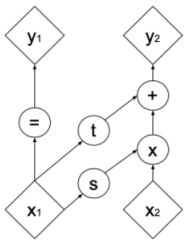

Figure 1: Affine coupling layer

At training time, we **maximize the log likelihood** and care about only log probabilities. If we choose a normal distribution for z, the *log cancels all exponentials in formula.* In general, we want to make sure that all variables are transformed. We can **alternate the roles**, swap x_1 and x_2 at every layer.

Since Real NVPs are invertible, they can be used as **generative models**. They can sample from the distribution they have learned. Just need to sample p_z (on the latent space), easy since the distribution is known and simple. Then go back through the whole architecture using the inverted version of the layers.

Generating data is often their primary purpose, but also for **super resolution**, **procedural content generation** and **data augmentation**.

Implementation: through `tensorflow_probability`, extension for probabilistic computations and easy manipulation of probability distributions.

Translation functions uses a Multi-Layer Perceptron ('relu') with the 'linear' activation function in the last layer.

Scaling function uses another MLP, with bipolar sigmoid (tanh) as output which limits the amount of scaling per affine coupling layer $\rightarrow$ makes it more stable. For the same reason, we use `kernel_regularizer=l2(reg)`.

Use the `RealNVP` model of keras. The function `score_samples` which performs density estimation and triggers the `call` method with `training=True` (transforms data points x into their latent representation z). Then, computes the log density of z using `tensorflow_probability` and then sums the log determinant.

`log_loss` computes the loss function. Done by obtaining the estimated densities via `score_samples`, summing up in log scale (product in original scale) and swapping the sign of the result since we want to maximize the likelihood.

`train_step` called by `fit` method. Before training the model, we need to standardize the dataset. Usually, we choose a large batch size (256) for density estimation approaches.

We can estimate the density of any data point by calling:
`nn.plot_distribution_2D(estimator=model..)`.

We can also generate data, by sampling from p_z and then calling `predict` which triggers the `call` method with `training=False`.

Can also plot the mapping for selected data points which gives an intuition of how the transformation works with `nn.plot_rnvp_transformation(model, figsize=figsize)`.

RNVPs can be used for **anomaly detection**. We build and compile the model with a simpler architecture. Dealing with higher dimensional data has actually some advantage since we have richer input for the s and t functions. After training, we can generate a signal as usual and tune the threshold.

4.3 Component Wear Anomalies

Anomalies due to component wear are investigated with **run-to-failure experiments**, i.e. the machine is left running until one of its components become unserviceable. In these problems, the behaviour becomes more and more distant from normal over time. There is a critical anomaly at the end of the experiment.

If dataset contains high-frequency data, we reduce the frequency via **binning**: apply a sliding window, but so that its consecutive applications do not overlap. Each window is called a bin from which we extract one or more features. The result is a series that contains a smaller number of samples. Usually, we extract **time-domain features** (mean, standard deviation) and **frequency-domain features** (FFT amplitudes) via `groupby` operation.

Choose the bin size so that it is a *submultiple of the segment length* to speed up the operation. Important to standardize sensor inputs and adopt a *categorical encoding* for the operating mode.

If there are strong peaks that prevent the reading of the alarm signal, we clip and smooth the signal. Then apply the usual KDE/Autoencoder/RNVP approach. Usually, create the model, train it, check loss evolution over time, obtain predictions and generate alarm signal. When using an Autoencoder, we can investigate the situation (peaks) using heatmap and check whether errors are concentrated on certain features. By focusing on the peak and returning to the original values we might observe an unusual oscillation. A domain expert may make sense of that.

5 RUL Based Maintenance Policies

Remaining Useful Life is a key concept in predictive maintenance. It refers to the time until a component becomes unusable. We can schedule maintenance operations only when they are needed. It can lead to significant savings.

Remember to bin data if they have high-frequency recordings and check that there are no missing values. Standardize data and split our data. Then plot parameters and sensors for a particular machine and check for operating conditions.

If we plot a particular column of the data we can check the trend, which is possibly correlated to component wear.

5.1 RUL prediction as Regression

We will estimate when RUL becomes too low in order to trigger maintenance: $f(x, \lambda) < theta$ where f is the regressor and θ is the threshold. Split our dataset in training and test. Then standardize all parameters and sensor inputs, then normalize RUL values. Define a regression model, i.e. MLP where `hidden` is the list of sizes of the hidden layers, i.e [64,32], if empty [] we obtain a simple Linear Regression. After training, we obtain predictions and evaluate their quality with `cmapss,plot_pred_scatter`, R2 score and `cmapss.plot_rul`. Our goal is not high accuracy but we just need to stop at the right time $\rightarrow$ cost model.

$cost(f, x_k, \theta) = op\_profit(f(x_k), \theta) + fail\_cost(f(x_k), \theta)$ where:

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

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

where x_k is the series for the machine, s the unit of machine operation are guaranteed and a failure costs C units.

s is determined by the fixed maintenance schedule and C must be determined by discussing with the customer. First collect, all failure times, and then define s and C based on statistics. The safe interval s is equal to the minimum failure time, while maintenance cost C is equal to the largest failure time.

After this, choose a threshold θ and evaluate the model in terms of cost.

Sequences provide information about the trend, it may allow better RUL estimate w.r.t using only the current state. We need to apply sliding windows on a per machine basis (`sliding_windows_by_machine` which outputs a tensor with shape `(nwindows,wlen,ndims)`. The function returns `tr_sw_m` which contains the corresponding machine values and `tr_sw_r` which contains the RUL values both in plain numpy array.

This is ideal for 1D convolutions in keras so we define a 1D convolutions NN with a flatten layer at the end tom get rid of the temporal structure. Then we optimize the threshold, check loss behaviour and see how the CNN fares in terms of cost.

5.2 RUL prediction as Classification

We build a classifier to determine whether a failure will occur in θ steps. We stop as soon as the classifier output a 0: $f_\theta(x, \lambda) = 0$

Build a MLP, but with a sigmoid layer as output. We define the classes: 1 if a failure is more than θ steps away and 0 otherwise.

We train the model and convert the predictions to integer via rounding (since it returns probabilities).

If the dataset is unbalanced, SGD optimization may have convergence issues (will push more on the overrepresented class) so we might use class weights.

In the classification case, we formally solve:

$$\min_\theta \sum_{k \in K} cost(f(x_k\, \lambda^*), 0.5)$$
$$\text{with: } \lambda^* = \text{argmin}_\lambda L(f(x_k, \lambda), \mathbf{1}_{y_k \geq \theta})$$

Unlike, the regression case, the problem cannot be decomposed since θ appears in the loss function. This means we need to optimize, θ and λ at the same time.

We will use **Bayesian (surrogate-based) optimization**. It's a family of methods designed to optimize blackbox functions (with unknown strucure). In practice, the algorithm balances exploration and exploitation. We need to explore regions where we cannot make confident predictions but also focus on regions with promising cost values.

Formally they address problems in the form $min_{x \in B} f(x)$ where B is a box (bounds for each component of x. In our case, the decision variable x would be θ and the function to be optimized is the cost.

We will use `bayesian-optimization` python model (for maximization). It relies on Gaussian Processes as a surrogate model. They are almost guaranteed to intercept every point in the training set and behave reasonably well even with a simple RBF kernel. Also, GPs provide confidence intervals.

Method setup: Define the function to be maximized, rely on warm-start to reduce the training time (stochastic process). We will optimize both θ and the class weights and store the weights of each model. Define the bounding box by considering that we stop from 1 to 15 steps before failure and weights for class 0 from 1 to 10. Then configure the optimizer and run it. Finally, obtain predictions.

5.3 Probabilistic models

- **Same Stop Change**: our regressor is probabilistic since we are implicitly training it with a Normally distributed output having mean $f(\hat{x}_i, \lambda)$ and uniform variance. As a consequence we put the same effort in approximating all examples. When RUL is larger, even with a poorer approx there will still be high chance that our threshold condition is false. But rather than this, we want the same chance of satisfying/not the condition. This is achieved if the standard deviation scales linearly with the RUL.

We established that training a normally distributed predictor with per-sample variance σ_i^2 is equivalent to MSE training with sample weights $w_i = \frac{1}{\sigma_i^2}$. We simply train our MLP with this modification and optimize the threshold. Cost should be on par with MLP but the number of fails may be slightly higher (more risky model).

- **Negative-Binomial Model**: the negative binomial distribution models the probability ro have a number 'failures' before a given number of 'successes', assuming a constant success probability p. In our process, we can view a 'sucsess as the end of the run, a 'failure' as an operating step. So,m if we assume a constant p for all future steps the RUL follows a negative binomial distribution $y \sim NB(1, p)$. We can use a neural model to estimate p based on the observed data: $y \sim NB(1, p(\hat{x}_i, \lambda))$ an hybrid neural-probabilistic model.

  Build the probabilistic model using `tensorflow-probability`, in particular build a custom loss function with `NegativeBinomial` class which swaps the roles of success and failure, support logit input in case sigmoid is applied and allows easy computation of log probabilities.

  After training, to obtain predictions, we call `predict` for the estimated probabilities, build NB distribution objects, and then we can obtain the means, quantiles and variances. We can perform threshold optimization using quantiles, i.e. use the 1st quantile to stop when the estimated probability drops below 25%.

- **Normal Distribution Model:** the distribution variance is tied to its mean. We may want to let the model free to adjust its confidence (variance) independently on the prediction (mean). Define a function to build the architecture. Rely on `DistributionLambda` which wraps a probability into a layer. We pass means and log standard deviations in a single tensor. Our loss function will be the negative log likelihood. At training time, keras repeatedly calls the model. For DistributionLambda, calling returns a distribution object. keras will aggregate by default via a sum, taking into account sample weights.

6 Health Domain Problems

6.1 Arrival Prediction

When we need to count occurrencies over time, it's worth checking Poisson distribution, which models the number of occurrencies of a certain event in a given interval, assuming events are independent and they occur at constant rate. The constant rate is true for the conditional probability (given by the estimator).

The **Poisson distribution** is defined by a single parameter λ, the rate of occurrence of the events. The distribution has a discrete support and the Probability Mass Function: $p(k, \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$. Both the mean and the standard deviation have the same value.

Rate depends on the hour of the day. To take this into account we use lookup table which contain average arrival values for each hour of the day and associate rate with `join` expression.

6.2 Resource Scheduling

Since ER has limited resources, we choose when to perform the activities to improve center efficiency.

We need to convert the flows into a more practical data structure (depends on the case).

The current state-of-the-art for scheduling is **Constraint Programming**. More precisely a hybrid of CP and SMT called **Lazy Clause Generation**. It relies on a SAT engine for DPLL search and conflict extraction. Some constraints are encoded directly into SAT formulas, others have an attached propagator.

CP solvers always work with finite domain variables. Before building any variable we need to determine a horizon for our schedule. The horizon must be larger than any possible schedule. A frequent choice is summing all activity durations.

We gonna use the Google ortools SP/SAT solver. Best performing C solver with permissive license and a python interface.

First priority is choosing the main decision variables: introduce start and end. For each activity, extract the duration and create interval variables. Then create the constraints (simple inequalities in some cases). We can also add precedence constraints. Use the schedule makespan as a cost function, simply the end time of the last task in the schedule. Constrain the makespan variables.

Capacity restrictions over time can be handled via two constraints. The *no-overlap constraint* enforces unary capacity (constraint cannot overlap in time). The *cumulative constraint* enforces a non-unary capacity (task is assumed to have a finite demand).

Many practical combinatorial optimization problems are NP-hard but maybe we can solve them in our size.

6.3 Waiting time

So far, we considered makespan as our cost metric (sum of the start times of the initial task). I.e. we want to track different waiting times depending on the color-code of the patients. We can build waiting time variables, add new constraints and rebuild the problem.

Goal Programming is a strategy for handling lexicographic costs. Consider multiple cost functions $f_0(x), f_1(x)$.. and let f_i be stricly more important than f_j. GP refers to this simple algorithm: Optimize $f_0(x)$ to obtain its optimal value z_0^*, post a new constraint in the model in the form $f_0(x) \leq z_0^*$, move to the next cost function and repeat.

We need to handle upper bounds on the waiting times and code the approach in a function. A drawback is that GP needs to restart from scratch at each iteration, but this can be mitigated by warm starting (via hints)

Large Neighborhood Search: exact optimization approaches enable convenient modeling and are very effective on small problems. Heuristic approaches (local search) can deal with large scale problems but have trouble escaping the local minima. Those two can be combined to obtain LNS. We operate by trying to improve an incumbent solution by searching in a local neighborhood. Each neighborhood is explored via a complete search on a restricted problem.

It can efficiently handle large-scale problems. Can escape reasonably well local minima. Choosing which variables to relax at each iteration can be done using domain knowledge but also with random selection.

Partial Order Schedule: is an augmented task graph. The graph contains all the original precedence relations plus additional precedences, introduced to prevent resource conflicts.

A POS can be obtained by solving a minimum flow problem. We implement it using a collection of outgoing and ingoing arcs (forward and backward star representation).

POS corresponds to a set of feasible schedules. Tasks can be freely shifted, but no resource constraint can be violated. There are no longer branching choices to be made. Choosing one of the possible schedules becomes a poly-time problem.

For using POS in LNS, we need to relax it. Typically, we choose a set of tasks to relax, connect all task predecessors tot heir successors and remove all the additional POS arcs linked to the relaxed tasks $\rightarrow$ the more POS constraints are added, the easier the problem is.

LNS is flexible,, robust, scalable and not difficult to implement. Goal Programming is not only about lexicographic costs but about simplifying a problem via constraints. Sometimes, including only some of the variables (red-coded patients) and simplifying problem formulation (fixed number of vehicles) may improve quality results.

6.4 Kidney Exchange

Kidney Exchange Problem KEP consists in assigning donor to recipients so as to form closed chains and to maximize the number of transplants. It is an *online problem* and consists in *selecting cycles in a graph.*

Only cycles up to a maximum length are considered, the weight of a cycle is given by its number of nodes/arcs. The objective is to maximize the weight and cycles are mutually exclusive if they share at least one node.

To do so, we try to create a precomputed set of cycles (**cycle formulation**). We can enumerate cycles by using simple Depth First Search with limited depth and store them in tuples. After this, we can build the model by using CBC solver: build variables with `IntVar`, cosntraints with `Add`, set the objective with `Maximize` or `Minimize` and set time limit with `SetTimeLimit`.

The main drawback is the limited scalability, number of cycles grows with graph size. High-degree polynomial, the exponential factor is the max length.

Column Generation: is a technique for solving problems with many variables. The idea is to dynamically generate only the variables that are needed. We have a linear program in the form: $min\{cx|Ax \geq b, x \geq 0\}$.

This can be solved in polynomial time with Interior Point algorithm or in pseudo-polynomial with Simplex algorithm. Both provide optima values x^* for the primal variables but also a vector of optimal dual multipliers λ.

The **dual multipliers** are associated to the problem constraints. If a constant i is violated we receive a penalty ($\lambda_i \geq 0$), if it is satisfied with a slack we receive a reward ($\lambda_i = 0$). The alternative formulation is called a **Lagrangian Relaxation**.

We start by solving an LP with a subset of variables. Its solution will be feasible, but not necessarily optimal. We consider the remaining variables and compute their reduced costs (**pricing problem**). If there are no variables with negative reduced cost, we know our LP solution is optimal. Otherwise, we add variables and repeat the process.

In some cases, the pricing problem should consider all cycles. Enumerating all of them could be expensive so we focus on the most negative reduced costs which is enough to check whether there is any negative r.c. Searching for the most negative r.c. is equivalent to **searching for minimum weight cycles**.

Constrained Minimum Cycle Weight: We will base our pricing algorithm on a *Time Unfolded version of the graph* (contains one copy of each node per time unit). The graph is layered and no arcs between nodes associated to the same time unit, Since graph is acyclic, shortest paths can be found using **Dijkstra's algorithm.** The process returns one cycle per root node and per valid weight. All shortest cycles have non-negative r.c. which is expected since the dual multiplier refer to an optimal solution. Focusing on minimum weight cycles gives a massive speed improvement.

Application: In the Column Generation approach, the initial pool of variables corresponds to all shortest cycles and start the main loop for cycle formulation which returns an optimized cycle pool. At each iteration of cycle formulation we solve the LP relaxation, find all shortest cycles and detect the cycles with negative reduced costs.

If there are not cycles with negative r.c. we have converged, otherwise we add all arcs with negative r.c. to the problem.

After solving the CG formulation, we have an optimal solution of the LP relaxation (not the actual solution) which may violate constraints. A simple strategy is to keep the set of variables and solve the original problem. This is guaranteed optimal only if LP-IP gap is zero, otherwise we should start branching (Branch & Price).

Considerations: Column Generation provides massive scalability improvements and it can lead to simpler problem formulations. The key is a clean master problem which should have a high-quality LP relaxation. The trick is packing most of the complexity in the definition of the problem variables.

7 Constrained Machine Learning Applications

7.1 Domain Knowledge Injection via SBR

If there are few run-to-failure experiments we can exploit data about normal operation with two base approaches define RUL:

- **Anomaly detection approach**: AE or density estimator to generate an anomaly signal, optimize threshold based on few run-to-failure experiments. But signals for different machines may grow at different rates.
- **Resort autoencoders and semi-supervided learning**: train autoencoder on unsupervised data, remove decoding layer and replace with classification one which we train on tun-to-failure experiments. Not so reliable.

Another way, rely on domain knowledge from experts to obtain information from unsupervised data and inject such information in the model by means of constraints.

We know that RUL decreases at a fixed rate so we identify a soft (not strict) and relational (involves pairs of examples, *tipically contiguous*) constraint. We use the constraint to derive a semantic regularizer (**Semantic Based Regularization**).

The regularizer represents a constraint that we think should generally hold. It is meant to assist the model by ensuring better generalization or by speeding up training or allowing to take advantage of unsupervised data.

We start by removing the end of the unsupervised sequences to simulate the fact that machines are still operating, then assign an invalid value (-1) to the RUL of unsupervised data. Build a single dataset containing both supervised and unsupervised data. SBR requires top have sorted samples from the same machine so we need to build a custom `DataGenerator` on modify functions to build batches, train step, initialize, etc.

Finally we train the model with only supervised, only unsupervised and both data, obtain for all predictions and plot them. Then, optimize the threshold and evaluate the cost.

Consideration: Regularized approaches for knowledge injection are very versatile. They can apply to any constraint for which we can obtain a good differentiable regularizer. Choosing the correct regularizer weight can be complicated since the is still improving generalization, we could use a validation set. However, if supervised data is scarce this may not be practical. Domain knowledge is ubiquitous. It is sometimes contrasted with deep learning (which can infer knowledge from data).

7.2 Shape Constraints in Lattice Models

Lattice Models are a form of piecewise lienar interpolated model. They are defined over a grid on their input variables and their parameters are the output

values at each grid points. The output values for input vectors not corresponding to a point of the grid is the linear interpolation of neighboring grid points (`tensorflow-lattice`).

Lattice models can represent non-linear multivariate functions and can be trained by (e.g.) gradient descent. 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". The input variables are expected to have bounded domains (namely $[0, n_i - 1]$) but this leads to scalability issues. Lattice parameters are **interpretable** which is a lot important in industrial applications (you have to explain how the model works).

To **evaluate predictions** in these cases, we use the **ROC curve** (Receiver Operating Characteristic) in which we consider the false positive rate (x) and the true positive rate (y). The larger the Area Under Curve AUC, the better. $[0, 1]$.

In `tensorflow-lattice`, scalability issues are mitigated via:

- Ensembles of small lattices
- Applying a calibration step to each input variables

Calibration for *numeric attributes* consists in applying a piecewise linear transformation to each input, a 1D lattice. It introduces complexity, without increasing lattice size. `PWLCalibration`.

Calibration for *categorical inputs* consists in applying a map, encoding inputs as integers. Calibration enables the use of fewer knots in th elattice. `CategoricalCalibration`.

Lattice models are well suited to deal with **shape constraints**: restrictions on the input-output function to be learned such as monotonicity (reducing price will raise sale volume) and convexity/concavity (too low or too high temperature lead to worse bakery products).

7.3 Fairness issues in Machine Learning

In Linear Regression, we can plot the weights by decreasing value. If there are many large-ish weights we can add an L1 regularizer to obtain **Lasso Regression**. The regularizer penalizes large and small weights with a fixed rate. This helps also preventing overfitting. Lasso weights are sparse meaning that only few attributes will have a significant impact.

About discrimination the most we can do is: given a set of categorical protected attributes (indexes) J_p, the regression of the **Disparate Impact Discrimination Index ($DIDI_r$)** is given by:

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

where we penalize the average group predictions for deviating from the global average.

Discrimination indexes can be used to state fairness constraints: $DIDI_r(y) \leq \theta$. If the chosen index is **differentiable**, then we try to *inject the constraint* via a semantic regularizer.

For example, we may use a loss function in the form:

$$L(y, \hat{y}) + \lambda \max(0, \mathrm{DIDI}_r(y) - \theta)$$

For non-differentiable indexes we can use a differentiable approximation or use other approaches that do not require differentiability. When using this approach, choosing the regularizer weight could be tricky since we would lose some accuracy while satisfying a fairness constraint.

Another simpler approach is **Lagrangian duality**. We can generalize the above formula as:

$$\mathcal{L}(y, \lambda) = L(y, \hat{y}) + \lambda \max(0, g(y))$$

and formulate the Lagrangian relaxation (relaxation because the optimum cannot be larger than the constrained optimum):

$$\min\{L(y, \hat{y}) \mid g(y) \leq 0\}$$

For any λ solving $\min_y \mathcal{L}(y, \lambda)$ gives a lower bound on the optimum of the constrained problem. And among many lower bounds we take the largest one: $\max_\lambda (\min_y \mathcal{L}(y, \lambda))$. Intuitively we choose y top minimize the Lagrangian and λ to maximize it (**Lagrangian dual**).

It can be approximately implemented without an outer search loop and make a gradient step over y and a second one over λ. This is not limited to fairness constraints and it works for any differentiable constraint function. Even when there are many constraints with disting weights.

When implementing, we no longer pass a fixed `alpha` weight but we use a trainable variable. Define two distinct gradient steps in the `train_step` function.