IoTwins
BIG DATA OPTIMIZE INDUSTRY AND SERVICES

Grant Agreement N°857191

Distributed Digital Twins for industrial SMEs: a big-data platform

DELIVERABLE 3.3 – FINAL VERSION OF NOMINAL MODELS, FAULT LIBRARY, AND AI SERVICES FOR DIGITAL TWINS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement № 857191

Document Identification

Project	IoTwins
Project Full Title	**Distributed Digital Twins for industrial SMEs: a big-data platform**
Project Number	857191
Starting Date	September 1st, 2019
Duration	3 years
H2020 Programme	H2020-EU.2.1.1. - INDUSTRIAL LEADERSHIP - Leadership in enabling and industrial technologies - Information and Communication Technologies (ICT)
Topic	ICT-11-2018-2019 - HPC and Big Data enabled Large-scale Test-beds and Applications
Call for proposal	H2020-ICT-2018-3
Type of Action	IA-Innovation Action
Website	iotwins.eu
Work Package	WP3 - AI services for distributed digital twins
WP Leader	UNIBO
Responsible Partner(s)	UNIBO
Contributing Partner(s)	THALES, ESI
Author(s)	Andrea Borghesi (UNIBO), Michela Milano (UNIBO)
Contributor(s)	Vincent Thouvenot (THALES), Artem Kolesnikov (ESI), Michel Quentin (ESI), Morgan Cameron (ESI)
Reviewer(s)	Tom Escoffier (THALES), Fernando Cucchietti (BSC)
File Name	D 3.3 – FINAL VERSION OF NOMINAL MODELS, FAULT LIBRARY, AND AI SERVICES FOR DIGITAL TWINS
Contractual delivery date	M30 – 28 February 2022
Actual delivery date	M34 – 10 June 2022
Version	7
Status	Final
Type	DEM: Demonstrator, pilot, prototype
Dissemination level	PU: Public
Contact details of the coordinator	Francesco Millo, francesco.millo@bonfiglioli.com

Document log

Version	Date	Description of change
v0	20/01/2022	First draft
v1	24/01/2022	Addition of Contextual RNN and shapkit parts
v2	01/02/2022	Addition of a section about ML for predictive maintenance
v3	14/03/2022	First internal review completed
v4	18/03/2022	Added section on simulation models
v5	29/04/2022	Second internal review completed
v6	23/05/2022	Document sent to the consortium for approval
v7	10/06/2022	Final version

Table of Contents

1. Introduction

In this document we describe the final list of Artificial Intelligence –based services developed and deployed on the IoTwins platform. The final list comprises entirely new services with respect to those presented already in D3.2 and revised versions of the previous ones, which were refined and improved following the suggestions from the industrial partners (SMEs and large data centres) within the IoTwins project. The official repository hosting the code of the services, together with the documentation, is remained in the original address: https://gitlab.hpc.cineca.it/iotwins/ai-services. This repository is hosted in a CINECA-controlled area (CINECA is one of the partners of the consortium), so access must be granted according to CINECA policy.

Since the first version of the AI-based services described in D3.2 the list of developed and deployed services has been expanded, mostly according to the needs and requirements stated by test-bed partners (see D3.1 and the second round of interviews in T3.1 and T3.2). The new services have been developed following the modality described in D3.2, by encapsulating the core functionality within Docker-based wrapper. This allows us to easily deploy and manage the services on the IoTwins platform.

2. Service development methodology

Keeping in line with the first version of the services (see D3.2), the new services were developed and offered as a series of self-contained Docker containers. As a very quick reminder the Docker framework aims at the creation of compact services using OS-level virtualization and the delivery of said software (SW) services as self-contained packages, endowed with all the required companion libraries and SW tools. In this way the AI-services can be used a standalone module, while at the same time it can be deployed on the IoTwins platform produced as outcome of WP2.

Additionally, as in D3.2 we proceed by presenting in the current document the list of services with their detailed description while the actual *demonstration* of these services can be fully tested only by running them on the IoTwins platform, which relies on the PaaS orchestrator provided by INFN (one of the partners in the consortium). The actual demonstrator can be reached at the following link https://iotwins-paas.cloud.cnaf.infn.it/ . Currently, the demonstrator is hosted at the facilities of INFN cloud and it is not accessible without registering to the INFN cloud service – only users with correct credentials can access the AI-based services and use them as standalone services; a subset of the services has been deployed in specific testbeds and is thus accessible accordingly to the relevant partner's policy (e.g., the anomaly detection service in TB06 or the time-series classification service in TB12). Differently from the demonstrator created in the first half of the project and described in D3.2 (which relied on INFN cloud infrastructure not devoted to the IoTwins project), the current set of services can now be deployed on the official IoTwins platform. An example of the usage of the services while deployed on the IoTwins platform can be found on this YouTube channel (the services shown on the video are only an exemplary selection of those available in the full catalogue): https://www.youtube.com/channel/UC_V0G8fF2_kqn-bGqEBKeMw.

In this document we will focus on the general description of the services without considering whether they run on the edge or on the cloud. The deliverable D3.5 will instead provide the detail concerning the deployment of AI-based services on the cloud. The general version of the services described this deliverable (and expected as the main outcome of the WP3) expect data to be provided in batch-mode, unless specified differently; this is done to provide a simple and general way to test the standalone service. In specific testbeds with their specific deployment environments the general services might require specific tailoring actions.

Batch-mode data can be provided in the form of Comma Separated Values (CSV) files to further simplify the user's interaction; however, the services have been developed to be easily extended to deal with different data sources, if they are compatible with the internal data representation. For instance, while currently both the Remaining Useful Life prediction service and the time-series classification service expect a CSV file as input data, internally they rely on a data format called *Pandas dataframe*, found in a commonly used open-source Python library, which are obtained by applying a data conversion; many other data sources (e.g., InfluxDB or Cassandra databases) can be converted into Pandas dataframes, according to the specific testbed needs.

The documentation is available in the online repository itself, as it is closely connected to the code implementation and the services interface. For instance, this is the documentation relative to the semi-supervised anomaly detection service:

anomalyDetect_semisupervised module

anomalyDetect_semisupervised.py

The module containing services specific for semi-supervised anomaly detection with Deep Learning techniques

Copyright 2020 - The IoTwins Project Consortium, Alma Mater Studiorum Università di Bologna. All rights reserved.

`anomalyDetect_semisupervised.`**`semisup_autoencoder`**(*df, user_id='default', task_id='0.0', hparams={'actv': 'relu', 'batch_size': 64, 'drop_enabled': False, 'drop_factor': 0.1, 'epochs': 20, 'l1_reg': 1e-05, 'loss': 'mae', 'lr': 0.0001, 'nl_0': 3, 'nl_u': 4, 'nnl_0': 10, 'nnl_u': 2, 'optimizer': 'adam', 'overcomplete': True, 'shuffle': True}, n_percentile=- 1*)

anomalyDetect_semisupervised.semisup_autoencoder::Creates an semi-supervised autoencoder to detect anomalies; the idea is to train the autoencoder on the normal data alone, then use the reconstruction error and a threshold to classify unseen examples (binary classification: allowed classes are only normal and anomaly). This models does not work on time-series but has a rather "combinatorial" fashion: after having being trained it makes it prediction (thus classifying the data point) based on the single test examples fed to it (disregarding precious data points). For details, see Borghesi et al., 2019, "A semisupervised autoencoder-based approach for anomaly detection in high performance computing systems", Engineering Applications of Artificial Intelligence.

:param df : *dataframe*
data to be used for training and testing. One column has to termed "label" and it has to contains the classes of the example - 0 means normal data point, any other integer number corresponds to an anomalous data point

:params user_id : *str*
user identifier

:params task_id : *str*
task identifier

Params hparams: Python dictionary hyperparameters to be used to build the autoencoder The hyperparameters must be specified as a Python dictionary

:params n_percentile : *int*
percentile to be used to compute the detection threshold; if no percentile is provided all values in the range [85, 99] will be explored and the one providing the best accuracy results will be selected

:return : *Keras model*
the trained autoencoder

:return : *string*
the name used to save the model – only alphanumerical characters are allowed, no file extension

:return : *Sklearn scaler object*
the scaler used to normalize the data

:return : *Python dictionary*
the summary of the detailed statistics computed on the autoencoder, not necessarily related to the classification task (which depends on the threshold as well), e.g. the reconstruction error for the every data point, and the accuracy results

:return anomaly_threshold : *float*
threshold to be used for distinguish between normal and anomalous points

3. Revised list of developed services

In this section we list and provide precise details about the final set of Artificial Intelligence-based services provided in the IoTwins platform. Each subsection will be devoted to a particular service. We are not covering the services already described in D3.2 again, that is the first version of the AI-based services. Those services were mostly kept unchanged in terms of functionality and, especially, usage mode and APIs; the only changes involve the documentation (for which we refer to the online repository) and internal implementation details (e.g., code optimization and refactoring) which we will not cover in this document (again referring to the online repository).

3.1 Remaining Useful Life Estimation

The remaining useful life (RUL) of an asset or system is defined as the length from the current time to the end of the useful life. Being able to predict this metric with sufficient reliability allows you to enter the domain of predictive maintenance. If the end of useful life of a component is predicted to be before the scheduled maintenance, an unexpected failure can be handled; if the scheduled maintenance is near, but the metric tells us the component is still healthy enough, more value can be extracted out of it before substitution. In recent years, Deep Learning data-driven approaches for RUL predictions have been introduced in literature and applied in practice in different contexts. Recurrent Neural Networks (RNN) and Convolutional Neural Networks (CNN), with their variant, the Temporal CNN (TCNN), have been shown to effectively handle multi-variate time-series to predict the RUL of industrial component. Hence, we decided to develop such models and offer them as a RUL estimation service. In the service developed we do not cover exclusively the model creation and training (two necessary phases of any DL-based approach), but we also take care of the data pre-processing, that is manipulating the data available in the form of time-series to make it suited to DL techniques.

The developed service is a *supervised* one: it relies on the presence of a label associated with the data (time-series) collected from the target system. This label must specify the status of the system at any time-step; this is required to train the DL models. Moreover, to effectively train these models the training data (the data used to teach the DL models) must contain some *run-to-failure events,* that is periods of time where a failure/a problem/an anomaly happened; there must be at least one run-to-failure, but a larger amount is welcomed and could greatly improve the accuracy of the service. It is pretty hard to assess the right number of run-to-failure sufficient to improve the RUL estimation, as it strongly depends on the target system's dynamics; broadly speaking, less than 10 run-to-failures should be sufficient to provide better accuracies, while larger numbers would allow for the training of even better DL models (and would especially be beneficial for validating and fine-tuning the model before its deployment). In the absence of critical events (e.g., the target system has a lifespan of several years and no failure has yet happened) this DL-based supervised service cannot be applied; in this case, simulated data could be used if possible. The service has been developed to be usable in different contexts, but it has also been tailored to a specific testbed, more precisely TB04.

3.1.1 Data pre-processing

First of all, some features are discarded: those which are clearly monotonic and those that present just a single value all the time; time-independent features are not helpful in the RUL estimation, and thus can be discarded. The first category of features is excluded, as they do not contain useful information and strongly correlate with time, offering the models an easy way to overfit; the ones in the latter category are plainly

useless. Then, the features are downcasted to float32 with huge saving both in terms of memory and time, while arguably having a minimal effect on prediction performance. All the features are normalized before being fed to the DL models, using the 0-mean and unit variance approach.

Optionally, users of the service can choose to perform *downsampling* on the data as well. Downsampling requires an aggregation function of some sort. This has been parametrized, but in practice, the max function was always used. The max function has the property of not creating new values: the output is a value that is also contained in the input set. This is fundamental for all those logical features that have a limited range of values (due to their abstract nature, e.g., status of a subsystem), where each precise value has a rich meaning. This solution is far from optimal, but it has the advantage of not relying on any description of underlying features; ad-hoc function can be built by the users with more domain knowledge. In the case that there are gaps in the data set filling becomes necessary. A simple approach has been used: filling with the last available value. Like in the choice of max function for aggregation in downsampling, this one also was dictated by the need of not introducing new values for logical features.

3.1.2 RUL Modeling

After the necessary pre-processing operations, the data is ready to be modeled for actual RUL estimation: this means a RUL label must be built. The classical way of obtaining a RUL label from time series is to segment it in runs-to-failure, i.e., periods of time where a component starts as perfectly healthy and goes towards the end of its useful life (where RUL is 0). In practice, there are two things to do: 1) segment the data into runs-to-failure and 2) define a RUL label for each run.

The service assumes that the data is annotated with a label describing the status of the system; this label indicates the presence of an anomaly in the data. We use this concept to compute the run-to-failure in the following fashion, performing run segmentation:

- (1) The first run starts with the first-time sample of the data.
- (2) Data is scanned along the time axis
  - (2A) if the end of data is reached, the procedure ends
  - (2B) if a (relevant) anomaly is found, go to point (3)
- (3) the found anomaly is the last time sample of the run.
- (4) drop the data after the found anomaly, for a time range of duration specified as a parameter X settable by the service user (if nothing is specified by the user, the default value is 10 days).
- (5) if end of data is not reached, a new run starts right after this X period.
- (6) go back to point (2).

The deletion of a time range of duration X after the end of a run is for multiple reasons: i) we just want to consider only the first anomaly in the “group”, ii) we want to ignore/drop the data for a period of time that includes the whole group of anomalies, as we want the new run to start in a new “healthy component” scenario.

Once we have separate runs, the RUL label must be created for each one. Different possibilities for run labeling are available, ranging from “linear” with a linear degradation from start to finish, to “piecewise”, where we apply a linear degradation for the last N hours (N is a user-specified parameters) and the label is

fixed to a specific level in the previous data in the same run; see an example of piecewise and linear labeling in Fig. 3.1.

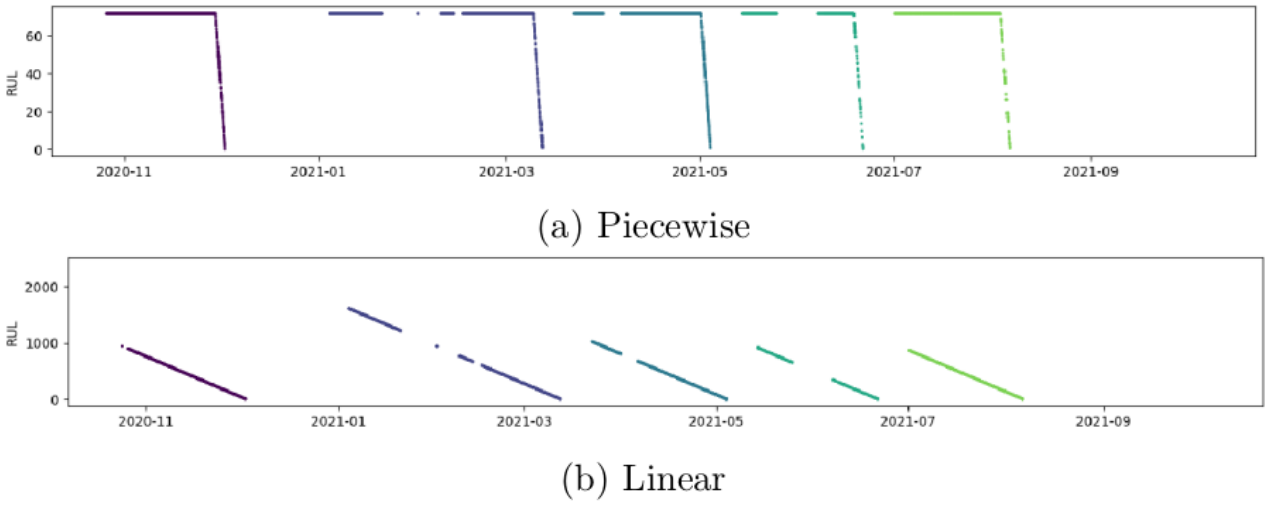


Figure 3.1: Piecewise and Linear labeling

3.1.3 Time sequences

Given N time steps and a sequence length L (L ≤ N), the total number of obtainable (contiguous) sequences is N − L + 1. Working with a sampling frequency of one sample each 0.5 seconds would result in millions of sequences, with each step in each one having multiple features: overall too expensive. The first possible modulation comes from down-sampling, but in general we want to be able to work with data at the highest frequency possible, to have as much information as possible to draw from. More help comes from using a stride factor when choosing sequences: taking only one sequence each "stride factor" possible sequences, considering them sequentially. The resulting number of sequences is the total divided by this factor: this can lead to huge savings in time and computational resources. A downside of using stride is that we lose some information, a given time step is not found any more at all possible positions in the resulting sequences. Depending on the data, the used sampling frequency, the sequence length (steps) and the stride factor itself, this effect can be strong.

Even employing all these mechanisms to reduce the number of sequences, generating them all at once is in most cases infeasible from an implementation point of view. A "generator" scheme[1] has been used, where data is loaded in RAM memory only when necessary, during the training of the model and during inference.

3.1.4 DL Models

We implemented two different types of neural network for handling the time series and predicting the RUL (after the data have been pre-processed): Recurrent Neural Networks (RNN) and Convolutional Neural Networks (CNN), both described in the following.

3.1.4.1 RNN

Recurrent Neural Networks (RNNs)[2] are a particular kind of neural network, suited to handle sequential data. The main characteristic is temporal dynamic processing of the input data, given by the presence of self-

[1] https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/TimeseriesGenerator

[2] The unreasonable effectiveness of recurrent neural networks. https://karpathy.github.io/2015/05/21/rnn-effectiveness

reference (loops): the neuron has a state which depends on the state generated in the previous time step. RNNs can be fed with sequences of variable length and can be applied in different ways, according to the relationship between the input and output. RNNs are notably hard to train in case of sequences of non-trivial length: this is a limit, as we want to capture long-distance dependencies in the data. The underlying problems, common to all Deep Learning, are due to the fact that back propagation calculations (its chained multiplications), can lead to increasingly small or big adjustments in weights of the early layers (Vanishing Gradient and Exploding Gradient). Among different methods employed to fix this issue, a sure way for getting an improvement is to use more advanced RNN units, invented specifically to counter these problems: LSTMs.

Long Short-Term Memory (LSTM) is an RNN unit built specifically to counter the problems about gradient stability during training, crucial when long-distance information dependencies need to be considered in the sequential data. This is done by handling in a more advanced way the state of the RNN; the key component of LSTM networks is LSTM units, as shown in Figure 3.2. For the RUL estimation service, we decided to implement an LSTM model.

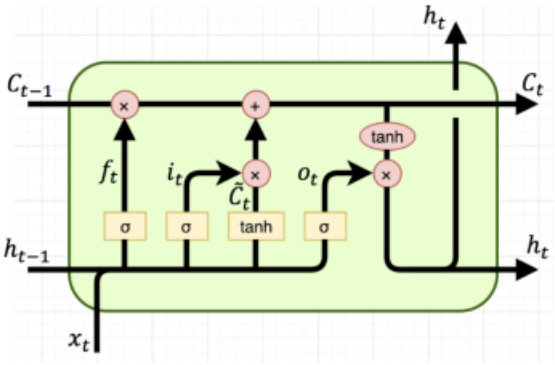


Figure 3.2: LSTM unit. Yellow boxes represent neural networks, pink ones operations on vectors

3.1.4.2 CNN

We developed two types of Convolutional Networks. First, 1D CNNs, as they offer a more flexible framework, which also results in the possibility of ingesting the data at raw frequency, while still covering a wide time interval and being practical in terms of training time and number of parameters; see a examples of 1D convolution Fig. 3.3. Convolutional Neural Networks (CNNs) are a type of Neural Network where a convolution operation is used instead of matrix multiplication (between input and weights), in at least some parts of the network. The weights of the neurons are convolutional filters, which are learned during training. CNNs can be employed when some spatial structure is present in the data, as they take into consideration spatial locality. This inductive bias is very general, and applicable to many practical scenarios; we are specifically interested in 1D CNNs, using time series data: here time is seen as a spatial dimension, and temporal locality is leveraged.

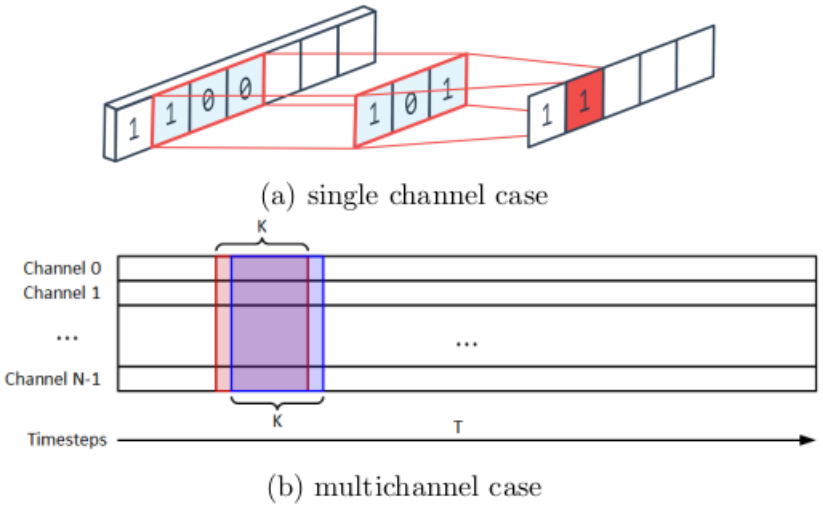


Figure 3.3: 1D convolution examples

We also offer the possibility to use Temporal Convolutional Networks, an architecture inspired by dilated computational schema of WaveNet[3]. Without entering too many details, we are interested in the overall computational schema that is employed: dilated causal convolutions. This configuration allows us to handle a large receptive field despite the high frequency of the data samples, while maintaining computational efficiency. The main characteristics are captured by the name:

- *Dilated*: the dilation factor is another possible parameter for CNNs, where the filter is applied in a "spread out" manner to the input; this factor increases exponentially from layer to layer; see Fig. 3.4. While many input samples are considered, the number of total neurons, and thus parameters and calculations, is contained, given the increasing sparseness in the deeper layers.
- *Causal*: a given convolution depends only on data from current and previous time steps: no leakage of information from future times.

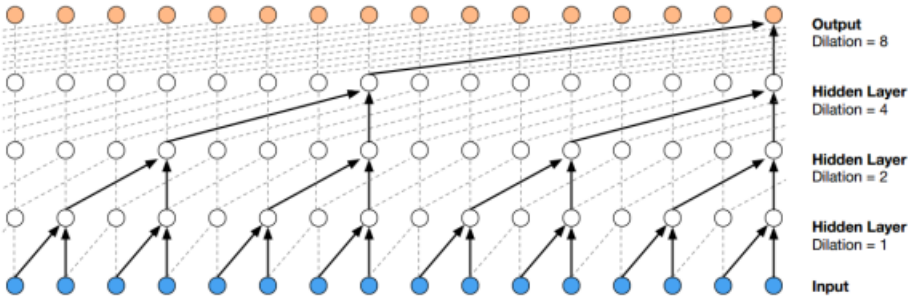


Figure 3.4: TCN (WaveNet) dilated causal convolution schema; stride factor is 1

Our case is simpler from a CNN architecture point of view, as we prepare sequences before feeding them to the model, in batches. This results in a non-sliding schema, while retaining the same advantages. This is realized by using only stride, the same for every layer, without a dilation factor. Using input sequences with a fixed size, which must be multiple of the stride factor, and no padding, the overall resulting schema for

[3] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. arXiv preprint arXiv:1609.03499, 2016

calculating an output at time t is equivalent to the WaveNet one. Recurrent blocks are used too. As stride is used, the skip connection needs to be adapted not only in depth (number of filters), but also in terms of output map length.

3.2 Multi-variable time-series supervised classification

Multi-variate time-series classification in a supervised context means being able to distinguish the difference among processes (or anything that can be described as a time-series) belonging to different classes. We assume to have a data set composed of a historical collection of *labelled* time-series, meaning that for each time-series in our data set there is an associated label reporting its class. For example, typical industrial processes can be described using a collection of temporally ordered data points; this time-series can then be classified depending on the outcome of the target process, e.g., the process has concluded with the expected outcome, or it has encountered a malfunction. Multi-variate indicates that the time-series is composed by multiple features, that is each data point belongs to a multi-dimensional space; multi-variate time-series are much harder to work with compared to uni-variate time-series (those time-series which include single features, e.g., the temperature measured by a meteorological station); in general, methods from the statistics area (e.g., auto-regressive approaches) have been shown to be extremely effective to model univariate time-series but much less capable of dealing with higher dimensionality. In this project we then decided to offer a service based on AI, namely on a data-driven approach, more precisely a Deep Learning model.

The multi-variable time-series classification service is based on a DL model, and it aims at distinguishing between the different classes recorded in the data. It requires a training set containing at least one example of all the classes of interest, albeit more than one example would be better. In case of extremely imbalanced data sets, a user might want to rely on a companion service developed as well during the IoTwins project, that is the multi-variable time-series data augmentation service, described in Sec. 3.3 of the current document. The DL mode used to distinguish the different classes of time-series must be capable of handling the temporal nature of the data, hence we restricted the pool of possible models to those described in the previous section as well, Recursive Neural Network, LSTM, Temporal Convolutional Neural Networks, and 1D Convolutional Networks. After a round of preliminary experiments, we converged on a specific architecture for the deployed service, namely a 1D convolutional network.

The service exposes two main methods: 1) `timeseries_classification` and 2) `ts_classification_inference`. The `timeseries_classification` method is needed to train the DL model, using training labelled data. The default hyperparameter describing the network topology is summarized in the following dictionary; clearly other values might be more suited to specific contexts and thus the user can change these values according to their needs – alternatively the hyperparameter optimization service (see Sec. 3.6 of this document) allows an automated fine-tuning depending on the desired goal:

```
default_hparams = {

'epochs': 2, # number of training epochs'batch_size': 64, # batch size

'conv1_layers': 1, # number of layers in the first convolution block

'conv2_layers': 1,     # number of layers in the second convolution block

'conv3_layers': 1,     # number of layers in the third convolution block
```

```
'conv1_filters': 64, # number of filters for the first convolution block'conv2_filters': 64,  # number of filters for the second convolution block

'conv3_filters': 64,  # number of filters for the third convolution block
'conv1_ks': 3,        # kernel size for the first convolution block

'conv2_ks': 3,        # kernel size for the second convolution block

'conv3_ks': 3         # kernel size for the third convolution block}
```

The service takes as input the data to be used for training and testing; this file must follow a specific format, described in the following. Every row of the first column of the file (<value_column>) contains a NxM matrix where N is the length of the timeseries, and M is the number of variables for the timeseries; N and M are parameters that need to be fed to the service as input. In addition, the service expects as input parameters the following ones: <value_column> which indicates name of the column containing the actual values of the time-series (the sequence of measurements composed of different features/variables); <label_column> which is the name of the column containing the label; <ts_index> which represents the name of the column indicating the index of the timeseries (multiple row in the input file can correspond to the same timeseries -- multi-variate timeseries, each row representing a variable).

The training data is provided to the method as a CSV file, <input_data>, fed as input to the training method timeseries_classification. A key aspect of the input data is that all time-series composing the data set must have the same length; this might require an additional pre-processing phase by users desiring to use this service (e.g., by performing gap-filling operation when not enough data is available). An example of the expected input data is the following (where the column <ID_var> indicates the identifier of the variable within the timeseries); in this case we have timeseries composed of 4 variables and T is the length of each timeseries; we also assume to have a single label for each timeseries and in this example the label is a integer number (different numbers indicates different classes):

```
ID_TimeSeries; ID_Var; Label; Values (separated with ‘,’)

0; 0; 1; V^{00}_1, V^{00}_2, … V^{00}_T

0; 1; 1; V^{01}_1, V^{01}_2, … V^{01}_T

0; 2; 1; V^{02}_1, V^{02}_2, … V^{02}_T

0; 3; 1; V^{03}_1, V^{03}_2, … V^{03}_T

1; 0; 1; V^{10}_1, V^{10}_2, … V^{10}_T

1; 1; 1; V^{11}_1, V^{11}_2, … V^{11}_T

1; 2; 1; V^{12}_1, V^{12}_2, … V^{12}_T

1; 3; 1; V^{13}_1, V^{13}_2, … V^{13}_T

….

N; 0; 1; V^{N0}_1, V^{N0}_2, … V^{N0}_T

N; 1; 1; V^{N1}_1, V^{N1}_2, … V^{N1}_T
```

```
N; 2; 1; V^{N2}_1, V^{N2}_2, … V^{N2}_T

N; 3; 1; V^{N3}_1, V^{N3}_2, … V^{N3}_T
```

Internally, the method performs a series of actions such as pre-processing the data, normalizing it, and then using it to train the neural network built accordingly to the hyperparameters (either the default ones or those specified by the user). As output, the `timeseries_classification` method provides the trained model (saved as Keras[4] model), the scaler model used to normalize the data (a binary object built using the scikit-learn Pyton library), and a text file summarizing the statistics computed over the validation set (part of the data automatically sampled from the input data set to perform validation checks – validation data is not used to train the DL model).

Then, the second method exposed by the service, `ts_classification_inference`, allows to use an already trained DL model to perform the classification task, that is to use the 1D Convolutional Network to predict the class of previously unseen, unlabeled data. The input expected by this method are the following: the length of the timeseries `N` and the number of variable composing each timeseries `M`; `<value_column>` which indicates name of the column containing the actual values of the timeseries (the sequence of measurements composed of different features/variables); `<ts_index>` which represents the name of the column indicating the index of the timeseries; `<input_data>`, which is the data that needs to be classified and it is expected to be a CSV file; `<scaler_object>`, that is the name of the binary file containing the scaler object used to normalize the training data (it is important to normalize new data according to the methods used for the training data). The output of the `ts_classification_inference` method is the list of predicted labels, expressed as a list of integer values, one label for each timeseries.

3.3 Multi-variable time-series data augmentation

A very common issue in industrial data and many real-world scenarios is the unavoidable imbalance in the data, and this is true for time-series (uni- or multi-variable) as well. When dealing with real, production systems, very few failures and anomalous situations are to be expected; this happens as industrial machines and large-scale systems are expensive to build and operate and are thus designed to be extremely robust and reliable. For instance, one hour of downtime for a data center could generate income losses for millions of Euros. Similarly, historical data sets typically used for training ML models contain very few rare events, by the same definition of rarity. Thus, some classes might be unrepresented with respect to other more common ones and this in turn causes great problem for supervised ML models which become more accurate as the training data encompasses *all* possible classes and in a balanced way, e.g., all classes are equally represented. This is very common, for instance, when dealing with classes that represent normal and anomalous states in industrial systems, as the anomalous classes (the failure or the warning states) are, by definition, much rarer than the normal state. To this end, multiple approaches have been studied, ranging from undersampling the majority class, interpolation to obtain additional data points, oversampling the minority class, etc. With this service we provide a data augmentation service to cope with the problem of imbalanced data, using an oversampling approach. The data augmentation service presented here only performs oversampling; it can be however combined with the general undersampling service described in D3.2, if domain knowledge suggests that a dual approach can be beneficial. The main idea is to generate synthetic samples by randomly selecting minority class time series that can act as guidance for generating new, similar series. This way the

[4] Keras is a Python library for Deep Learning, often used in conjunction with the TensorFlow DL environment.

synthetic samples will be close to the original time series in feature space, and they will consist of similar — yet not identical — sequences of values.

3.3.1 Sampling Synthetic Time Series Values

To generate the synthetic time series data points, we do not sample random numbers or just duplicate existing values. Instead, we pair each synthetic time series with a time series from the original dataset and we generate values which follow the same trend. Since we only know the length of the new time series, we pair synthetic time series to random original series which belong to the same window. The synthetic data points follow the trend of the paired time series values. The value at timestep *t* in a synthetic time series will be like a data point close or equal to *t* in the paired original time series. This process consists of two steps:

1. First, a random position close to *t* in the original time series is selected. The random sampling is not uniform, but rather it follows the standard normal probability density function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2},$$

Where $\mu = t$ is the position of the synthetic data point, and the standard deviation σ determines the amount of dispersion away from the mean. The lower the standard deviation, the higher the probability of choosing an original data point close to timestep t. Since the random sampling returns floating point values, the result is truncated to obtain the reference timestep.

The probability of choosing a reference data point close to timestep *t* (e.g., *t − 3, . . ., t + 3*) in the original time series is higher than selecting a point far in the "past" (e.g., *t − 6, t − 7, t − 8, . . .)* or far in the "future" (e.g., *t + 6, t + 7, t + 8, . . .)*, as shown in Fig.

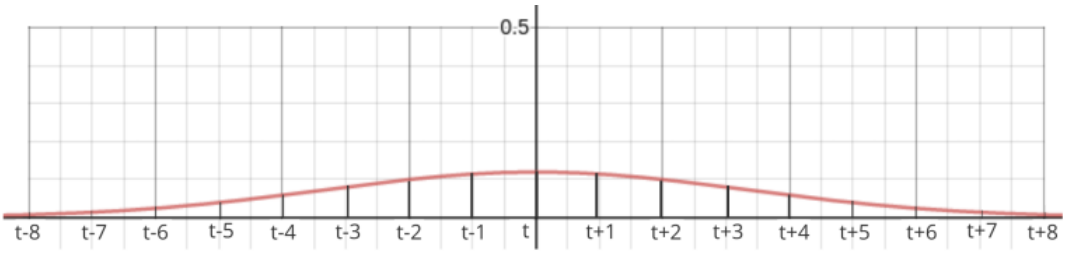


Figure 3.5: Random sampling probability density function

2. To introduce a slight randomness in the synthetic time series values and not merely duplicate the original data points, we randomly compute a new value close to the selected original one. The goal is to sample a random value inside a circle at the center of which is the selected data point. Since the time series may have high dimensionality, a uniformly random point on a hypersphere is generated. A simple way to generate points uniformly at random on the interior of a *d*-sphere is by the generalization of the Muller algorithm [28]. The procedure consists of:
    a. Generating *d* random variables x_i, *i = 1, 2, . . ., d* normally distributed in the range [0, 1].
    b. For each point x = (x_1, x_2,. . , x_d), with *r* being a normally distributed random variable which determines the distance from the center of the sphere, the equation for locating a point *y* on the unit *d*-sphere having the *d*-direction cosines is given by:

$$\frac{r \cdot x_i}{\sqrt{\Sigma_{i=1}^{d} x_i^2}}, \qquad i = 1, 2, \ldots, d,$$

Figure 3.6 broadly illustrates the oversampling process described above. Time series (a) represents a job in the original dataset, the timesteps are placed on the x-axis and the values on the y-axis. For this example, the series has only one dimension. Time series (b) shows the reference points, which are randomly sampled from (a), for each new synthetic data point. For instance, for the fifth data point in (b), the algorithm randomly chose the fourth data point (i.e., the (t − 1)-th element) in the paired time series. Time series (c) is equivalent to (b) but with a slight randomization of series values using the Muller algorithm. A moving average can also be applied to smooth the resulting values. In conclusion, time series (c) has a similar (but not identical) trend to time series (a), which is the goal of this oversampling technique.

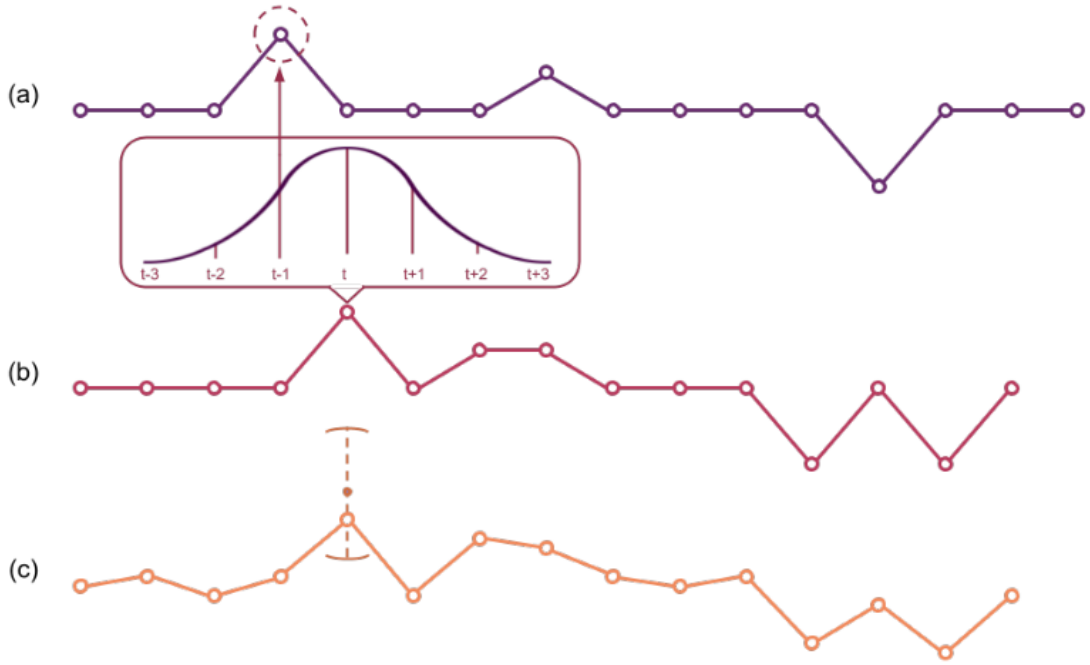


Figure 3.6: Oversampling process. (a) is an original time series selected as reference to sample a new synthetic one. (b) each point at time t in the synthetic time series corresponds to a data point in the paired series close or equal to time t. (c) the selected values in the step before are slightly randomized

3.4 Contextual Recurrent Neural Network

The core goal of the model is to learn and to forecast some time series. The method is based on types of Deep Learning model called Recurrent Neural Network [Connor et al., 1994] and Gated Recurrent Unit [Cho et al., 2014], and use embedding vector [Guo and Berkhahn, 2016] to add contextual information. Time series forecasting often used past observation to predict future instances. However, the time series future can depend on several exogenous features that can be used to improve forecasting quality. For this component, we work in a context where we want to predict times series coming from several sensors which can interact together. For instance, it can be wind turbine field or the passenger flow in a subway. We propose to learn such spatial features in embeddings vectors within recurrent neural networks.

3.4.1 Methodologies

We consider $N*S$ samples $\boldsymbol{xs} \in \mathbb{R}T$, with N the number of days, that are encoded into a singular hidden state. Moreover, there is a single couple (E, D) shared across all the spatial features that allows us to consider the correlations between the stations and reduces the number of weights. This time, spatial context is explicitly learned in the form of a matrix of spatial embedding $\boldsymbol{Z}s \in \mathbb{R}S \times \lambda s$. An embedding allows representing a categorical feature in continuous variables. The advantages compare to a one-hot encoding is that we can choose the embedding dimension and the representation is less sparse. For each spatial feature s, the corresponding embedding $\boldsymbol{Z}s$ is concatenated to the observation. At time step t, for a spatial feature s, the observation is concatenated to the embedding of size λs. The resulting vector and the hidden state are encoded via the common encoder E into a hidden state representing only this station. This state is then decoded into a single-valued prediction. See [Thouvenot et al., 2019] for more details.

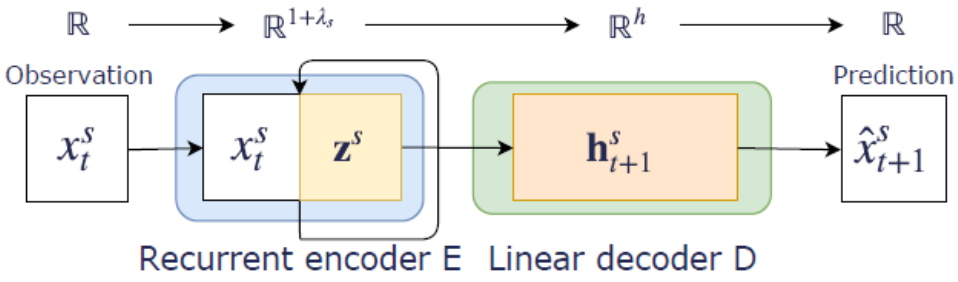


Figure 3.7: Approach architecture

3.4.2 API

In this section we provide the programming interface of the contextual RNN service. This API can be used to extend and/or tailor the behaviour of the generic service to the specific testbed. The API provided here follows the same format used in the GitLab repository.

spatial_train – Train the time series forecasting model.

- Param df_fname : string
  - name of the csv file containing the dataframe (data to use for training)
  - It must be a csv file, each field is separated by a separator symbol (see «separator» parameter). Each row corresponds to a day of series for one spatial feature
  - First column must be termed «sid» and consists of the spatial feature. Second column must be termed «date» and consists of the date. The following columns contain the time series associated to the data and the spatial feature
- param filename_performance : string
  - name of a csv file where some information about model performance is saved
- param filename_serie : string
  - start of name of text files where the predicted series are saved.
  - there is one text by day and spatial feature
- param filename_model : string
  - name where the model is saved in pickle format
- param nb_stations : integer
  - Number of unique spatial features
- param seq_size : interger

- Number of instances in a day, e.g., if the time series is hourly, then seq_size = 24
- param separator: string
  - Separator used in csv file
  - default value is «, »
- param hparams_file (optional): string
  - name of pickle file containing the python dictionary with hyperparameters to be used to build the contextual RNN
- return df: string
  - performance of the model on the training set

This an example (and the default) dictionary containing the hyperparameters values for the function spatial_train.

```
default_hparams_train = {

    'epochs': 100, # number of training epochs

    'mode': "rnn", # class of neural network used

    'batch_size': 200, # batch size

    'hidden_size': 200, # Hidden size

    'embedding_size': 80, # Size of embedding vector

    'decoder_class': lin, # Decoder type used, implemented in models file

    'lr': 0.001, # learning rate

    'remove_first' : True, #remove first row of the datasets

    'register_loss' : True, # Register the loss function during the training

    'save_prediction' : False # Save the predictions

}
```

spatial_validation – Apply a contextual RNN on a validation set to evaluate the model performance in generalization

- Param df_fname : string
  - name of the csv file containing the dataframe (data to use for validation, which has not been used for the model training)
  - it must be a csv file, each field is separated by a separator symbol (see « separator » parameter). Each row corresponds to a day of series for one spatial feature
  - First column must be termed « sid » and consists of the spatial feature. Second column must be termed « date » and consists of the date. The following columns contains the time series associated to the data and the spatial feature
- param filename_performance : string
  - name of a csv file where some information about model performance is saved
- param filename_serie : string

  - start of name of text files where the predicted series are saved.
  - there is one text by day and spatial feature
- param filename_model : string
  - name where the model is loaded in pickle format
- param nb_stations : integer
  - Number of unique spatial features
- param seq_size : integer
  - Number of instances in a day, e.g., if the time series is hourly, then seq_size = 24
- param separator: string
  - Separator used in csv file
  - default value is «, »
- param hparams_file (optional): string
  - name of pickle file containing the python dictionary with hyperparameters to be used to build the contextual RNN
- return df: string
  - performance of the model on the training set

This an example (and the default) dictionary containing the hyperparameters values for the function spatial_validation.

```
default_hparams_test = {

      'remove_first' : True, #remove first row of the datasets

      'batch_size': 16, # batch size

      'hidden_size': 200, # Hidden size

      'embedding_size': 80, # Size of embedding vector

      'mode': "rnn", # class of neural network used

      'decoder_class': lin, # Decoder type used, implemented in models file

      'save_prediction' : False # Save the predictions

}
```

spatial_predict – Apply a contextual RNN to predict some time series. The data provided consists of the first instants of a day, the algorithm then predicting the remaining instants of the day by injecting the predictions of t-1, t-2, etc. to predict t, if the true values of t-1, t-2, etc. are unknown, and the true values if they are known.

- Param df_fname : string
  - name of the csv file containing the dataframe (data to predict)
  - it must be a csv file, each field is separated by a separator symbol (see « separator » parameter). Each row corresponds to a day of series for one spatial feature
  - First column must be termed « sid » and consists of the spatial feature. It must be numerated from 0 to N-1, with N the total number of unique values of spatial feature. Second column must be termed « date » and consists of the date. The following columns contains the time series associated to the data and the spatial feature

- param filename_serie : string
  - start of name of text files where the predicted series are saved.
  - there is one text file by day and spatial feature
- param filename_model : string
  - name where the model is loaded in pickle format
- param nb_stations : integer
  - Number of unique spatial features
- param seq_size : integer
  - Number of instances in a day, e.g., if the time series is hourly, then seq_size = 24
- param start: integer
  - Time start of the prediction. If start = 8, then the 8 first instants are supposed known, and then the remaining instants are predicted. Caution: as we inject the predictions of the previous moments to predict the farthest moments, the quality of the predictions is likely to be low
- param separator: string
  - Separator used in csv file
  - default value is «, »
- param hparams_file (optional): string
  - name of pickle file containing the python dictionary with hyperparameters to be used to build the contextual RNN

This an example (and the default) dictionary containing the hyperparameters values for the function spatial_predict.

```
default_hparams_forward = {

    'remove_first' : False, #remove first row of the datasets

    'batch_size': 16, # batch size

    'hidden_size': 200, # Hidden size

    'embedding_size': 10, # Size of embedding vector

    'mode': "rnn", # class of neural network used

    'decoder_class': lin, # Decoder type used, implemented in models file. Can be MLP2 too.

}
```

3.4.3 Software requirement

The contextual RNN service was developed using Python language. In the Docker provided, the list of requirements is:

- Python 3.8
- Python module:
  - torch==1.4.0
  - torchvision==0.2.1
  - numpy==1.19.1

- pandas==1.1.0
- tqdm==4.31.1
- matplotlib==3.0.2
- pickleshare==0.7.5
- datetime==4.0.1

3.4.4 Service usage

In this section we illustrate the use of the component and explain the inputs/outputs.

3.4.4.1 Use case description

We illustrate the use of the component on GefCom 2012 dataset [Hong et al., 2014]. Short term load forecasting provides load forecasts at hourly or sub hourly intervals for the following one day to two weeks. The forecasts are used by all sectors of the utility industry, from generation and transmission to distribution and retail. The reasons why businesses need short term load forecasts include unit commitment, T&D (transmission and distribution) operations and maintenance, and energy market activities. In the hierarchical load forecasting track, the participants were required to forecast hourly loads (in kW) for a US utility with 20 zones. They are 4.5 years of hourly load history data. For this application, we consider the year 2004, 2005 and 2006 as the training set and 2007, 2008 as testing sets, and we forecast the load at one hour timestamp.

3.4.4.2 Dataset format description

sid	date	t0	t1	...	t23
0	2004-1-1	0.0373601184670225	0.0364667387873091	...	0.0326981396421161
0	2004-1-2	0.0313791299412985	0.0311197616471882	...	0.0338242857396208
0	2004-1-3	0.0320087076808484	0.0316384982354089	...	0.0275418092822814

- `sid` corresponds here to the zone. Its values are from 0 to 19 according to the zone
- `date` corresponds to the date
- Then, there are 24 columns, one for each hour of day. The load has been normalized before the datasets was used for the training

3.4.4.3 Application of spatial_train_function

The objective of the function is to train the neural network. We use the following hyperparameters:

```
default_hparams_train = {

      'epochs': 100, # number of training epochs

      'mode': "rnn", # class of neural network used

      'batch_size': 200, # batch size

      'hidden_size': 200, # Hidden size

      'embedding_size': 10, # Size of embedding vector

      'decoder_class': lin, # Decoder type used, implemented in models file

      'lr': 0.001, # learning rate

      'remove_first' : True, #remove first row of the datasets

      'register_loss' : True, # Register the loss function during the
      training
```

```
    'save_prediction' : False # Save the predictions
}
```

There are 20 stations. According to these hyperparameters, we will represent 10 continuous features for this information.

Then the model can be used like that:

```
df_name = "gefcom-train.csv"

filename_performance = "performance-gefcom.csv"

filename_serie_train = "train-gefcom"

filename_model = "model-gefcom.pth"

spatial_train(df_fname = df_name,

nb_stations = 20,

seq_size = 24,

filename_performance = filename_performance,

filename_serie_train = filename_serie_train,

filename_model = filename_model)
```

In command line, that gives:

```
python scr/rnn_main.py 0 <df_name> <filename_ performance> <filename_serie_train> <filename_model> 20 24
```

[5]

Figure 3.8 is automatically saved and it gives the loss value in function of the number of epochs. It helps to choose the hyperparameters. Performance is also saved. Here the model performance on the training sets is given by Table below; absolute value for the loss is reported on the y-axis.

[5] python scr/rnn_main.py 0 "/home/vincent/Documents/gefcom/gefcom/gefcom-train.csv" "/home/vincent/Documents/gefcom/gefcom/performance.csv" /home/vincent/Documents/gefcom/gefcom/serie" "/home/vincent/Documents/gefcom/gefcom/model.pth" 20 24

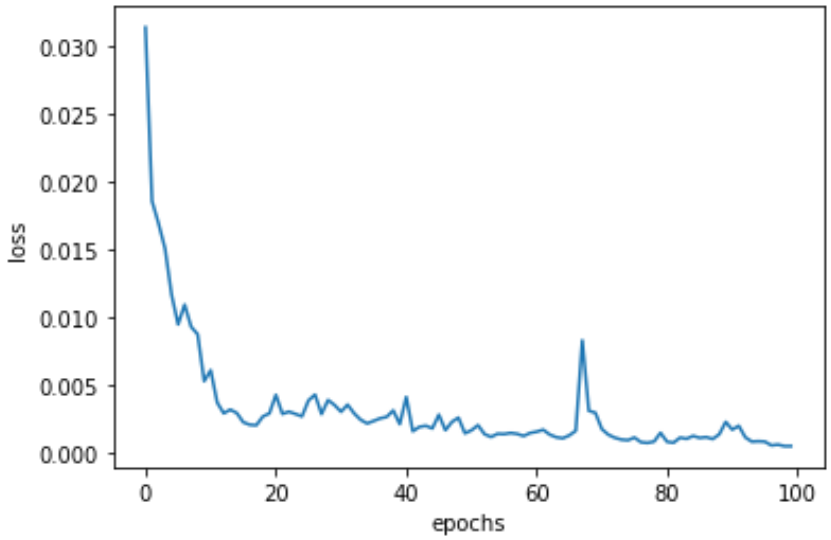


Figure 3.8: Loss function evolution

The Root Mean Square Error and it is equal to 0.002; the Mean Absolute Error is equal to 0.03.; the smooth L1 loss is 0.001. The computation time is 185 seconds (about 3 minutes).

The trained model is saved. More precisely, we save the state_dict() of the model. In PyTorch, the learnable parameters (i.e., weights and biases) of a torch.nn.Module model are contained in the model's *parameters* (accessed with model.parameters()). A *state_dict* is simply a Python dictionary object that maps each layer to its parameter tensor. Note that only layers with learnable parameters (convolutional layers, linear layers, etc.) and registered buffers (batchnorm's running_mean) have entries in the model's *state_dict*. Optimizer objects (torch.optim) also have a *state_dict*, which contains information about the optimizer's state, as well as the hyperparameters used. Because *state_dict* objects are Python dictionaries, they can be easily saved, updated, altered, and restored, adding a great deal of modularity to PyTorch models and optimizers.

Moreover, the user can choose to save all the series and their predictions.

3.4.4.4 Application of spatial_validation function

The objective is to test a trained model on a new dataset to evaluate the model's performance.

```
spatial_validation(df_fname = df_name,
      nb_stations = 20,
      seq_size = 24,
      filename_performance = filename_performance,
      filename_serie_test = filename_serie_test,
      filename_model = filename_model)
```

In command line:

```
python scr/rnn_main.py 1 <df_name> <filename_ performance> <filename_serie_train> <filename_model> 20 24
```
[6]

The output of the function is the performance on the testing set, namely Root Meas Squared Error equal to 0.0007, Mean Absolute Error equal to 0.05 and Smooth L1 Loss equal to 0.0035.

Again, the user can save the series.

3.4.4.5 Application of spatial_predict function

The objective of the function is to predict the resulting instants of one day for all the elements of the contextual feature. For our example, we want to predict the load for all the stations the 2007-1-10 from 9AM to the end of the day. We assume that all the loads are known until 8h and predict with model the load. To predict the load at 10AM we inject the prediction made for 9AM and so on. The more we inject the prediction, the more we inject errors and uncertainty, so this prediction should be done as soon as possible when we have new information.

In command line (see also[7]): `python scr/rnn-main.py 2 <df_name> <filename_model> 20 24 8`

[6] E.g., python scr/rnn_main.py 1 "/home/vincent/Documents/gefcom/gefcom/gefcom-train.csv" "/home/vincent/Documents/gefcom/gefcom/performance.csv" /home/vincent/Documents/gefcom/gefcom/serie" "/home/vincent/Documents/gefcom/gefcom/model.pth" 20 24

[7] E.g., python scr/rnn-main.py 2 <data_file> <model_name> 20 24 8 "/home/vincent/Documents/gefcom/gefcom/gefcom-test-truncated-2007-1-10.csv" "/home/vincent/Documents/gefcom/gefcom/pred/serie_test_truncated" "/home/vincent/Documents/gefcom/gefcom/model.pth" 20 24

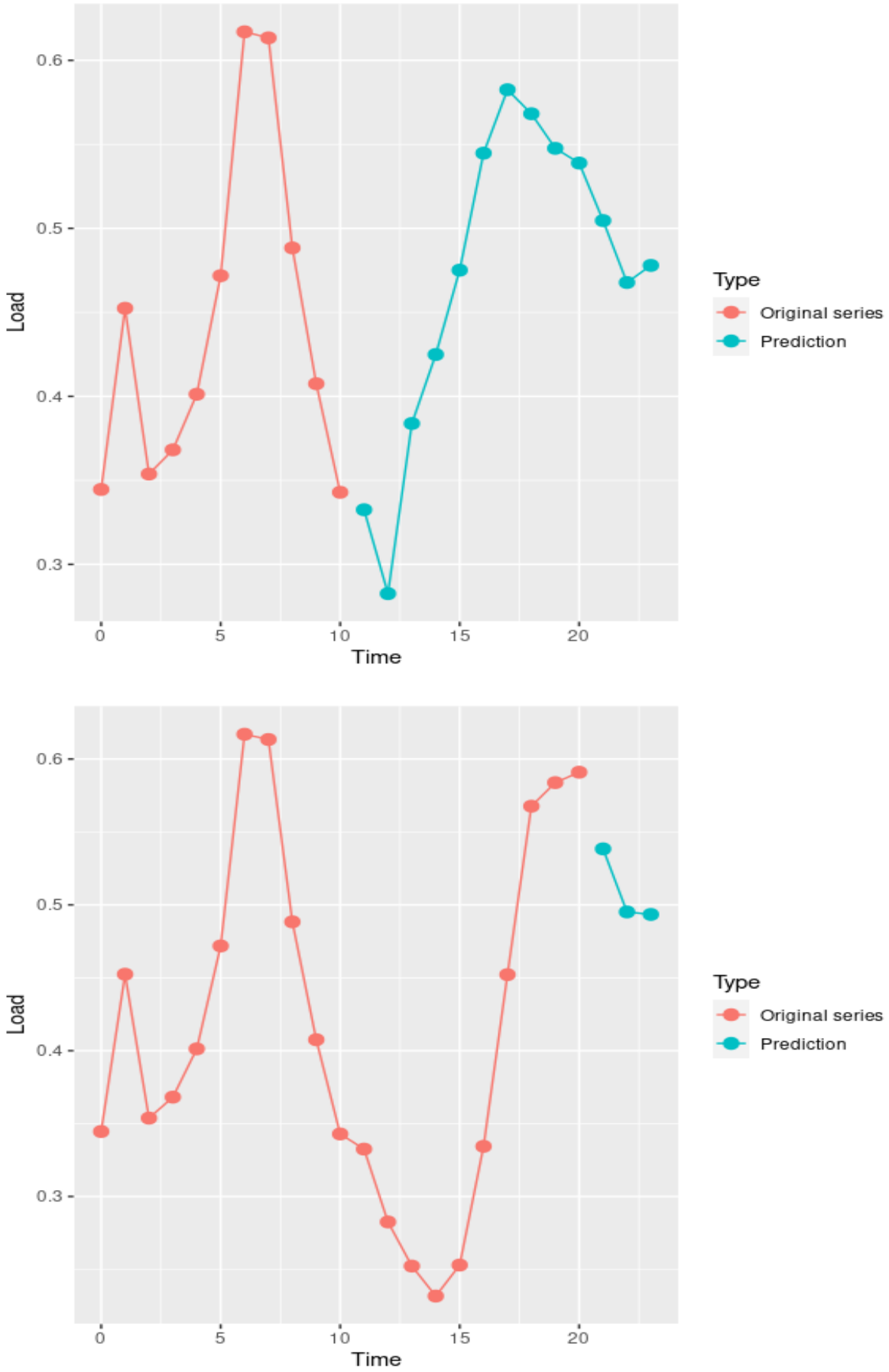


Figure 3.9: Example of the application of the spatial prediction service

3.4.5 Reference

[Cho et al., 2014] Cho, van Merrienboer, Gulcehre, Bahdanau, Bougares, Schwenk, and Bengio, Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation, 2014 |

[Connor et al., 1994] Connor, Martin, and Atlas, Recurrent neural networks and robust time series prediction, 1994

[Guo and Berkhahn, 2016] Guo and Berkhahn, Entity Embeddings of Categorical Variables, 2016

[Hong et al., 2014] Hong, T., Pinson, P., & Fan, S. (2014). Global energy forecasting competition 2012. *International Journal of Forecasting*, 30(2), 357-363. https://doi.org/10.1016/j.ijforecast.2013.07.001.

[Thouvenot et al., 2019] Baskiotis, Becirspahic, Brault, Duguet, Guigue, Guiguet, Thouvenot A Deep Approach of Affluence Forecasting in Subway Networks. *WCRR 2019 - 12th World Congress on Railway Research*, Oct 2019, Tokyo, Japan.

3.5 Local explanation for machine learning model based on Shapkit

3.5.1 Methodology

Machine Learning models are used for various applications with already successful results. Unfortunately, a common criticism is the lack of transparency associated with these algorithm decisions. This is mainly due to a greater interest in performance (measurable nonspecific tasks) at the expense of a complete understanding of the model. Global method of interpretability aims at explaining the general behaviour of a model, whereas a local method focuses on each decision of a model. The agnostic category (also called post-hoc explanation) considers the model as a black box. On the other hand, inherent or non-agnostic methods can modify the structure of a model or the learning process to create intrinsically transparent algorithms.

Local explanation focuses on a single instance and examines what the model predicts for this input and explains why. In Collaborative Game Theory, Shapley Values [Shapley, 1953] allows to distribute a reward fairly among players according to their contribution to the win in a cooperative. Shapley Values offer a solution of local explanation from additive feature importance measure class ensuring desirable theoretical properties. A prediction can be explained by assuming that each feature value of the instance is a “player” in a game where the prediction is the payout. The objective is to fairly distribute the payout around all features to obtain the prediction. We can make the following correspondence between Game Theory and model interpretability:

- The features are the players.
- The model is the game.
- The feature attribution is the gain attribution.

A major challenge for Shapley Values computing is the number of calculi to perform: the potential coalition, and so this number, grows exponentially in function of the number of features. Some approaches to approximate them has been proposed [Strumbelj and Kononenko, 2014] [Lundberg and Lee, 2017] [Aas et al., 2019] [Grah and Thouvenot, 2020]. Shapkit [Grah and Thouvenot, 2020b] is a Python module to use Shapley Values as local explanation of Machine Learning model, based both on Monte Carlo approximation

and weighted least square optimization problem. The method is a post-hoc explanation, so the user does not have to change her usual pipeline.

In a first use case, a machine learning model outputs the probability that a given individual makes over 50000$ a year. Assuming that the algorithm is quite effective, however it only returns a probability score without any details on how it has made its choice. We would like to know how each attribute influences the model output. Furthermore, contributions explain the difference between the individual prediction and the mean prediction for all references. These references are defined by the user. As an example, in that case, references are selected into another predicted class. That is to say, if we want to interpret the result of an individual classified into the high-income class, we pick as references other individuals predicted as earning less than 50000$ a year.

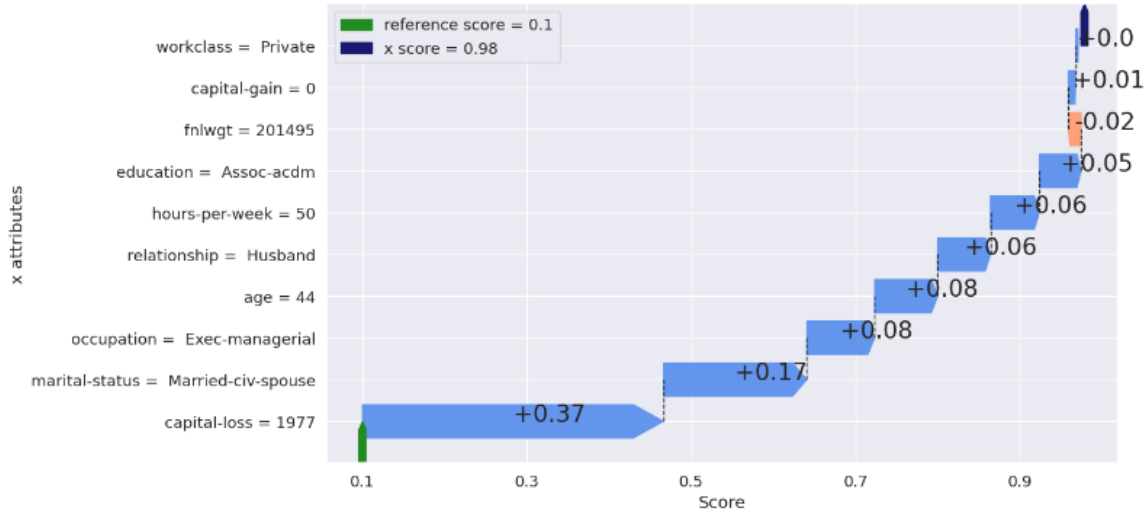


Figure 3.10: Example of the results provided by the local explanation service for income predictions

Figure 3.10 displays the kind of interpretation associated to a given prediction for the individual x. We want to understand the model decision associated to an individual x. As an example, here the selected individual has a probability of 98% to earn more than 50000$ a year (the blue tick at top right). Attribute importance are computed with respect to one or several references. On this example, we chose only predicted low-income class individuals as good elements of comparison. The mean probability for that group of references is about 1% (green tick at the bottom left). Finally, the gap between our individual prediction and the mean reference prediction is split by the attribute importance. The sum of all contributions equals that difference. Now, we can analyze that the capital-loss, the “married-civ-spouse” status and the “exec-managerial” occupation of the individual x increase respectively by 37%, 17% and 8% the estimated probability.

We propose an API to deal with three supervised machine learning tasks: regression, binary classification, and multi-label classification. For regression, the reference population is the entire population. In this case, the Shapley value computed are about the difference between the prediction made for an average instance and the instance of interest. We are interested here in studying the difference between the prediction for the opposite class and the instance of interest. We look at the features that contribute the most to change the class. For multi-label classification, we compute the Shapley value according to the maximum probability of the instance of interest and we consider as reference population the population predicted as another class than the instance to explain, or one class chosen by the user.

Task	Reference population	Value of interest
Regression	Whole dataset	The model prediction f(x)
Binary classification	Opposite class instances	The score given by the model $P_f(Y=1)$
Multi-label classification	Another classes observation that the instance of interest or a class chosen by the user	The score given by the model $P_f(Y=s)$ with s the class predicted for the instance of interest.

3.5.2 API

We propose two services: one dedicated to sklearn model format used for regression, binary classification, and multi-label classification and the second one for Tensorflow model format used for regression, binary classification, and multi-label classification.

3.5.2.1 Sklearn model format

shapley_computation – Compute the Shapley Values for a given instance.

- Param path_model: string
  - name of the pickle file containing the model to explain
  - it must a pickle file with a model train with sklearn (or any Python module with similar format, e.g. catboost)
- param path_data : string
  - name of a csv file where the data are loaded. Data must be in the same format that the data used for train the model after data processing
- idx_individual : integer
  - Index of the instance to explain
- param path_download_shap : string
  - start of name of csv files where the Shapley values are saved.
- param type: string
  - Type of model considered
  - Can be “regression”, “classification” or "multiclassification”, respectively for a regression task, a binary classification and a multi-label classification
- param n_ref: integer
  - Size of reference population sample
  - If None, then all the reference population is used
- param n_attribrute : integer
  - Number of features keeping for the plot
  - Only n_attribute features with the strongest absolute value of Shapley values are kept
  - If None, then all features are represented
- param path_image: string
  - path to save the plot
  - default value is “fig.png”
- param missing_data: string
  - Format of missing data in the original dataset, e.g., NA
  - If None, no missing data in the dataset

- param header: integer
  - Raw with the header in the dataset
  - Default value is 0
- param hparams_file (optional): string
  - name of pickle file containing the python dictionary with hyperparameters to be used

This is an example (and the default) dictionary containing the hyperparameters values.

```
default_hparams = {

    "path_reference": None # string. Path to load a csv file with the reference instances (at the same format that the training data after pre-processing. Default is None. In this case, reference data come from the data load with path_data parameter

    , "method":None # String. Shapley Value approximation used. Default, None, is MCBatch. Other techniques are "mc", for monte carlo approximation, and "sgd"for a projected stochastic gradient algorithm

    , "n_iter":10000 # Number iteration of approximation algorithm. Default is 10000

    , "savefig" : True # True or False according if we save or not the figure with shapley value approximation

    , "fig_format":"png" # Format of the saved figure. Default is png

    , "no_show":False # True or Flase. Does the Figure be open?

    , "step":.1 # step hyperparamter

    , "step_type":"sqrt" # type of step used

    , "threshold":0.5 # For binary classification task, what is the threshold of classification? In (0,1)

    , "multi_class_target_pop":None # For multi-label classification. Determine the reference population. If None, reference population is the entire population predicted in a different class that the instance to explain. If an integer, the reference population is the predicted in the class given by path_data.

}
```

3.5.2.2 TensorFlow model format

shapley_computation – Compute the Shapley Values for a given instance.

- Param path_model: string
  - name of the file containing the model to explain
  - It must a file with a model train with Keras/Tensorflow and saved with save attribute of the model
- param path_data : string

    - name of a csv file where the data are loaded. Data must be in the same format that the data used for train the model after data processing
- idx_individual : integer
    - Index of the instance to explain
- param path_download_shap : string
    - start of name of csv files where the shapley values are saved.
- param type: string
    - Type of model considered
    - Can be “regression”, “classification” or "multiclassification”, respectively for a regression task, a binary classification, and a multi-label classification
- param n_ref: integer
    - Size of reference population sample
    - If None, then all the reference population is used
- param n_attribute : integer
    - Number of features keeping for the plot
    - Only n_attribute features with the strongest absolute value of Shapley values are kept
    - If None, then all features are represented
- param path_image: string
    - path to save the plot
    - default value is “fig.png”
- param missing_data: string
    - Format of missing data in the original dataset, e.g., NA
    - If None, no missing data in the dataset
- param header: integer
    - Raw with the header in the dataset
    - Default value is 0
- param hparams_file (optional): string
    - name of pickle file containing the python dictionary with hyperparameters to be used

This is an example (and the default) dictionary containing the hyperparameters values.

```
default_hparams = {

    "path_reference": None # string. Path to load a csv file with the reference instances (at the same format that the training data after pre-processing. Default is None. In this case, reference data come from the data load with path_data parameter

    , "method":None # String. Shapley Value approximation used. Default, None, is MCBatch. Other techniques are "mc," for monte carlo approximation, and "sgd"for a projected stochastic gradient algorithm

    , "n_iter":10000 # Number iteration of approximation algorithm. Default is 10000

    , "savefig" : True # True or False according if we save or not the figure with shapley value approximation
```

```
    , "fig_format":"png" # Format of the saved figure. Default is png

    , "no_show":False # True or Flase. Does the Figure be open?

    , "step":.1 # step hyperparamter

    , "step_type":"sqrt" # type of step used

    , "threshold":0.5 # For binary classification task, what is the
    threshold of classification? In (0,1)

    , "multi_class_target_pop":None # For multi-label classification.
    Determine the reference population. If None, reference population is
    the whole population predicted in a different class that the instance
    to explain. If an integer, the reference population is the predicted
    in the class given by path_data.

}
```

3.5.3 Software requirement

The two local explanation services were developed using Python language.

In the provided docker dedicated to modeling using Sklearn format, the list of requirements is:

- Python 3.8
- Python modules:
  - numpy==1.20.0
  - pandas==1.1.0
  - seaborn==0.10.1
  - shapkit==0.0.3
  - fire==0.3.1
  - scikit-learn==0.22

In the provided docker dedicated to modeling using TensorFlow format, the list of requirements is:

- Python 3.8
- Python module:
  - numpy== 1.17.3
  - pandas==1.1.0
  - seaborn==0.10.1
  - shapkit==0.0.3
  - fire==0.3.1
  - tensorflow==2.3.0

3.5.4 Service usage

In this section we illustrate the use of the component and explain the inputs/outputs. We illustrate the two services on regression case, on binary classification case and on multi-label classification case. To simplify the user manual and avoid repetition, we only demonstrate the use of service based on TensorFlow models.

In command line, the service can be launched according to the command below:

```
sudo docker run -p 9091:9091 shaptf python scr/tensorflow-format-main.py
"data/iris-tf" "data/iris.csv" 0 "data/download.csv" multiclassification
```

where:

- 9091:9091 is the port used
- shaptf is the docker image name
- python is used to launch a python script
- scr/tensorflow-format-main.py is the path to the script to launch
- "data/iris-tf" is the path the model
- "data/iris.csv" is the path to the dataset
- 0 is index of the instance of interest
- "data/download.csv" is the location where the Shapley values are saved
- multiclassification is the task (regression, classification or multiclassification)

3.5.4.1 Regression case

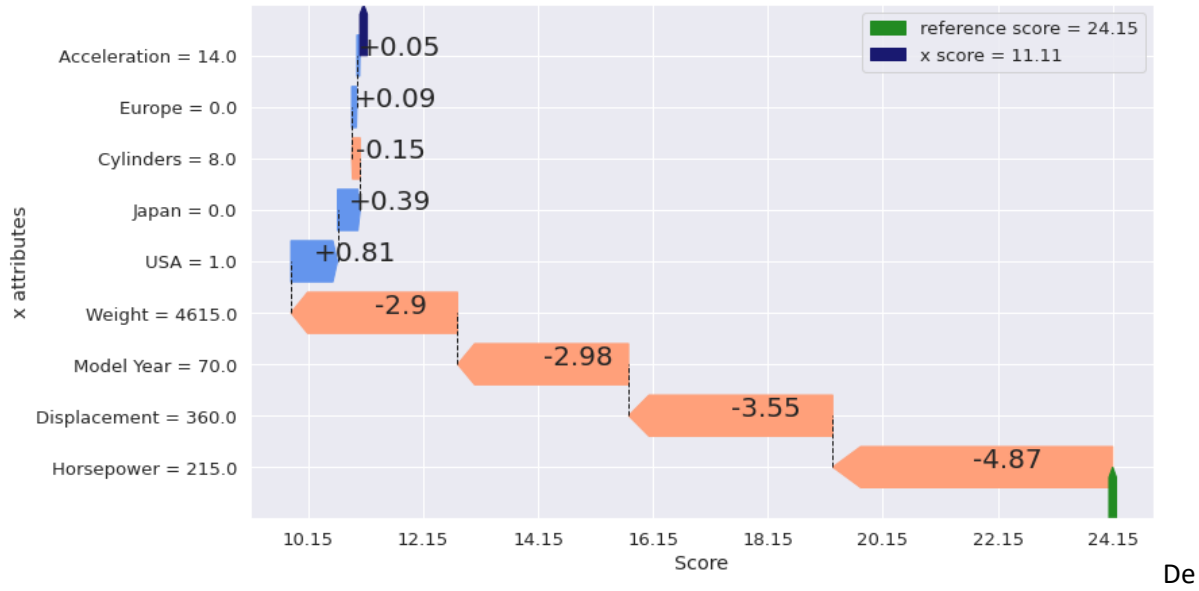
The simpler case is the regression case. The reference population is always (a sample of) the entire population. In the function below, we load a Keras model saved in the holder "mpg." The data[8] concerns city-cycle fuel consumption in miles per gallon, to be predicted in terms of 3 multivalued discrete and 5 continuous attributes. The Keras model is a neural network with a normalization layer, with hidden dense layers using ReLU activation function and one dense output layer. We are interested in the second instance of the dataset. According to the shapley_computation function sample 100 instances from the dataset, which will be the reference population. As n_attributes=None in the function, we keep all the features in the plot.

```
default_hparams_2={
    "path_reference":None
    , "method":None
    , "n_iter":10000
    , "savefig" : True
    , "fig_format":"png"
    , "no_show":False
    , "step":.1
    , "step_type":"sqrt"
    , "threshold":None
    , "multi_class_target_pop":0
}


shapley_computation(path_model="mpg"
                        , path_data="auto-mpg-test.csv"
                        , idx_individual=1
                        , path_download_shap="shap_mpg.csv"
                        , path_image="fig-mpg.png"
                        , type="regression"
                        , n_ref=100
                        , n_attributes=None
                        , missing_data=None
                        , header=0
                        , hparams_file=default_hparams_2)
```

Figure 3.11: Setup of the local explanation service for the regression task (miles-per-gallon estimation)

[8] http://rasbt.github.io/mlxtend/user_guide/data/autompg_data/



De

Figure 3.12: Example of the results provided by the local explanation service for the regression task (miles-per-gallon estimation)

The plot above explains how the features contribute to deviation to the prediction for an average instance and the instance of interest. For instance, according to the Shapley values, for the model, the value of the horsepower decreases the value predicted for the instance of interest.

3.5.4.2 Binary classification case

In the case of the classification task, the choice of the reference population is trickier than for the regression task. By default, the reference population is the instances that are predicted in the other class than the instance of interest when the class is predicted using the simple rule if the model score is greater than 0.5, then the instance is predicted as class 1, else as class 0. The user can change this value of 0.5 by modifying threshold parameter in the hparams_file as in the example below. The classification rule is then if the model score is greater than threshold, then the instance is predicted as class 1, else as class 0.

On the example below, we use a benchmark dataset about credit card fraud detection[9]. The model is a keras model with a first dense layer, a dropout layer, and a dense layer. We study the contribution of each feature to the prediction made for the first instance. The threshold of classification used here is 0.2. With this threshold, the instance is predicted as fraud. The features are the result of a PCA (Principal Component Analysis) transformation.

[9] https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

```
default_hparams_2={
    "path_reference":None
    , "method":None
    , "n_iter":10000
    , "savefig" : True
    , "fig_format":"png"
    , "no_show":False
    , "step":.1
    , "step_type":"sqrt"
    , "threshold":0.2
    , "multi_class_target_pop":0
}


shapley_computation("credict-tf"
                            , path_data="credit.csv"
                            , idx_individual=0
                            , path_download_shap="shap_credit.csv"
                            , path_image="fig.png"
                            , type="classification"
                            , n_ref=None
                            , n_attributes=None
                            , missing_data=None
                            , header=0
                            , hparams_file=default_hparams_2)
```

Figure 3.13: Setup of the local explanation service for the binary classification task (credit card fraud detection)

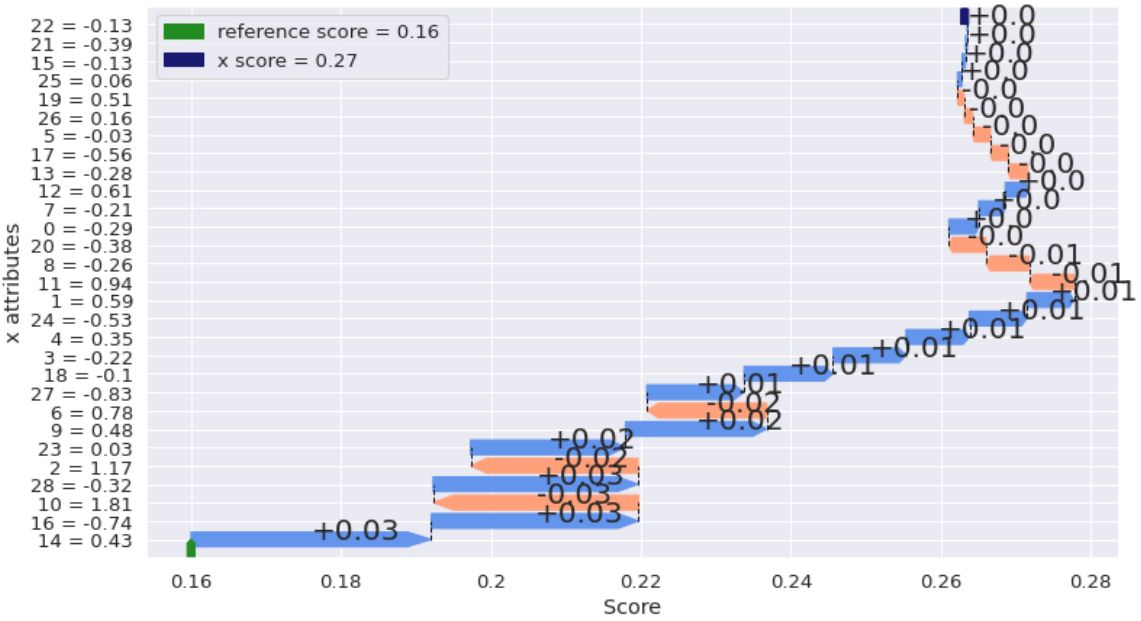


Figure 3.14: Example of the results provided by the local explanation service for the binary classification task (credit card fraud detection) - threshold 0.2

In the second case below, we consider that the threshold is 0.5, as we use the default value, thanks to the element "threshold":None. With this threshold, the instance is classified as non-fraudulent. The reference population is the instance that has a score greater than 0.5.

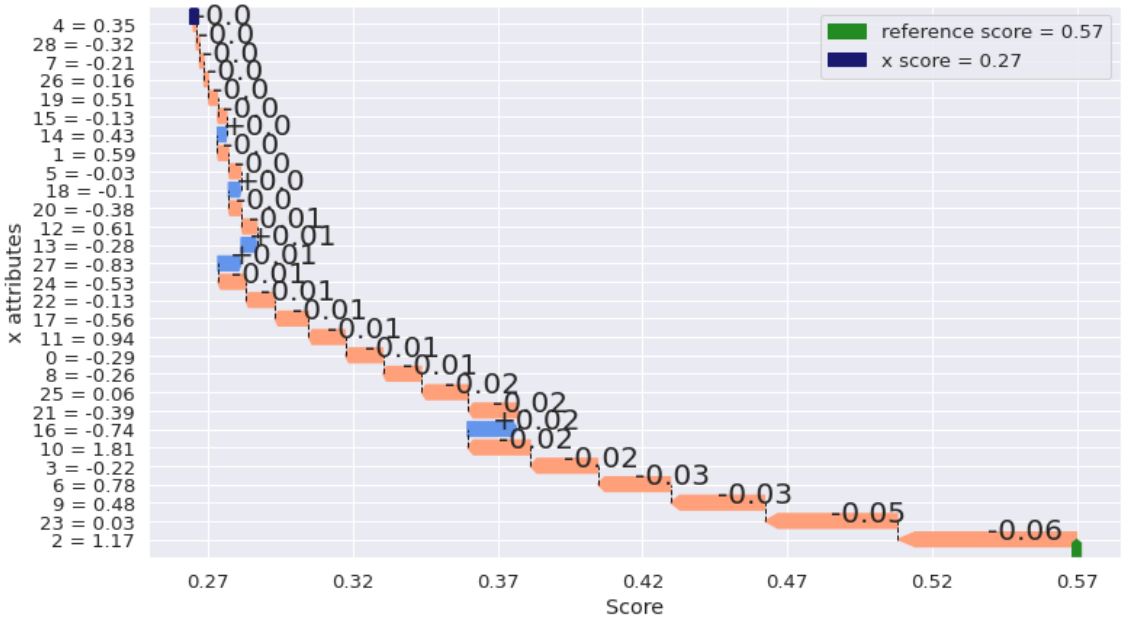


Figure 3.15: Example of the results provided by the local explanation service for the binary classification task (credit card fraud detection) - threshold 0.5

3.5.4.3 Multiclass classification case

In the case of multi-class, the choice of the reference population is more varied than in previous cases. We study the deviation between the score of the instance for the class for which this score is strongest and this same score of the reference population. The reference population can be all the instances predicted in another class that the instance of interest or in a class chosen by the user.

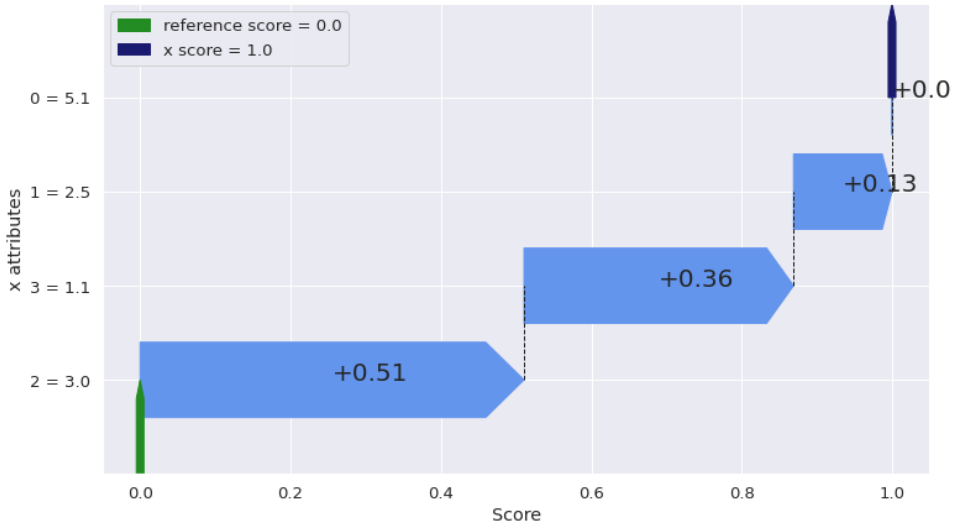
For instance, given a neural network noted f, an instance noted c for a three-dimensional classification, the output of *f* is given by *f(x)=(p1, p2, p3)*. If *p2* is the strongest for the instance of interest, we compute the Shapley values according *p2*.

The example below is given on Iris dataset, a popular benchmark dataset[10]. On this example, the reference population is given by the instances predicted in the first class ("multi_class_target_pop":0).

[10] https://gist.github.com/curran/a08a1080b88344b0c8a7

```
default_hparams_2={
    "path_reference":None
    , "method":None
    , "n_iter":10000
    , "savefig" : True
    , "fig_format":"png"
    , "no_show":False
    , "step":.1
    , "step_type":"sqrt"
    , "threshold":None
    , "multi_class_target_pop":0
}
```

Figure 3.16: Setup of the local explanation service for the multi-classification task (Iris dataset)



De

Figure 3.17: Example of the results provided by the local explanation service for the multi-classification task (Iris dataset) - "multi_class_target_pop":0

In the example below, the reference population is given by all the instances predicted in another class that the instance of interest (**"multi_class_target_pop":None**).

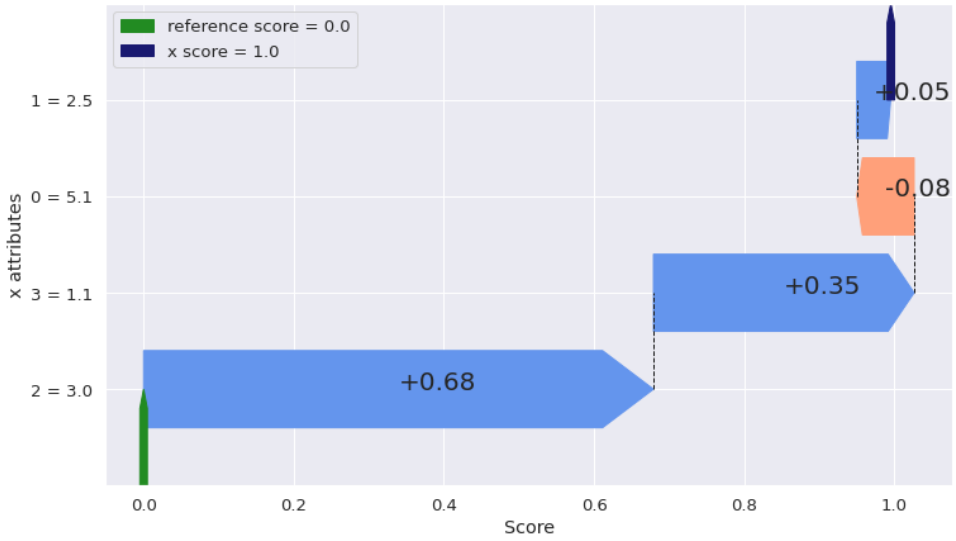
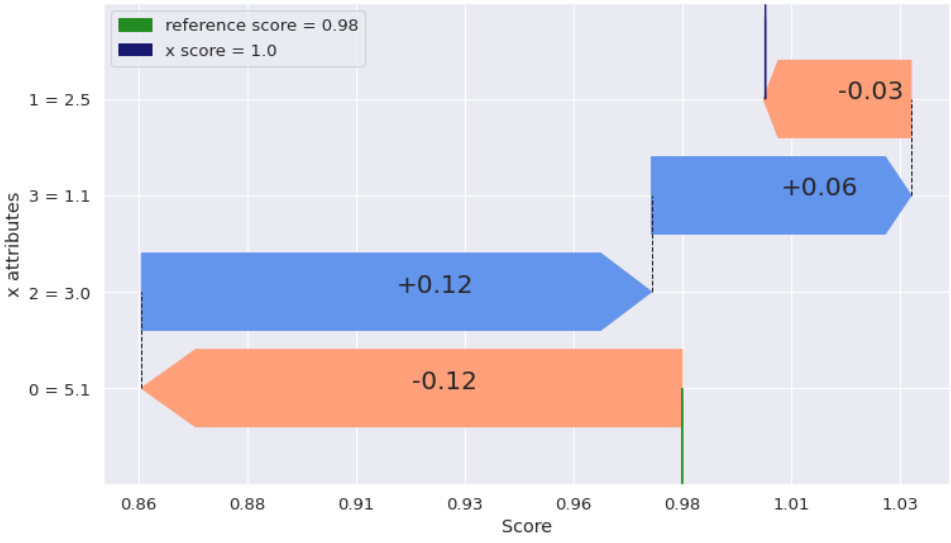


Figure 3.18: Example of the results provided by the local explanation service for the multi-classification task (Iris dataset) - **"multi_class_target_pop":None**

In the case where we choose as reference population the population that is in the same class as the instance of interest, then a warning message is raised and Shapley values are computed, like in the example below.



De

Figure 3.19: Example of the results provided by the local explanation service for the multi-classification task (Iris dataset) - reference population in the same class as the instance of interest

3.5.5 Reference

Aas, K., Jullum, M., & Løland, A. (2019). Explaining individual predictions when features are dependent: More accurate approximations to Shapley values.

Grah and Thouvenot (2020), A Projected Stochastic Gradient Algorithm for Estimating Shapley Value Applied in Attribute Importance.

Lundberg, S. M., & Lee, S.-I. A. (2017). A unified approach to interpreting model predictions.

Shapley. (1953). A value for n-person games.

Štrumbelj, E., & Kononenko, I. (2010). An Efficient Explanation of Individual Classifications using Game Theory.

Štrumbelj, E., & Kononenko, I. (2014). Explaining prediction models and individual predictions with feature contribution.

3.6 Hyperparameter Fine-tuning

A common need expressed by multiple partners has been a method for automatically finding the optimal hyperparameters for the other ML models.

The *Bayesian optimization* service developed in IoTwins is meant to support and allow clients to make hyperparameter fine-tuning. This service can be used to automatically improve the performance of the already existing services as well as future services that will be developed and added in the IoTwins context. This hyperparameter tuning service belongs to a family of approaches from the Artificial Intelligence area called **Bayesian Optimization** [11], a methodology based on building a surrogate for the objective and quantifying the uncertainty in the surrogate using Gaussian Process regression (a Bayesian ML technique). The goal is to obtain a good approximation of the target function with the minimum number of iterations; many literature works show that Bayesian optimization outperform random search and grid search approaches[12]. Bayesian optimization methods are competitive in finding optimal parameter configurations and good approximation of the target function for a given algorithm.

Why was this service developed? The IoTwins services are applicable in across a wide range of contexts, e.g., anomaly detection, time-series classification, RUL estimation, etc. To work best, these services need specific and not always easy to find configurations that usually involve experts' work. The *Bayesian optimization* aims to remove this gap and allow any clients to automatically individuate the best hyperparameters configuration in an acceptable time. The *Bayesian optimization* service is a wrapper of the Hyperopt python library[13] for serial and parallel optimization over awkward search spaces, which may include real-valued, discrete, and conditional dimensions.

Simply defined, Bayesian Optimization is a sequential design strategy for global optimization of black-box functions that does not assume any functional forms. It is employed to optimize expensive-to-evaluate functions. Bayesian approaches keep track of past evaluation results which they use to form a probabilistic model (the surrogate function) mapping hyperparameters to a probability of a score on the objective function: *P(score | hyperparameters)*. Bayesian model-based optimization can be described in an intuitive manner: choose the next input values to evaluate based on the past results to concentrate the search on more promising values. The end outcome is a reduction in the total number of search iterations compared to uninformed random or grid search methods.

3.6.1 Hyperparameter Fine-tuning - Details

Formulating an optimization problem requires four main parts, two of which are completely managed from the service:

- An *objective function*[mandatory]: takes in an input and returns a loss to minimize.

[11] Frazier PI. A tutorial on Bayesian optimization. arXiv preprint arXiv:1807.02811. 2018 Jul 8.
[12] Alibrahim, Hussain, and Simone A. Ludwig. "Hyperparameter optimization: Comparing genetic algorithm against grid search and bayesian optimization." *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2021.
Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams. "Practical bayesian optimization of machine learning algorithms." *Advances in neural information processing systems* 25 (2012).
[13] http://hyperopt.github.io/hyperopt/

- A *domain space*[mandatory]: the range of input values to evaluate.
- An *optimization algorithm* [default TPE]: the method used to construct the surrogate function and choose the next values to evaluate; TPE stands for Tree Parzen Estimator.
- An evaluation metric [not mandatory]: score, value pairs that the algorithm uses to build the model.

The objective function can be any function that returns a real value that we want to minimize, for instance the detection error of an anomaly detection model. In addition, a "saving model" function is needed as well, to decide what and how to log during the optimization process.

```
def objective(x, verbose=0):  # search the best 'x'
    """Objective function to minimize"""
    if verbose == 1:
        print("[objective] startig")
    # Create the polynomial object
    f = np.poly1d([1, -2, -28, 28, 12, -26, 100])

    # Return the value of the polynomial
    if verbose == 1:
        print("[objective] returning value")
    return f(x) * 0.05, {}, {}
```

Figure 3.20: Objective function code example

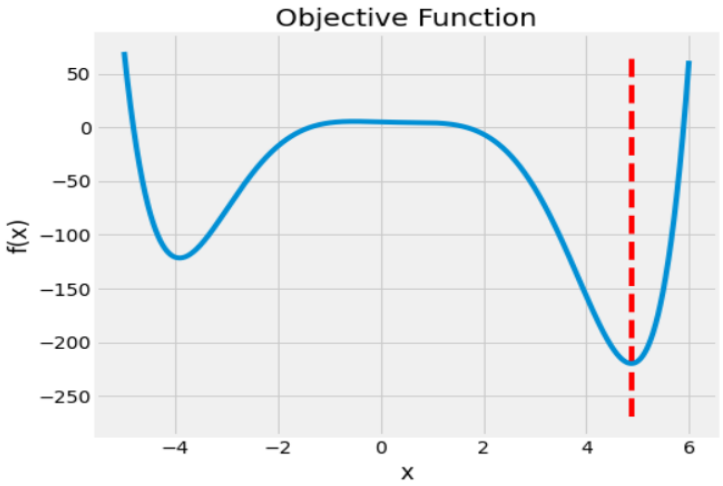


Figure 3.21: Objective function example

The domain space is the input values over which we want to search. This part may be the most difficult part to handle, however the Bayesian optimization service requires the same space domain format as defined by the Hyperopt library[14]. In practice, the domain space reflects the type of parameters we want to calibrate; for instance, whether we are dealing with a parameter with continuous values, or a discrete range, or again a categorical one. If no prior information is available a reasonably viable choice consists of choosing a uniform domain space.

[14] http://hyperopt.github.io/hyperopt/getting-started/search_spaces/

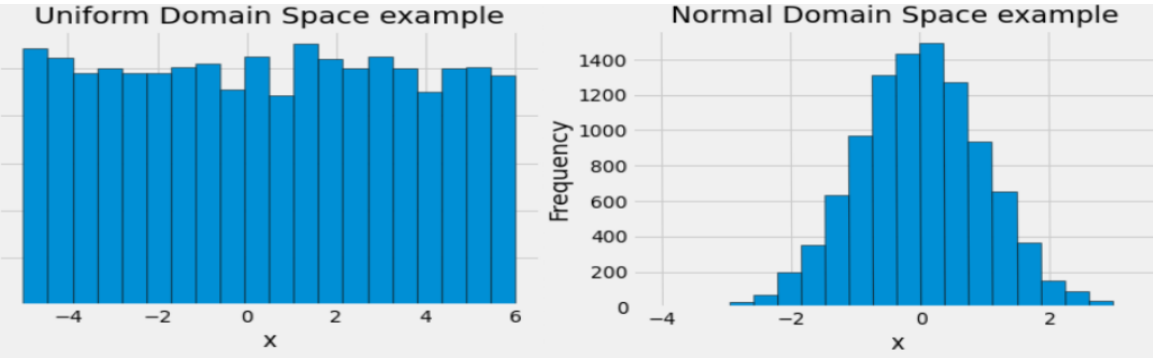


Figure 3.22: Domain space examples - Uniform and Normal

The optimization algorithm is a not mandatory parameter of the service which specifies the surrogate model employed. As default the Tree-structured Parzen Estimator model is used inside the service, but it is possible to change it via an appropriate parameter. The service supports three possible surrogate models: Random Search, Tree of Parzen Estimators (TPE) and Adaptive TPE. Once the problem is defined, the objective function can be minimized, thus finding the optimal configuration of the target ML model. In addition to the four elements specified above, the fine-tuning service also takes as input the maximum number of evaluations. The service will then proceed by exploring the possible hyperparameter configurations, until the maximum number of evaluations is reached, while doing so it builds a surrogate model aiming at learning the relationship between a set of hyperparameters value and their quality. This complex function is learned thanks to the points generated through the evaluation of the selected candidate configurations.

4. Simulation and fault modelling services

As part of task 3.4, various services have been developed and deployed for the simulation and modelling of faulty behavior within operating assets. These approaches represent key enablers towards the development of methods for predictive maintenance, an important requirement in the realization of digital twins and are completely complementary to the services described in the previous section(s).

4.1 Nessy2m solver

4.1.1 Overview

The Nessy2m program has a unique temporal approach applied to machining process modeling: milling, boring and turning operations. Nessy2m is a software dedicated to the analysis of dynamic phenomena in machining with prediction of geometric defects generated on machined surfaces: form, waviness and partly roughness.

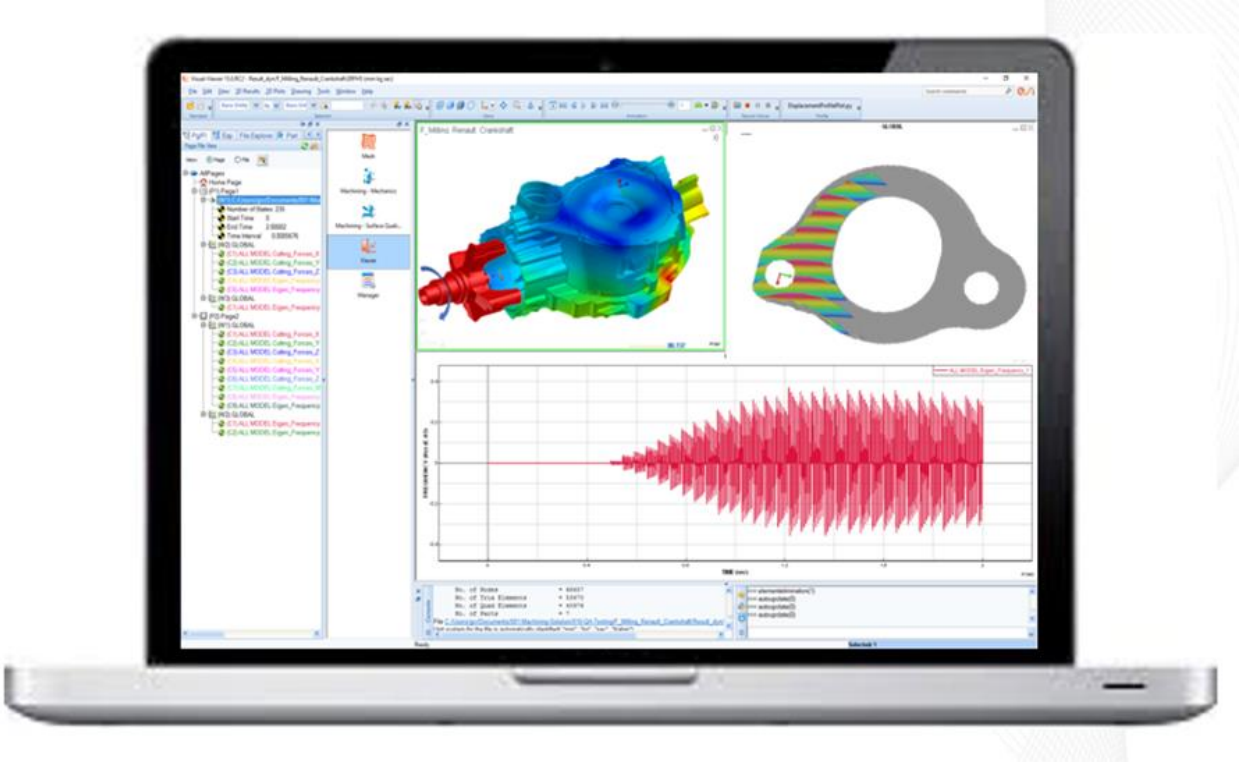


Figure 4.1: Nessy2m solver interface

4.1.2 Introduction

Nessy2m is based on an incremental time domain approach. At each instant it can handle workpiece deformation and tool vibration interacting through cutting forces. At each instant these cutting forces result from the contact between the workpiece and the tool which itself depends on the current geometry of the machined surface. This geometry evolves continuously due to matter removal by rakes faces of the tool.

The result of research works taking their roots in 1992 with a first thesis held in 1995 (Khaled Dekelbab) and followed by 3 other theses (Erwan Beauchesne, 1999, Audrey Marty, 2003, Stéphanie Cohen-Assouline, 2005). Nessy2m is currently developed by members of the DYSCO group of the PIMM laboratory.

4.1.3 Theory

During each time step, interferences between the volume swept by the rake faces of the teeth of the tool and the current machined surface are detected. This intersecting matter is then removed (see Figure 4.2 below) and no chip formation is simulated. The tool acts as a rubber.

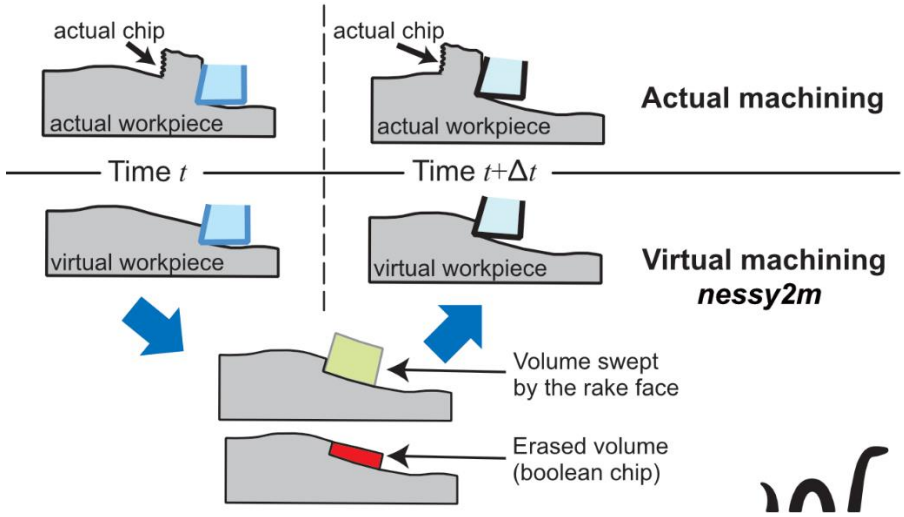


Figure 4.2: Nessy2m actual and virtual machining

The machined surface is thus updated at each time step, and cutting forces are applied between the part and the teeth in contact. These cutting forces are deduced from the quantity of matter removed and current cutting conditions. These forces are the main cause of deformation and vibration of the part or the tool.

The geometric models of the workpiece/tool follow the deformation/vibration of the workpiece/tool if they are flexible. This allows to define accurately their relative position, matter erasing and vibration history and thus the geometry of the final surface.

Figure 4.3 gives a summary of these ingredients.

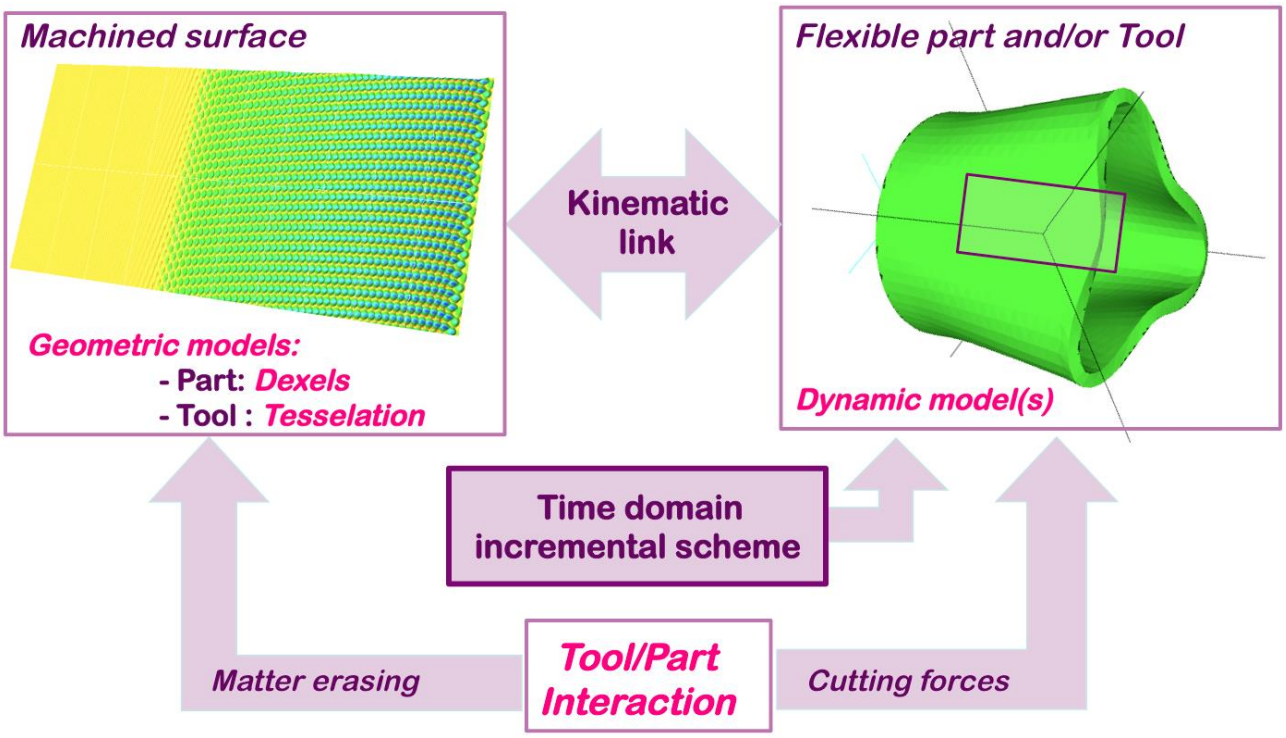


Figure 4.3: Nessy2m main components

4.1.4 Pre-processing data

The different ingredients of the simulation model are listed below. The first are mandatory, the others depend on what the user wants to take into account or what is necessary to be close to reality.

At least:

- the geometry of the workpiece before machining, (only the volume involved in the machining process is mandatory),
- the geometry of the teeth,
- the tool path,
- parameters of time domain incremental scheme,
- list of wanted output data, ...

If the user wants the evolution of cutting force, cutting power, or if they may induce deformations:

- cutting laws associated to each teeth/matter pair (the tool may have teeth of different shapes and the piece may be composed of several materials).

If the vibration of the part, or of the tool, may affect the geometry of the machined surface:

- a dynamic model of the workpiece,
- a dynamic model of the tool.

If clamping effect is present:

- static deformation of the part (that may evolve during machining).

4.1.5 Post-processing data

According to the requests of the user, the results are of two kinds: Time series (power, cutting forces, displacements, …) and meshing (machined surface).

A non-exhaustive list of the available output data:

- 3 components of the global force applied on the tool,
- 3 components of the global torque applied on the tool,
- average global cutting power,
- volume of erased matter per second during a fraction of tool revolution,
- velocity of the generalized degrees of freedom,
- acceleration of the generalized degrees of freedom,
- mesh of the machined surface,
- …

4.1.6 An example of inputs

Figure 4.4 is an example of a milling study. We can see the meshed workpiece, the active teeth of the tool, the trajectory of the tool and the machining zone.

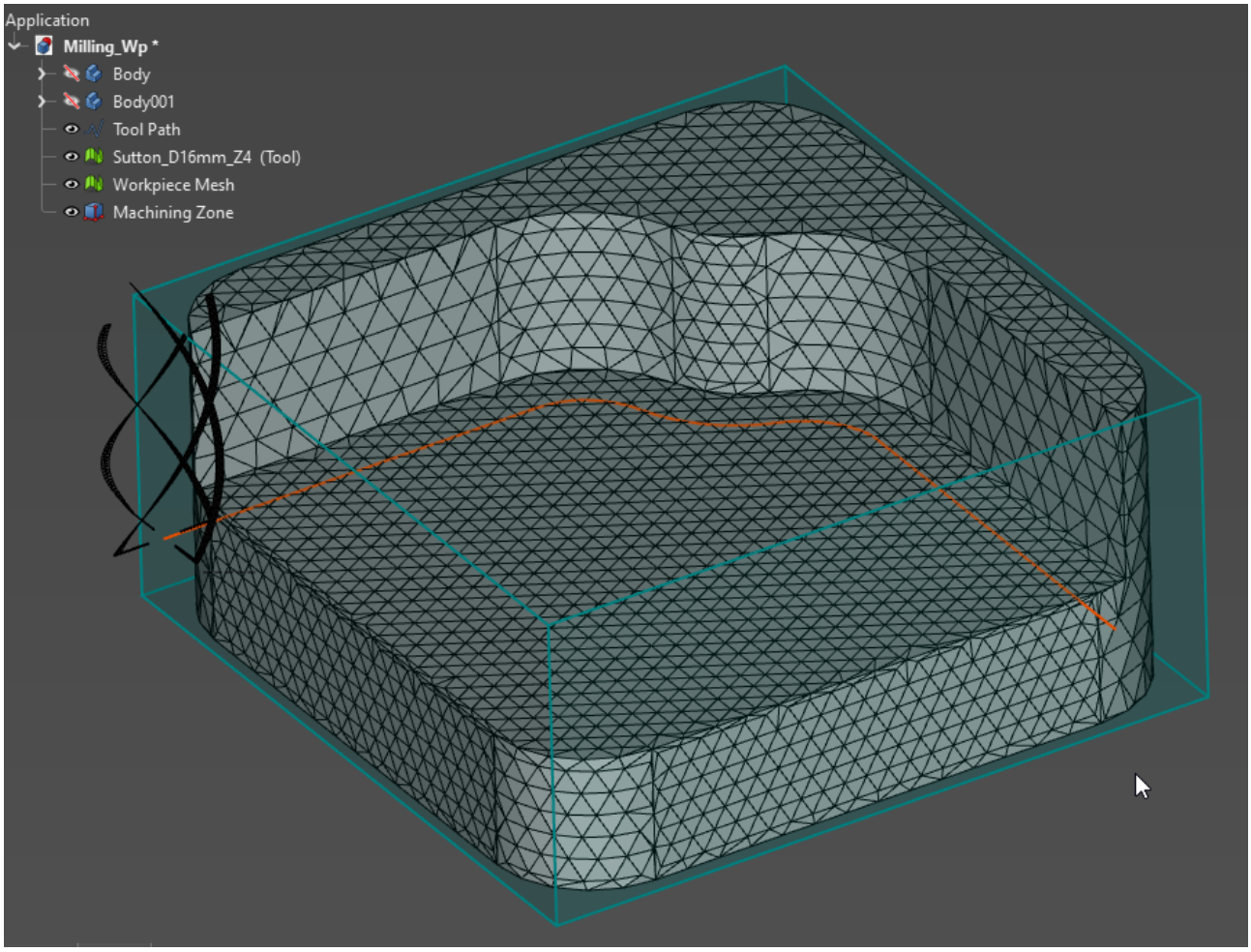


Figure 4.4: Nessy2m milling study example

These geometrical data are completed by the characteristics of the material, the cutting forces definition and the dynamic behavior of the workpiece in case of dynamic study.

4.1.7 Examples of outputs

An example of cutting forces plot during the milling:

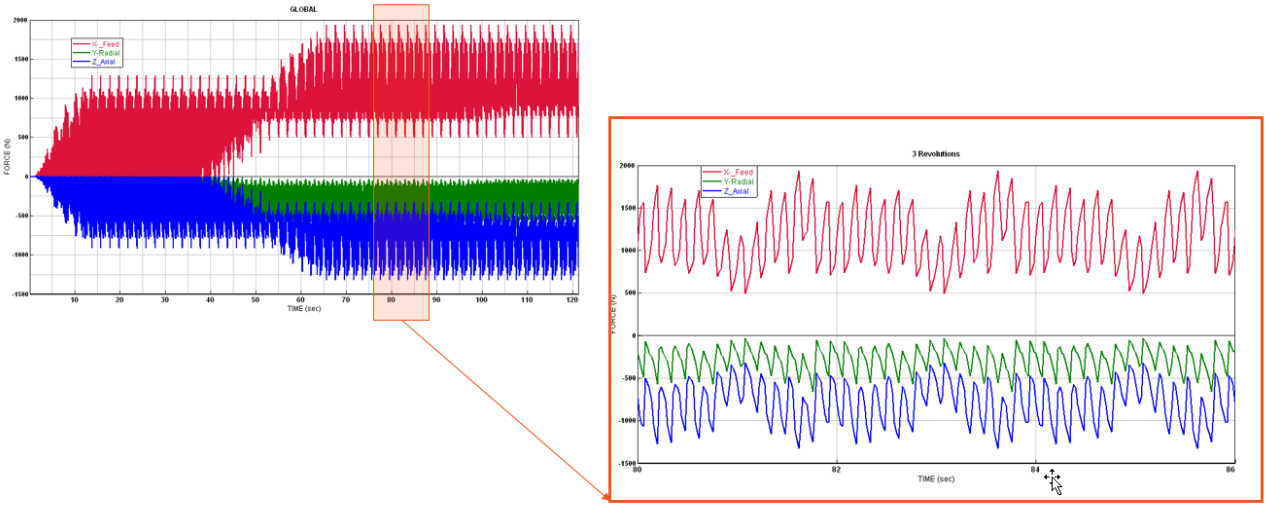


Figure 4.5: Nessy2m cutting forces plot during the milling

Example of computed machined surface and associated roughness computation:

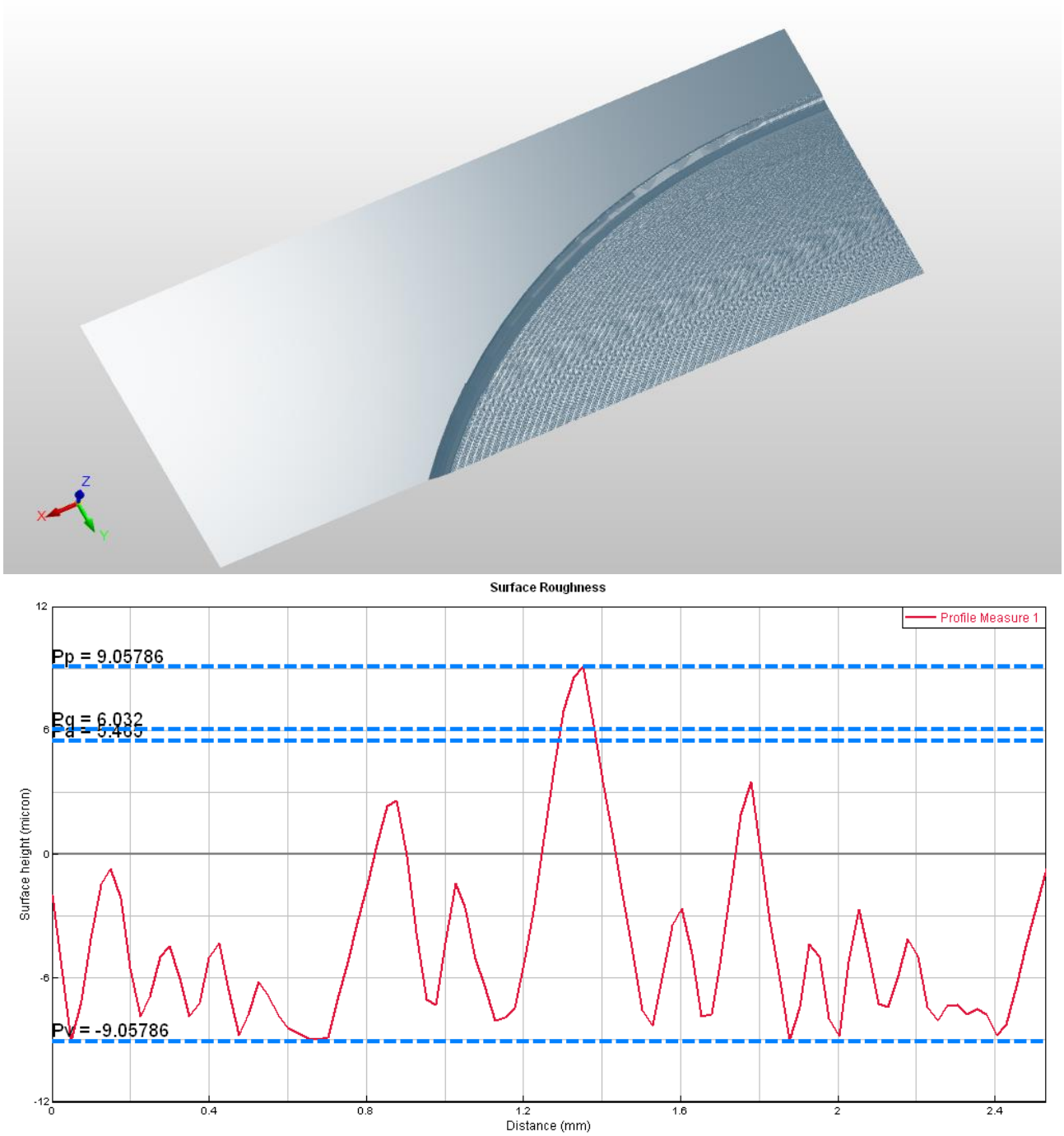


Figure 4.6: Nessy2m computed machined surface and associated roughness computation

An example of side milling effects on a flexible workpiece:

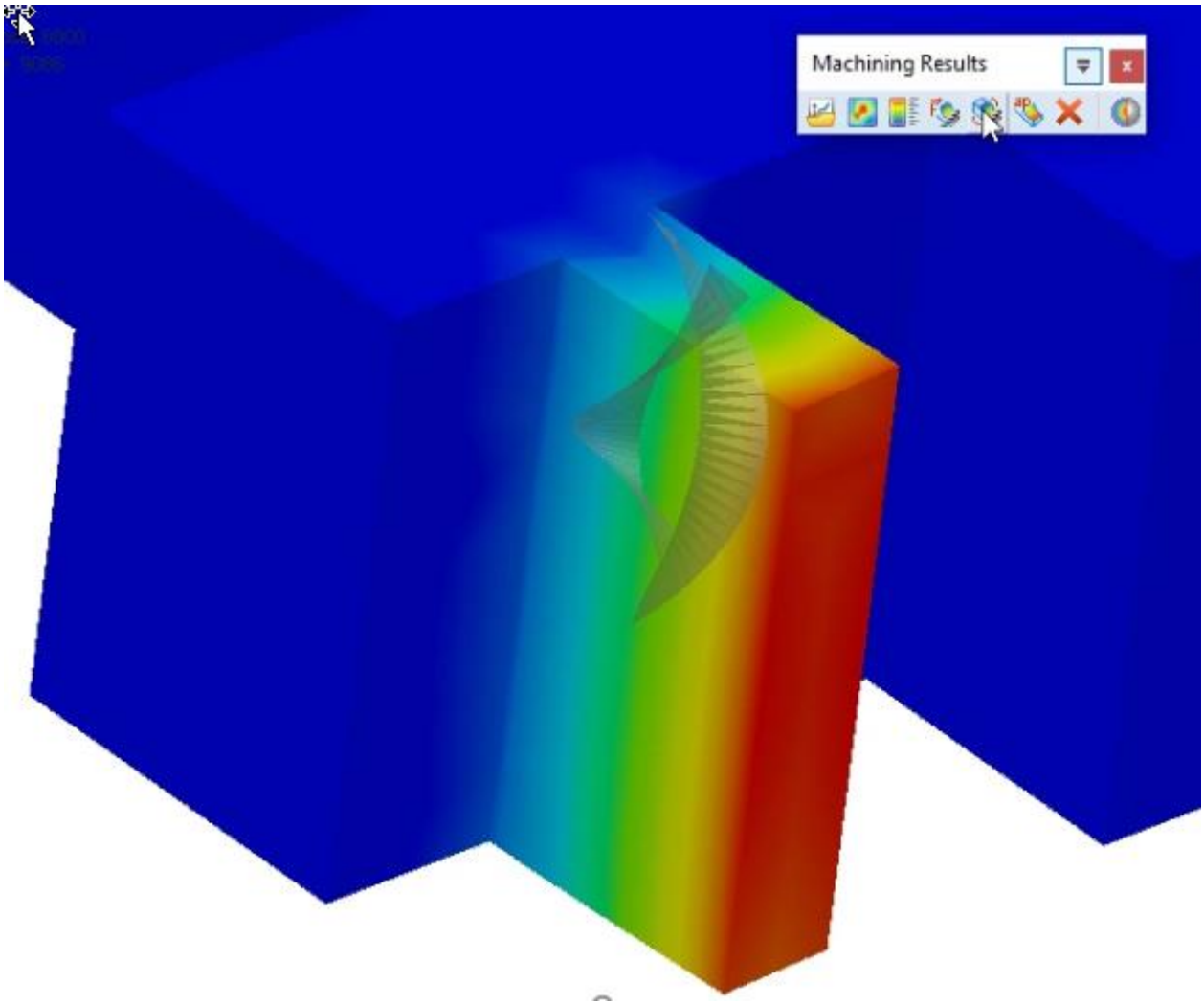


Figure 4.7: Side milling effects on a flexible workpiece

4.2 System simulation and fault data analytics services

4.2.1 Overview

System simulation is an appealing tool for applications requiring real-time (or near real-time) response, such as Digital Twins. Within the system simulation paradigm, the architecture and physical behavior of (cyber-)physical systems are represented using "lumped-parameter" models that capture important physical phenomena, while abstracting away detailed, computationally intensive behavior. The resulting simulation models run quickly (often faster than real-time) with a low computational and memory footprint, making them ideal candidates for deployment on resource-constrained hardware e.g. Edge devices.

A number of stakeholders within the system simulation have promoted the development of open modelling (Modelica[15]) and co-simulation (FMI) standards that promote the interoperability of models between different modelling tools and simulation environments. We have exploited these standards within the services that we have developed in order to ensure the maximum applicability of the provided tools.

[15] https://modelica.org/

4.2.2 System simulation for generation of synthetic fault data

A motivation for using system simulation for representation of asset fault modes is to compensate for a lack of available operational data containing faulty behavior. In this case, fault modes are modelled explicitly within the simulation model and can be thus used to generate synthetic fault data. As an example of how this approach can be implemented, the system simulation software SimulationX provides the System Reliability Analysis tool. Within the SimulationX modelling environment, the tool allows the user to augment a system model representing the nominal (non-faulty) operation of a physical asset (e.g. a model of the key components of a wind turbine) with different fault modes for the various physical domains represented in the model, e.g. mechanical, fluid, hydraulic, electrical. These fault modes may take the form of parametric faults, representing degradation in individual components, or connection faults, representing losses or leakages between components. The degree, or intensity, of the fault is parameterized, allowing for representation of any scenario between nominal (fault intensity=0) and full degradation of a component (fault intensity=1). An example of such a fault-augmented model can be seen in Figure 4.8.

A workflow within the tool allows the user to generate various datasets of synthetic faults data that can be used for the feature extraction for fault sensitivity and diagnosability analysis purposes. To facilitate these tasks, a library of model components is provided for feature extraction, e.g. via statistical measures.

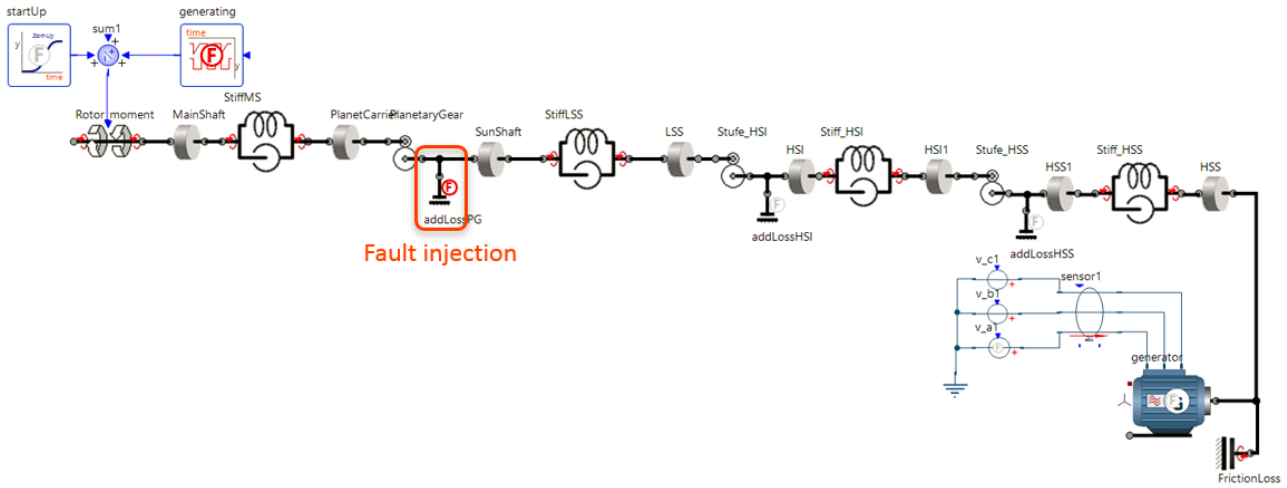


Figure 4.8: Fault-augmented wind turbine model in SimulationX

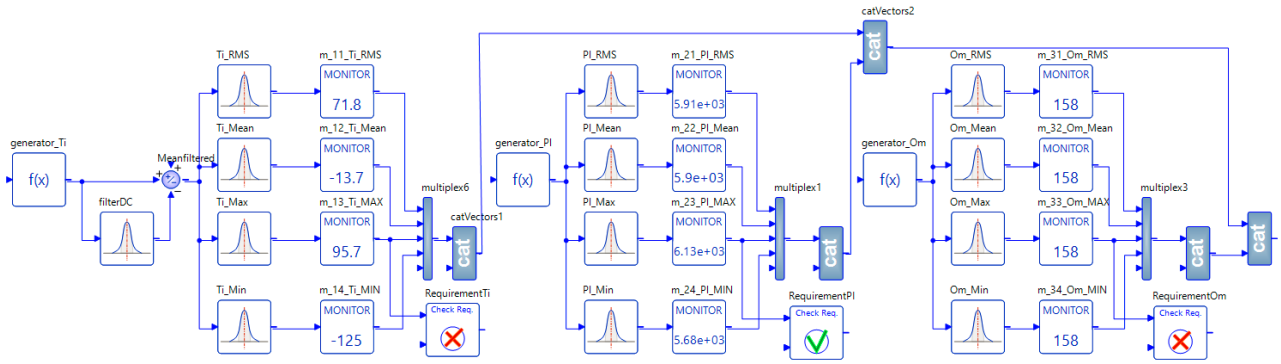


Figure 4.9: Feature extraction model in SimulationX

To facilitate the deployment of the services on the IoTwins platform, and also to ensure that they are agnostic with respect to the simulation tool, it was decided use models that conform to the Functional Mockup

Interface[16], i.e. functional mockup units (FMUs). In the examples presented in the following sections, the FMUs were generated using SimulationX. However, any FMUs that utilize the same convention for representing fault modes (e.g. normalization of fault intensity between 0 and 1) can be used. The FMI specification describes two types of FMUs: Model Exchange and Co-simulation. Both types are supported by the services described in this section.

On the next picture two Functional Mock-up Units (FMU) with results are shown generated from the wind turbine model above.

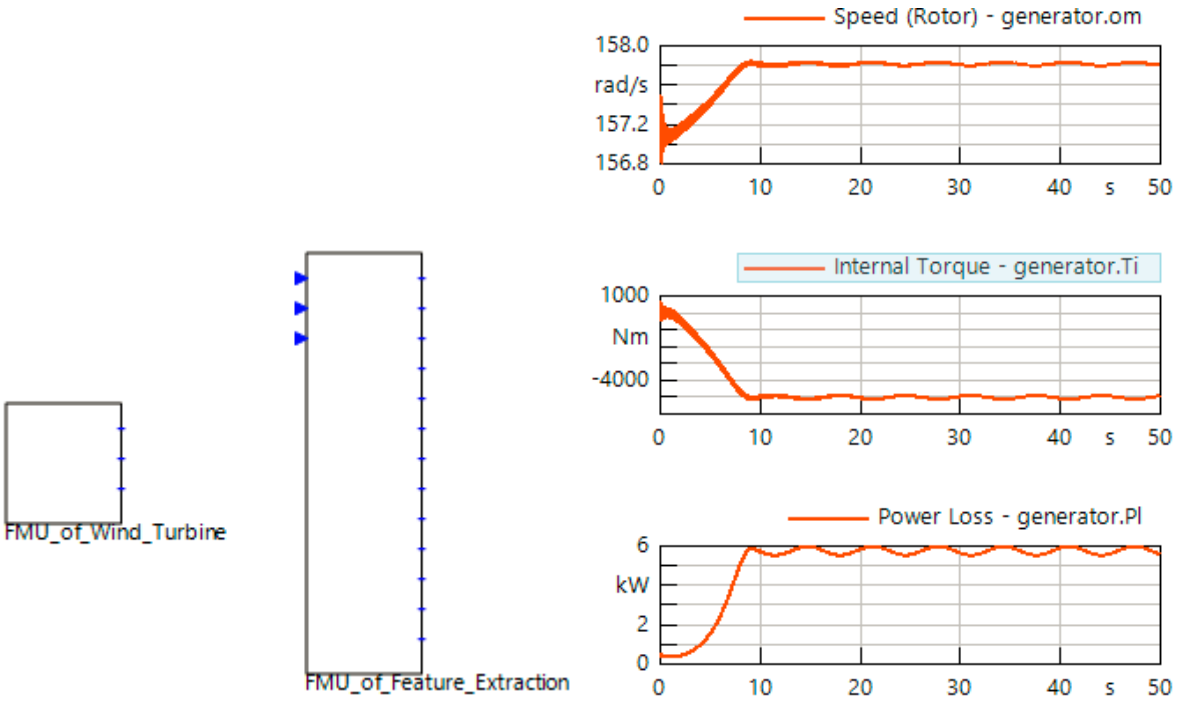


Figure 4.10: FMUs of wind turbine and feature extraction to analyze and extract features from generator rotor speed, torque and power losses.

4.2.3 FMU-based services for sensitivity and diagnosability analysis

The various FMU-based services described here are managed via a browser-based UI (web application) hosted within a Docker container that is deployable via the IoTwins platform. Screenshots of the UI demonstrate the various workflows or "options" that the user can follow, depending on the desired analysis type.

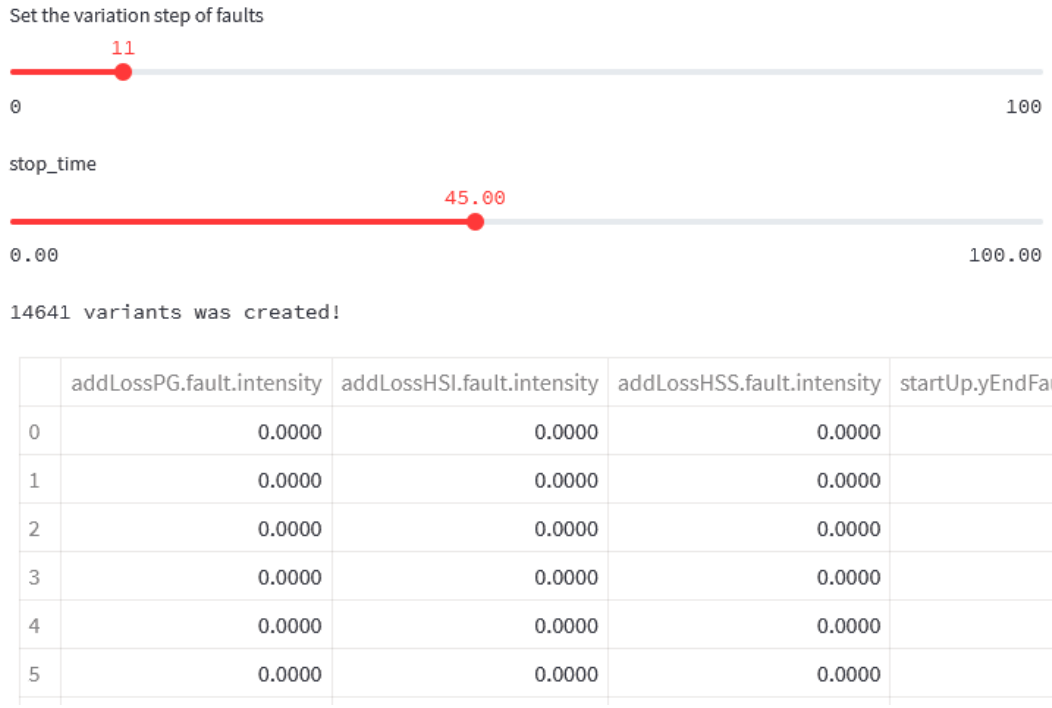
Fault data collected using FMUs can be used to generate Machine Learning (ML) classifiers for fault sensitivity or fault diagnosability purpose. In the case of the fault sensitivity the classifiers are used for analytics tasks to predict which faults (including their combination and various intensities) are crucial for system requirements and behavior. On the other hand, the fault diagnosability approach focuses on fault class systematization and selection of the system outputs related for the fault detection/prediction. Based on these two purposes, the implementation of the FMU-based services using ML approach is described below.

The FMU-based services presented were implemented in python, making use of freely-available libraries for managing FMUs (FMPy) and creating ML-based models (Scikitlearn). The resulting functionality is unified in the UI, but divided into four separate and independent options that are described below.

[16] https://fmi-standard.org/

4.2.3.1 Option 1: "Generate Fault Data"

In this part the time series data are generated by FMUs simulating variants for selected faults. To start the variants simulation, a FMU file must be selected and its active faults defined, outputs (and inputs if needed) as well as the step number of the selected faults and the simulation time as shown in the figure below. After the variants simulation has been performed, all results can be saved as time series data in a CSV file that the user can subsequently download.



	addLossPG.fault.intensity	addLossHSI.fault.intensity	addLossHSS.fault.intensity	startUp.yEndFault.intensity
0	0.0000	0.0000	0.0000	0.0000
1	0.0000	0.0000	0.0000	0.1000
2	0.0000	0.0000	0.0000	0.2000
3	0.0000	0.0000	0.0000	0.3000
4	0.0000	0.0000	0.0000	0.4000
5	0.0000	0.0000	0.0000	0.5000

Figure 4.11: Parameter settings for variants simulation to generate fault data.

4.2.3.2 Option 2: "Extract Features"

Option 2 deals with the feature extraction from the time series data generated in option 1 (or a dataset provided by user themself e.g., from measurements data). To extract features, the feature extraction FMU must be loaded and parametrized by the selection of inputs, outputs and adjustment of the extraction conditions if needed as well as the simulation time as show on the picture below.

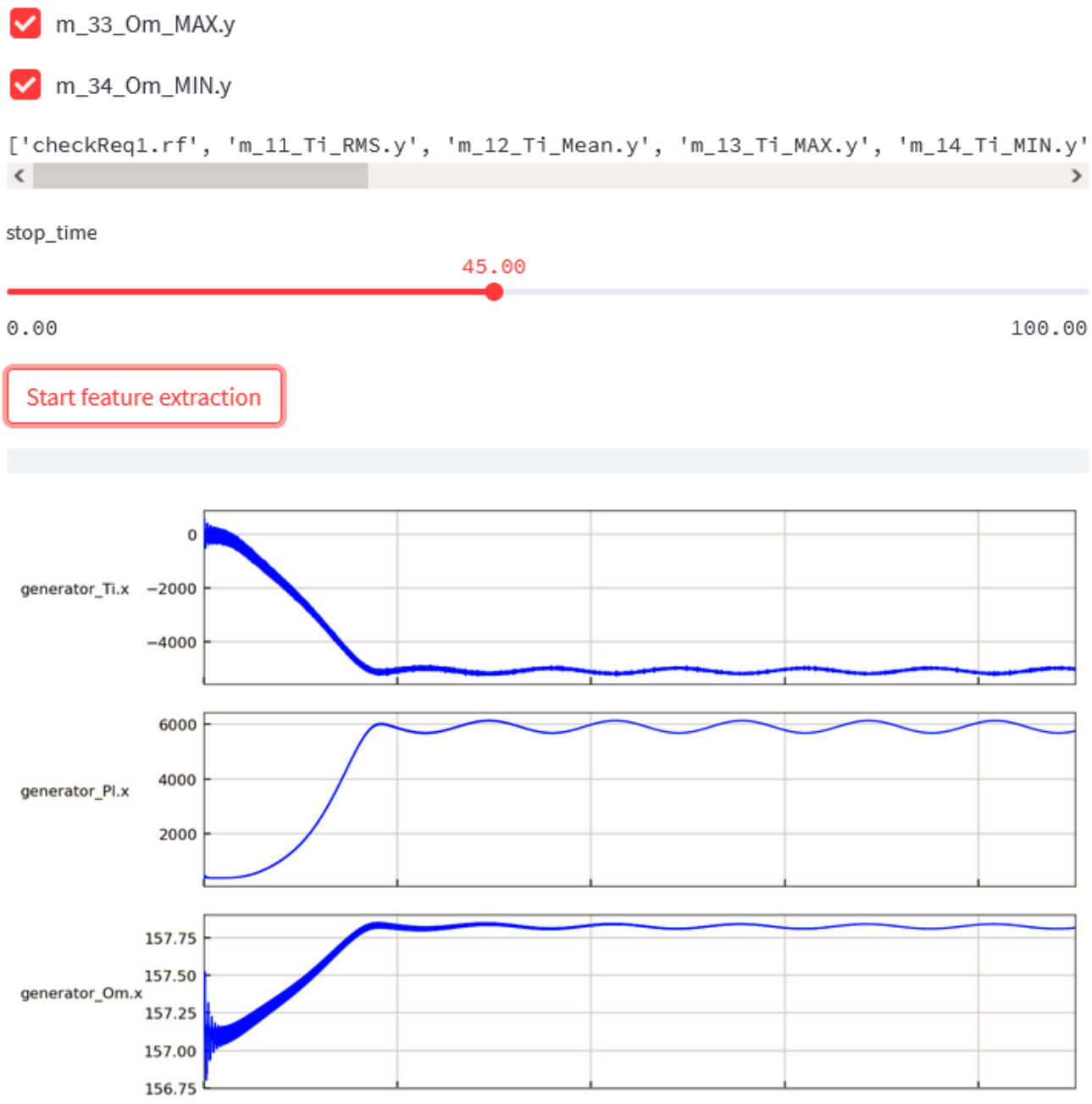


Figure 4.12: Parameter settings and input signals for Feature Extraction FMU.

After the completion of the feature extraction, all results with extracted features and faults can be saved as a discrete fault data set in a CSV file, as shown in Figure 4.13.

addLossPG.fi	startUp.yEnd	m_11_Ti_RM	m_12_Ti_Me	m_13_Ti_MA	m_14_Ti_MI	m_21_Pl_RM	m_22_Pl_Me	m_23_Pl_MA	m_24_Pl_MI	m_31_Om_R	m_32_Om_M	m_33_Om_M	m_34_Om_M	checkReq1.rf
0	0	74.5013348	-6.94115316	147.898303	-189.673094	5883.82047	5881.55914	6133.1237	5680.59678	157.824627	157.824627	157.842581	157.809765	0
0	0.1	74.3999898	-7.07775732	158.611748	-181.585794	6462.32578	6460.03252	6725.47615	6247.90127	157.863879	157.863879	157.881934	157.848976	1
0	0.2	74.3629853	-7.07108593	144.061145	-184.833533	7072.31898	7069.99408	7349.54255	6846.51772	157.903443	157.903442	157.921706	157.888392	1
0	0.3	74.4423513	-7.03411325	147.080058	-176.501377	7714.32784	7711.97125	8005.87075	7476.95595	157.943339	157.943338	157.961752	157.928245	1
0	0.4	74.4684928	-7.01742729	157.328433	-170.003239	8388.92328	8386.5347	8695.05015	8139.77439	157.983591	157.98359	158.002121	157.968264	1
0	0.5	74.4581698	-6.88387376	162.260662	-173.887258	9096.71757	9094.29635	9417.69998	8835.55762	158.024221	158.024221	158.042925	158.008821	1
0	0.6	74.5219548	-7.04958457	167.346263	-175.144339	9838.37228	9835.91765	10174.523	9564.95558	158.065257	158.065256	158.084163	158.049743	1
0	0.7	74.5636768	-6.97504454	141.282763	-188.262925	10614.6035	10612.1144	10966.2474	10328.6679	158.106723	158.106723	158.125859	158.090987	1
0	0.8	74.3771729	-6.99849877	162.406639	-179.759808	11426.1761	11423.6512	11793.6712	11127.4438	158.148649	158.148649	158.167908	158.132791	1
0	0.9	74.3885511	-7.03624808	138.43855	-181.325647	12273.9254	12271.3634	12657.6402	11962.0801	158.191065	158.191064	158.210524	158.175021	1
0	1	74.4693458	-7.06894818	171.755808	-183.779326	13158.7409	13156.14	13559.0955	12833.461	158.234002	158.234001	158.253723	158.217777	1
0.1	0	74.4646933	-7.01990932	176.344347	-181.263445	5869.47052	5867.21005	6118.41597	5666.5345	157.823629	157.823629	157.841549	157.808792	0
0.1	0.1	74.5005346	-7.07111651	160.47458	-185.05887	6447.17242	6444.88001	6709.96369	6233.04663	157.862874	157.862873	157.880985	157.847922	1
0.1	0.2	74.5005572	-6.97903989	158.35487	-203.491002	7056.35231	7054.02823	7333.20964	6830.85284	157.902429	157.902428	157.920629	157.887415	1
0.1	0.3	74.4604858	-6.91900468	148.793398	-198.50031	7697.5342	7695.17844	7988.70041	7460.46815	157.942316	157.942316	157.960789	157.927204	1
0.1	0.4	74.4746438	-6.9660771	160.376996	-187.847986	8371.28353	8368.89578	8677.0244	8122.44586	157.982558	157.982557	158.001093	157.967216	1
0.1	0.5	74.3911304	-7.05608972	145.274866	-188.420703	9078.21742	9075.79709	9398.81324	8817.37458	158.023178	158.023178	158.041862	158.007809	1
0.1	0.6	74.4617666	-6.98350686	140.810702	-173.456881	9818.99495	9816.54124	10154.7466	9545.90388	158.064203	158.064202	158.083035	158.048664	1
0.1	0.7	74.4009377	-7.00090205	158.082762	-189.877038	10594.3269	10591.8387	10945.5649	10308.726	158.105658	158.105657	158.12472	158.089897	1
0.1	0.8	74.5255821	-7.09074976	155.647656	-179.246924	11404.9828	11402.4589	11772.0569	11106.5844	158.147571	158.147571	158.166946	158.131542	1
0.1	0.9	74.4808736	-7.07212348	167.860369	-186.053604	12251.7895	12249.2284	12635.077	11940.2909	158.189974	158.189973	158.209441	158.173925	1
0.1	1	74.4441896	-7.01606141	169.860801	-189.49524	13135.6388	13133.0388	13535.5541	12810.718	158.232897	158.232896	158.252733	158.216681	1
0.2	0	74.4549714	-7.00347633	144.10818	-180.023268	5855.13891	5852.87927	6103.73145	5652.49184	157.822632	157.822632	157.840569	157.807865	0

Figure 4.13: Fault data with extracted features.

4.2.3.3 Option 3: "Generate Classifiers"

To enable fault sensitivity and fault diagnosability analyses, the following ML-algorithms of Python/Scikit-learn were implemented in option 3:

- Decision Tree
- Random Forest
- Ada Boost
- Support Vector Machine
- Multi-layer Perceptron
- Gaussian Naïve Bayes
- Linear Discriminant Analysis
- Logistic Regression

All (hyper)parameters for fault data and each ML-classifier can be set in the UI, as depicted in the next Figure 4.14 for the Decision Tree classifier.

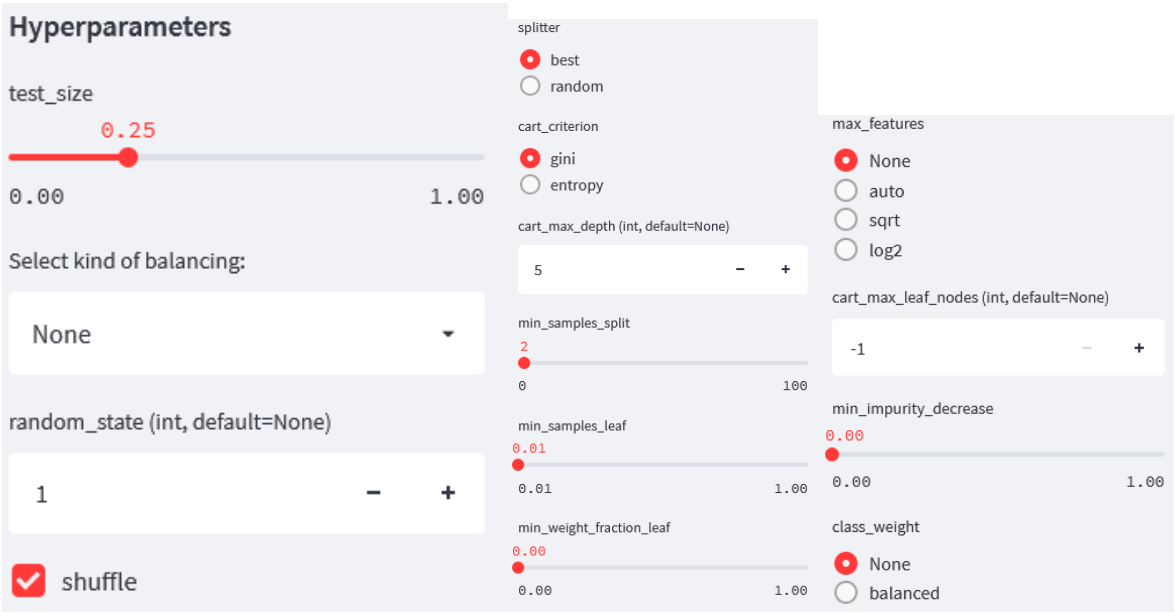


Figure 4.14: Parameter settings for fault data and ML-algorithm.

After the algorithm selection and parametrization, the classifiers can be generated and compared e.g., through accuracies and cross value scores.

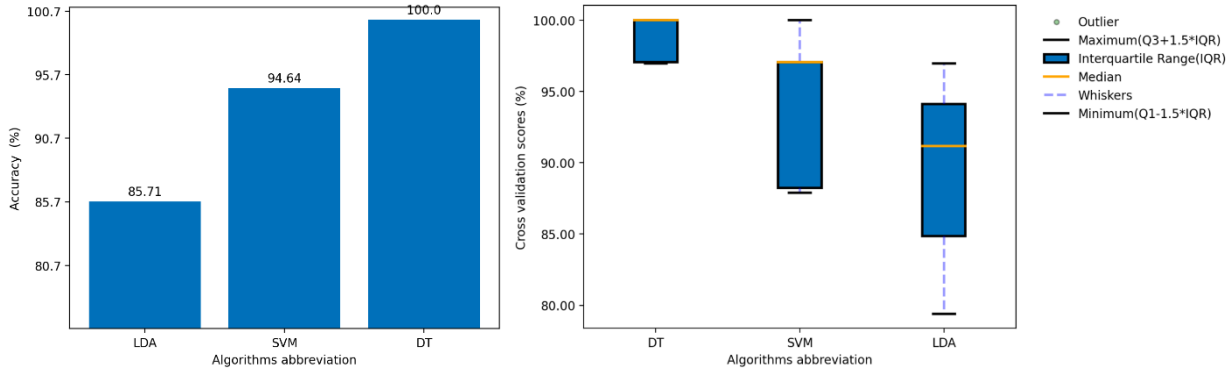


Figure 4.15: Accuracy comparison and cross value scores of generated classifiers.

As a result, for each classifier the confusion matrix, main metrics (precision, recall and f1-score) as well as the feature importance (for Decision Tree, Random Forest and Ada Boost classifiers) and Decision tree map (for Decision Tree classifier) are created as shown below.

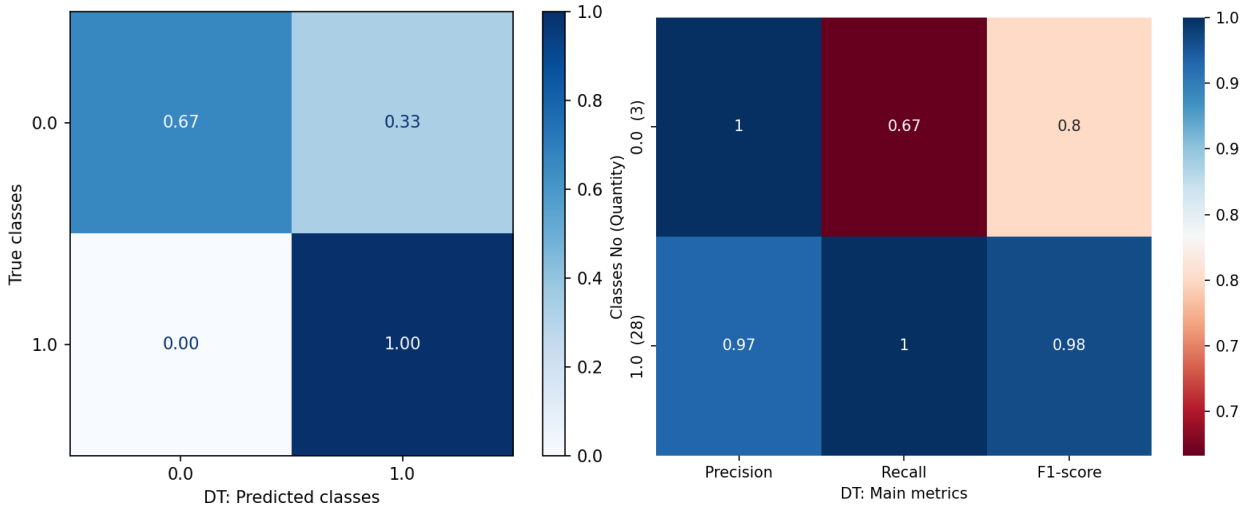


Figure 4.16: Confusion matrix and main metrics of generated Decision Tree classifier.

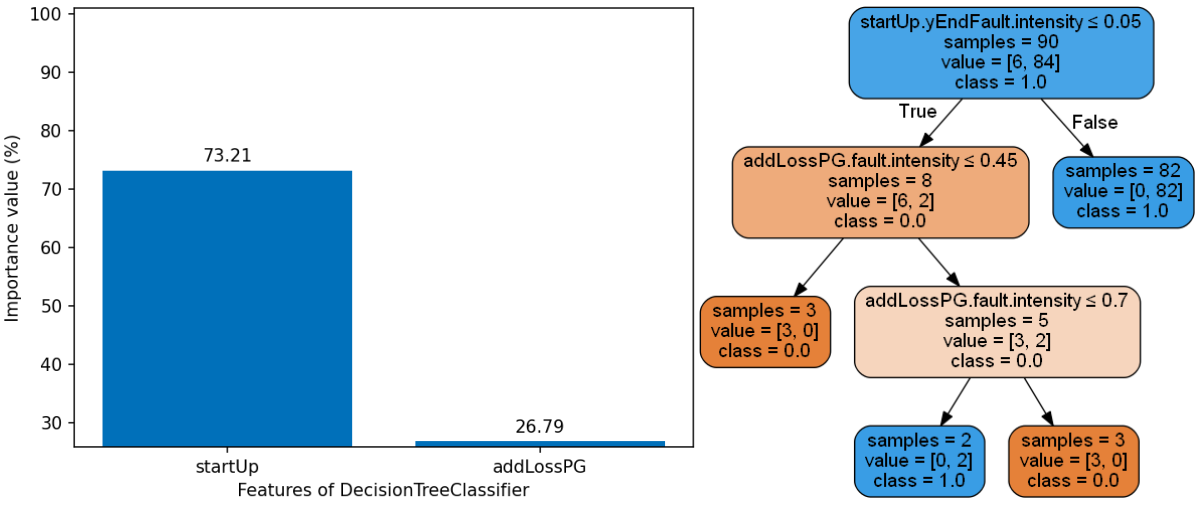


Figure 4.17: Feature importance and decision tree map of generated Decision Tree classifier.

Furthermore, option 3 can handle imbalanced fault data and allows the user to create fault classes for diagnosability purposes. All classifiers generated in this option can be saved as SAV-files for easy deployment, e.g., as part of a dataflow for predictive maintenance.

4.2.3.4 Option 4: "Reuse Classifiers"

A classifier saved as SAV-files can be used in option 4 "Reuse Classifiers" for other fault data with different fault intensities (or measurements data) to investigate the reliability and the quality of the classifier. For investigation, the confusion matrix and main metrics can be created as depicted below. As mentioned in option 3, the feature importance (for Decision Tree, Random Forest and Ada Boost classifiers) and Decision tree map (for Decision Tree classifier) can also be shown and saved as pictures in PNG-format.

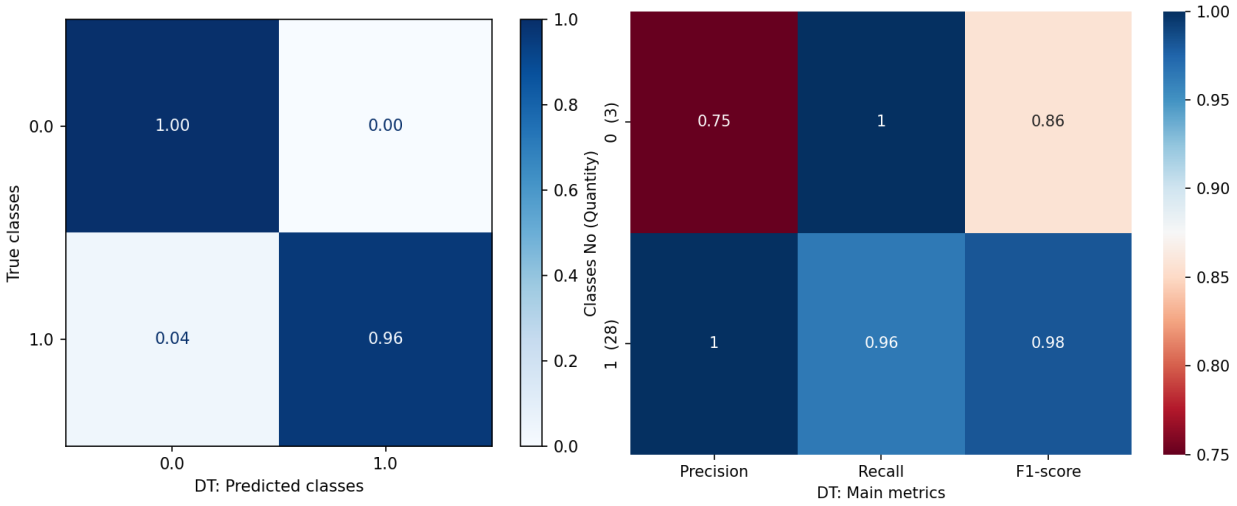


Figure 4.18: Confusion matrix and main metrics of reused Decision Tree classifier.

5. Changes upon feedback from testbeds

After the completion of D3.2, testbed partners started to use the first version of the offered services, either through direct access via the IoTwins platform or by adopting them in their already existing frameworks. This allowed the final users of the services to provide extremely useful feedback to improve the services and prepare the final version, described in this deliverable as well in D3.4 and (partially) in D3.5.

One aspect that was identified by all partners is the non-negligible effort necessary to choose the right service from the catalogue and the subsequent tailoring phase to make it more suitable to the specific testbed. On one hand, such difficulty was to be expected given the heterogeneous nature of the tasks to be addressed and the unavoidable effort needed to bridge the gap between AI practitioners and domain experts (the testbed partners). However, such difficulty can be (and has been) mitigated with a series of corrective actions to ease up the follow-up work of all involver partners. The following actions were performed:

- Identification of a few, selected AI experts to provide direct and extensive support to the testbed partners for the selection of the best services and their adaptation for the specific testbed.
- Improved and extended documentation, with the additional creation of more exhaustive tutorials and guidelines (see for example Sec. 5 of the current document or the tutorial on the Bayesian optimization service: https://gitlab.hpc.cineca.it/iotwins/ai-services/-/tree/master/hyperparameters_tuning_services/examples_tutorials/).
- Seminars and workshops were held to transfer knowledge from AI-experts to domain experts.

In some cases, the testbeds identified how the first version of the provided services was not sufficiently accurate, for example for the Remaining Useful Life estimation (TB04). In this case, in collaboration with the relevant partner (GCL), the service was improved and changed radically, adopting an entirely different Deep Learning model. In other situations, the changes were less drastic but nevertheless impactful. For instance, in the large-scale datacentre facility (TB06) at the beginning of the project labelled data was very scarce, and this prevented the usage of fully supervised anomaly detection services (anomaly detection being among the desired tasks for the TB06). Thus, the initial focus of WP3, which started the development of the AI services following testbeds' desiderata, was towards the development and deployment of unsupervised and semi-supervised models (see the autoencoder neural network described in D3.2). During the project the monitoring infrastructure employed in TB06 enabled the collection of information associated to the time points and describing the state of the target system – these are the *labels* of ML terminology, whose presence allows the adoption of supervised models. As supervised models tend to perform better (due the exploitation of labels) upon receiving the feedback from the TB06 partners, we developed a new version of the anomaly detection model, again based on a deep architecture composed of autoencoder networks but this time fully supervised, obtaining a marked performance increase (this extension will be fully described in D5.4).

6. Overview of ML dedicated to Predictive Maintenance

In this Section, we provide our feedback about the use of ML model dedicated to predictive maintenance. Predictive maintenance techniques are designed to help determine the condition of in-service equipment in order to estimate when maintenance should be performed. This approach promises cost savings over routine or time-based preventive maintenance because tasks are performed only when warranted. Thus, it is regarded as condition-based maintenance carried out as suggested by estimations of the degradation state of an item. The evolution of modern techniques (e.g. Internet of things, sensing technology and artificial intelligence) reflects a transition of maintenance strategies from Reactive Maintenance (RM) to Preventive Maintenance (PM) to Predictive Maintenance (PdM). RM is only executed to restore the operating state of the equipment after failure occurs, and thus tends to cause serious lag and results in high reactive repair costs. PM is carried out according to a planned schedule based on time or process iterations to prevent breakdown, and thus may perform unnecessary maintenance and result in high prevention costs. In order to achieve the best trade-off between the two, PdM is performed based on an online estimate of the "health" and can achieve timely pre-failure interventions. PdM allows the maintenance frequency to be as low as possible to prevent unplanned RM, without incurring costs associated with doing too much PM. The main promise of predictive maintenance is to allow convenient scheduling of corrective maintenance, and to prevent unexpected equipment failures. Predictive maintenance differs from preventive maintenance because it relies on the actual condition of equipment, rather than average or expected life statistics, to predict when maintenance will be required. Predictive maintenance is primarily focused on physical assets. Physical assets can be categorized in the following way:

- Manufacturing assets: these include production line and assembly machinery used to create a product.
- Field-level assets: these can further be sub-categorized: consumer appliances, vending machines, connected transportation, heavy equipment machinery, commercial energy generation, networks and Buildings: property, real estate, universities, stadiums, and corporate offices

The common denominator for these assets is that they all have the capability to collect a lot of information. Manufacturers increasingly collect big data from Internet of Things (IoT) sensors in their factories and products and using different algorithms for the collected data to detect warning signs of expensive failures before they occur. Typical data includes measurement data, log data and failure data. ML model can be used to perform PdM, with mainly focusing on three tasks: classification models to identify (early) failure, anomalous behaviour detection and regression models to predict Remaining Useful Life.

6.1 Dataset involved typologies

Different types of datasets can be encountered in Predictive Maintenance. We describe several common types of datasets, but this list is not meant to be exhaustive. In the first case described here, we have access to a fleet of several machines. A machine has a certain number of sensors whose value is recorded over time, at a regular interval. Moreover, at each measurement, the state of the machine is reported (normal operation or failure). The objective is to predict at each time step whether the machine has a failure or is in a normal state. A more sophisticated version of this dataset details which component of the machine has a failure. Another variation of this dataset replaces the failure column by a Remaining Useful Life (RUL) column. RUL is the length of time a machine is likely to operate before it requires repair or replacement. The objective is to predict at each time step the RUL of each machine. In the second case, at each time step, the values are recorded by several sensors over a period of time. In other words, at each time step, we have a multivariate time series. Note that an important feature of predictive maintenance datasets is the multivariate nature of the time series. With the fast-growing industrial IoT market, industrial assets are monitored by many sensors. The datasets contain several assets with multivariate time series. The number of sensors (and thus, of time series) can be large, which suggests to use variable selection. Variable selection enables to identify the features of interest and to speed up the calculation time. Sensors may also fail or give abnormal values.

6.2 Illustration on simulated data: binary classification

The simulation is inspired by Shadgriffin [17]. The simulated dataset contains 421 machines monitored during 721 days with 8 sensors. Each machine has one equipment failure during its lifecycle. The equipment failure is denoted by 1 (positive class), while the normal state is denoted by 0 (negative class). Some additional information is given for each machine: the region in which the machine resides, the vendor, the manufacturer, the type of machine and the age of the machine in days. Before feeding the time series to a machine learning model, we pre-process the dataset. Hereafter, we discuss some pre-processing steps specific to predictive maintenance. Note that once the pre-processing is done, a machine learning algorithm for binary classification is run on the dataset (for example, LightGBM).

We obtain probabilities of failure **p** or alerts **a** for each time step. An alert a is a number equal 0 or 1 where 0 represents no alert (the machine is assumed to be in a normal state) and 1 a suspicion of failure. If **p** is the probability of failure, we get an alert **a** by filtering **p** by a threshold **s**. The alert **a** equals 1 if **p** is greater than **s**, else 0. The sensors report the measurements over time, creating multivariate time series. We create running summaries of the sensor values. For each sensor, we run a feature *window* of length 10 and we compute the mean, standard deviation, max, min, median over the feature window. This is a standard pre-processing for time series. Fortunately, the failure rate in predictive maintenance is low.

[17] GitHub - shadgriffin/machine_failure

For supervised learning, the dataset is very unbalanced. Moreover, the objective of predictive maintenance is to predict damage in advance. Predicting a breakdown on the day the event occurs is not very useful. We therefore extend the target window, considering that the *n* points before the failure are themselves considered as positive, where *n* is the length of the target window. Predicting a failure up to *n* time steps before the actual failure is considered as a valid prediction; this is equivalent to looking for a failure in any of the next *n* steps. This approach fits naturally in predictive maintenance and enables to mitigate the unbalanced dataset issue. Recall that the target window expands the failures to *n* time steps behind the actual failure, where *n* is the length of the target window. The objective is to balance the dataset and to predict a failure before the actual failure; the aim is to anticipate the breakdown. If the binary classification algorithm works well, we face a problem in the prediction phase: we may have up to *n* alerts for one true failure of the equipment, preceding the breakdown. Filtering the alerts is thus necessary. In this simple case, we scan sequentially the predicted alerts and we remove the alerts that are less than *m* steps away where *m* is the length of the *forecast window*. Moreover, we consider that a predicted alert is a valid alert if it occurs less than *m* steps before the failure. The forecast window has thus two goals: filtering and removing redundant alerts and providing the authorized anticipating time to predict a breakdown before the actual failure.

Once the alerts obtained by the binary classification algorithm are filtered, via the use of the forecast window, classical metrics for binary classification may be evaluated. In particular, the confusion matrix can be computed. Note that redundant alerts (too close in time) have been eliminated by the forecast window, and an alert is declared as valid if it happens less than m steps before the actual failure where *m* is the length of the forecast window. According to this rule, we evaluate the number of *true positive* (TP), *true negatives* (TN), *false positive* (FP) and *false negative* (FN) for each machine. TP is equal to 1 if there is an alert less than *m* steps before the failure, else 0. FN is the opposite of TP (equals 1 if TP is 0 and 0 if TP is 1). FP is equal to the number of alerts minus TP. In predictive maintenance, *anticipation* is crucial. Another essential point is the true *cost* associated to TP, FP and FN. A TP requires the replacement of the equipment before the actual failure. The associated cost is often much lower than the replacement of the equipment the day of the failure (FN). Besides, a FP translates in practice to a maintenance visit. As a way of comparison, for the simulated dataset, we may suggest that a TP costs 7500, a FN 30000 and a FP 1500. A good predictive maintenance algorithm has no FN (and thus a number of TPs equal to the number of failures), and as few FPs as possible.

On the simulated dataset, we apply a feature window of length 10, a target window of length 28 and a forecast window of length 28. We compute the mean, median, standard deviation, max and min over the feature window. Despite the use of a target window, the dataset remains very unbalanced, with few positive samples (failures) compared to the number of negative samples (normal state). Several complementary approaches may be used to mitigate the problem. First, the imbalanced learn python package offers several tools to deal with unbalanced datasets. In that particular case, the SMOTENC algorithm oversamples the minority class. A second option is to tune the scale pos weight parameter in LightGBM to put more weight to the positive samples (minority class). Specifically, we set scale pos weight equal to the ratio of the number of negative samples over the number of positive samples. A third option is to use the Focal Loss instead of the classical cross entropy loss to train the LightGBM algorithm. The Focal Loss is a variation of the Cross Entropy loss to deal with unbalanced datasets. We train LightGBM with a focal loss, without SMOTENC, and setting scale pos weight parameter equal to 1. We stop the training if the AUC does not improve over 10 steps. In this example, AUC is a sufficiently good metrics for the illustrative purpose of this task; in general, more robust metrics could be considered, such as the F1-score or the Matthews Correlation Coefficient. We obtain a training AUC equal to 0.822 and a test AUC equal to 0.64. To get the alerts from the predicted probabilities, we set a threshold *s* equal to the 0.995 quantile for each machine. We filter the alerts via the

forecast window. We compute the confusion matrix (TP, FP, FN) for each machine and sum them. We get TP equals to 66, FP equals to 223 and FN equals to 19 for 85 machines (test dataset). The average cost by machine is 16464 compared to 30000 without predictive maintenance.

6.3 Illustration on Turbofan dataset

In this Section, we perform a regression task on the Turbofan dataset to predict the RUL. The turbofan dataset[18] consists of four separate challenges of increasing difficulty. Engine degradation simulation was carried out using C-MAPSS. Four different sets were simulated under different combinations of operational conditions and fault modes. Several sensor channels are recorded to characterize fault evolution. The data is contaminated with sensor noise. The engine is operating normally at the start of each time series, and develops a fault at some point during the series. In the training set, the fault grows in magnitude until system failure. In the test set, the time series ends some time prior to system failure. The objective of the competition is to predict the number of remaining operational cycles before failure in the test set, i.e., the number of operational cycles after the last cycle that the engine will continue to operate. A vector of true RUL values is provided for the test data. For each engine, we have the unit member, the time in cycles, three operational settings, and the values of 21 sensors measurements. The RUL is computed subtracting the time in cycles to the maximum time reached by the engine. In this deliverable, we focus on a dataset characterized by one unique operating condition and one fault mode; the operational settings can thus be dropped. An exploratory Data Analysis enables to determine sensors 1, 5, 6, 10, 16, 18 and 19 hold no information related to RUL as the sensor values remain constant throughout time. We put them aside and drop their columns. We consider the Root Mean Square Error (RMSE) and the coefficient of determination to evaluate the models performance. Note that RMSE is *symmetric* with respect to the true RUL, *i.e.* if y denotes the true RUL. In predictive maintenance, predicting a RUL lower than the truth is better than predicting a higher RUL, because *anticipating* the breakdown is key.

We first fit a baseline linear regression model on the FD001 dataset. We drop the unit nr, time cycle, settings columns and sensors which hold no information. The RUL column of the training set is stored in its own variable. For the test set, we drop the same columns. In addition, we are only interested in the last time cycle of each engine in the test set as we only have True RUL values for those records. We use the scikit-learn linear regression. The results are given in Table below.

	RMSE	R2
Train	44.67	0.58
Test	31.95	0.41

Note, the RMSE is lower on the test set, which is counter-intuitive, as commonly a model performs better on the data it has seen during training. A possible explanation could be the computed RUL of the training set ranging well into the 300s. Looking at the trend of Figure 6.1, the higher values of linearly computed RUL do not seem to correlate very well with the sensor signal. Since RUL predictions of the test set are closer to failure, and the correlation between the lower target RUL and sensor signal is clearer, it may be easier for the model to make accurate predictions on the test set. The large difference in train and test RMSE can be seen as a flaw of the assumption of linearly declining RUL.

[18] https://www.kaggle.com/datasets/behrad3d/nasa-cmaps

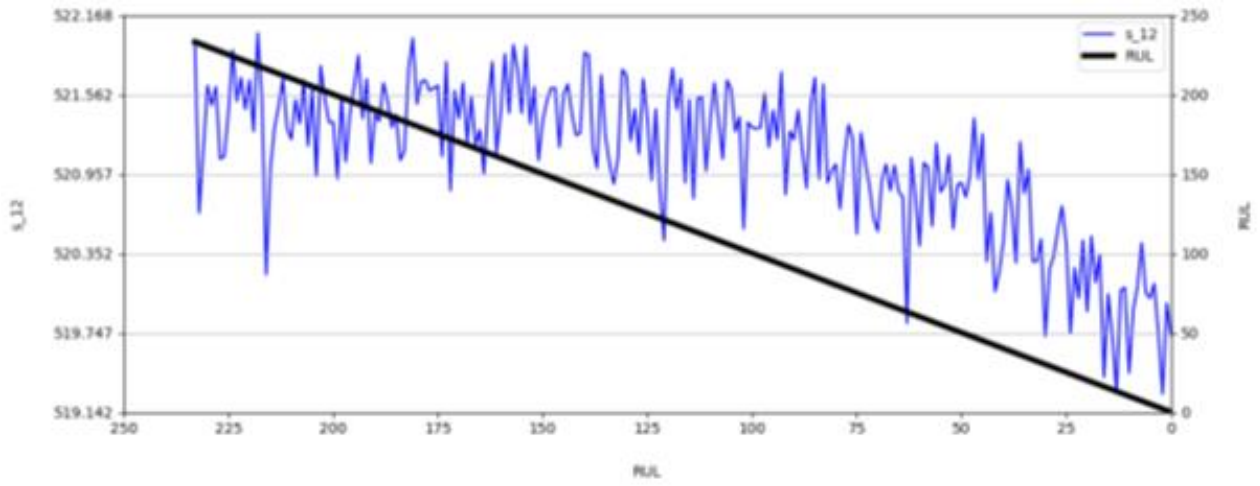


Figure 6.1: Turbofan dataset - RUL signal.

Another essential point specific to predictive maintenance and RUL prediction is the fact that engines only develop a fault over time. Looking at the sensor signals, see Figure 6.2, many sensors seem rather constant in the beginning. The bend in the curve of the signal is the first bit of information provided to us that the engine is degrading and the first time it is reasonable to assume RUL is linearly declining.

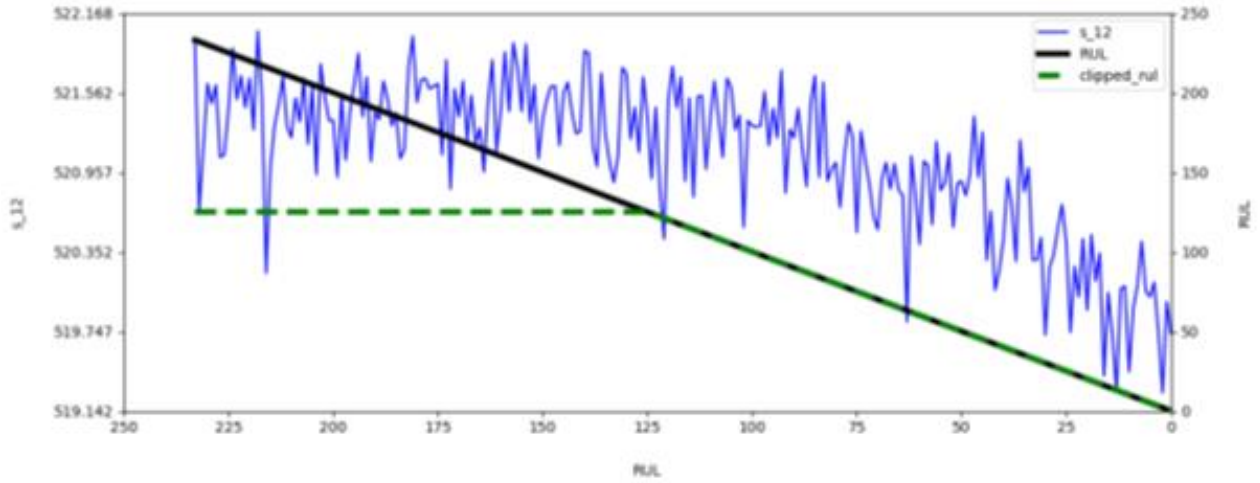


Figure 6.2: Turbofan dataset - original RUL signal and clipped RUL signal.

We can update the assumption of linearly declining RUL to reflect this logic. Instead of having RUL declines linearly, we define RUL to start out as a constant and only declines linearly after some time. In this way, initially constant RUL correlates better with the initially constant mean sensor signal. Consequently, this clipped RUL allows our regression model to more accurately predict low RUL values, which are often more interesting/critical to predict correctly. Testing multiple upper bound values indicated clipping RUL at 125 yielded the biggest improvement for the model. The true RUL of the test set remains untouched. We run the baseline linear regression on this updated dataset. The test RMSE reduced from 31.95 to 21.90, which indicates that the updated assumption is beneficial for modelling true RUL.

Survival analysis focuses on *T* a non-negative random lifetime taken from a population under study. In our case, *T* corresponds to the lifetime of the engine. The *event* corresponds to the occurrence of the phenomenon of interest, in our case, the breakdown of the engine. *Censoring* occurs when the observations have stopped but the subject of interest did not have their “event” yet. In our case, there is no censoring in the training dataset because we observe the engines until their breakdown but the test dataset has censoring, since the observations stop at a certain time before the breakdown. We artificially right-censor

our dataset by disregarding any records after 200 times cycles. This allows to play around with the data in a bit more realistic setting, with a mix of engines which did and did not have their breakdown yet. The random time *T* can be characterized by diverse quantities. The survival function is defined for all *t* >0 by, S(*t*) = P(*T* > *t*). S is the inverse of the cumulative density function of *T*. The probability of the event occurring at time *t*, given that the event has not occurred yet is the hazard function h defined for all *t* >0 by:

$$\mathrm{h}(t) = \lim_{\delta t \to 0} \mathbb{P}\left(t \leq T \leq t + \delta t | T > t\right) = -\frac{\mathrm{S}'(t)}{\mathrm{S}(t)} .$$

We have then

$$\mathrm{S}(t) = \exp\left(-\int_0^t \mathrm{h}(z)\mathrm{d}z\right) = \exp\left(-\mathrm{H}(t)\right)$$

Where H(t) is the cumulative hazard function. About the dataset, we clip RUL values above 125 as discussed above. We add a breakdown column indicating whether the engine broke down (1) or is still functioning (0).

The Kaplan-Meier curve gives an empirical estimation of S(*t*) for *t* >0, see Figure 6.3. This method gives us a crude tool to estimate the probability to survive past time *t* for an engine from the same population. For example, engines have a 100% probability of surviving the first 128 time cycles. After that point the first engines start to break down, but there is still a 46% probability of the engine surviving past 200 time cycles.

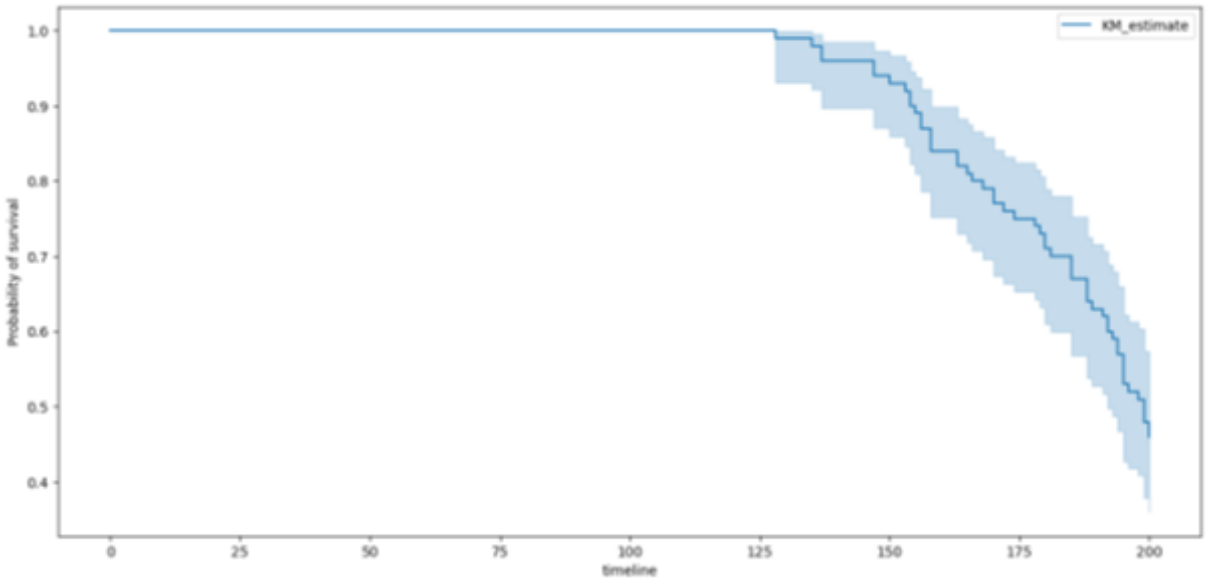


Figure 6.3: Turbofan dataset - RUL estimate with Kaplan-Meier curve

A common model which provides more information is the Cox Proportional Hazards model. Given the sensor values $S_{j\,t}$ for *j* = 1, . . ., *s* at time *t*, the Cox Proportional Hazards model assumes that the hazard function can be written under the following form for all *t* > 0:

$$\mathrm{h}(t|\mathrm{S}) = \mathrm{h}_0(t) \exp\left(\sum_{j=1}^{s} a_j (\mathrm{S}_t^j - \overline{\mathrm{S}^j})\right)$$

where h_0 is the baseline hazard function, and $(a_j)_{sj=1}$ the regression coefficients. We use the Lifelines Python package to fit the model, see CoxPHFitter and CoxTimeVarying-Fitter. The baseline hazard is modeled by a non-parametrically model, using Breslow's method. In our case, we expect the baseline hazard to be mostly constant with respect to time, since all the variations are handled in the sensor values S_{jt}. The results are reported in Figure 6.4. Since the baseline hazard is mostly constant, we only have to inspect the partial or log-partial hazard to get an indication of the risk of failure. The partial hazard only has a meaning in relation to other partial hazards from the same population. An engine with a partial hazard of 2 is twice as probable to breakdown compared to an engine with a partial hazard of 1. Since the partial hazard values are rather large, it's easier to display the log of the partial hazards.

The exp(coef) shows the scaling hazard risk, e.g., an increment of 1 unit for the sensor values of sensor 11 increases the risk of breakdown by 167.43.

covariate	coef	exp(coef)	se(coef)	coef lower 95%	coef upper 95%	exp(coef) lower 95%	exp(coef) upper 95%	z	p	-log2(p)
s_20	-5.13	0.01	2.02	-9.09	-1.17	0.00	0.31	-2.54	0.01	6.49
s_7	-1.09	0.34	0.50	-2.07	-0.11	0.13	0.90	-2.17	0.03	5.06
s_13	12.90	399292.51	5.51	2.09	23.70	8.09	19712757469.56	2.34	0.02	5.69
s_2	2.05	7.75	0.72	0.64	3.45	1.90	31.64	2.85	0.00	7.84
s_15	5.07	159.96	9.53	-13.61	23.76	0.00	20873966596.67	0.53	0.59	0.75
s_17	0.41	1.51	0.20	0.01	0.81	1.01	2.25	2.03	0.04	4.56
s_4	0.16	1.18	0.05	0.07	0.26	1.07	1.29	3.49	0.00	11.03
s_11	5.12	167.43	1.70	1.78	8.46	5.95	4714.95	3.01	0.00	8.56
s_14	0.04	1.04	0.04	-0.04	0.12	0.96	1.13	0.95	0.34	1.54
s_21	-4.23	0.01	2.67	-9.46	1.01	0.00	2.75	-1.58	0.11	3.14
s_9	-0.02	0.98	0.04	-0.10	0.05	0.91	1.05	-0.58	0.56	0.82
s_8	-5.03	0.01	4.84	-14.51	4.45	0.00	85.79	-1.04	0.30	1.74
s_3	0.07	1.07	0.04	-0.01	0.15	0.99	1.16	1.73	0.08	3.57
s_12	-1.14	0.32	0.53	-2.17	-0.10	0.11	0.90	-2.15	0.03	4.99

Figure 6.4: Turbofan dataset - hazard table

Figure 6.5 essentially displays the coefficients and confidence intervals of the features.

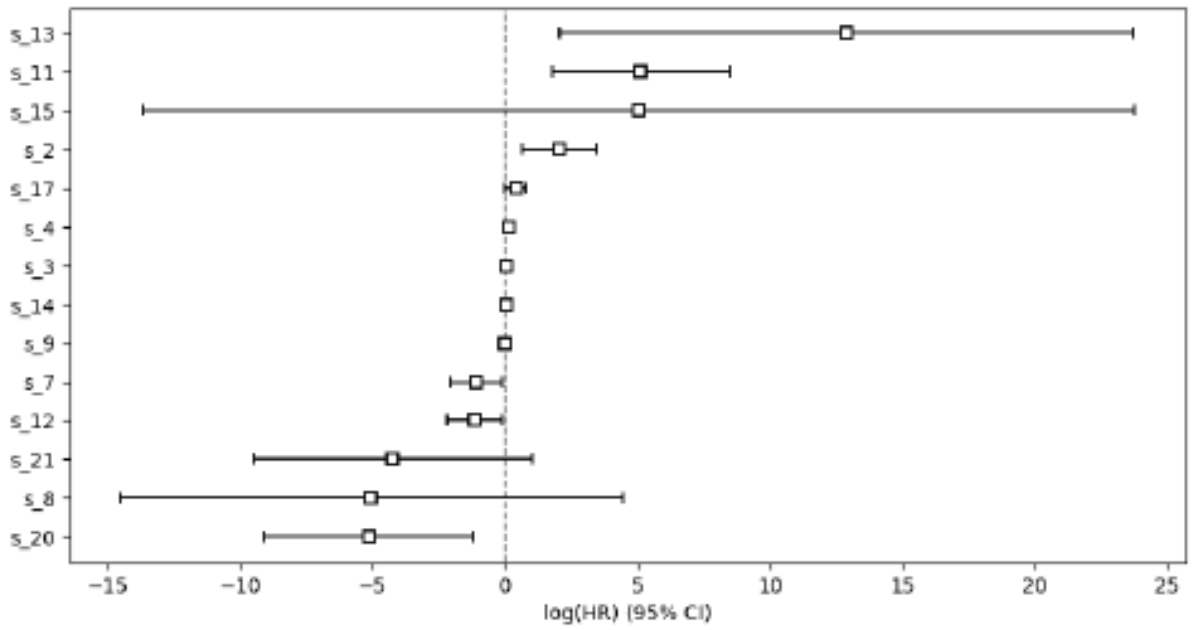


Figure 6.5: Turbofan dataset - feature coefficients and confidence intervals

When comparing the log partial hazard with computed RUL in Figure 6.6, we observe that it generally informs quite well about imminence of breakdown. Higher log partial hazards are returned for engines more at risk of breaking down.

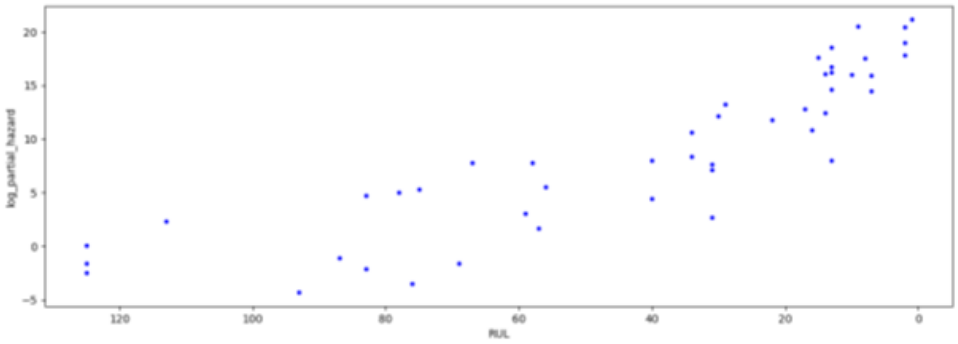


Figure 6.6: Turbofan dataset - computed RUL and log partial hazard

Since we deal with time series data, the log partial hazard can also be predicted over time, see Figure 6.7. One possible way to anticipate breakdown would be to set a threshold for the log partial hazard after which maintenance should be performed. However, that provides no information on RUL. A possible improvement would be to develop a time-series model to predict when this threshold is reached to get a ”time to event” prediction.

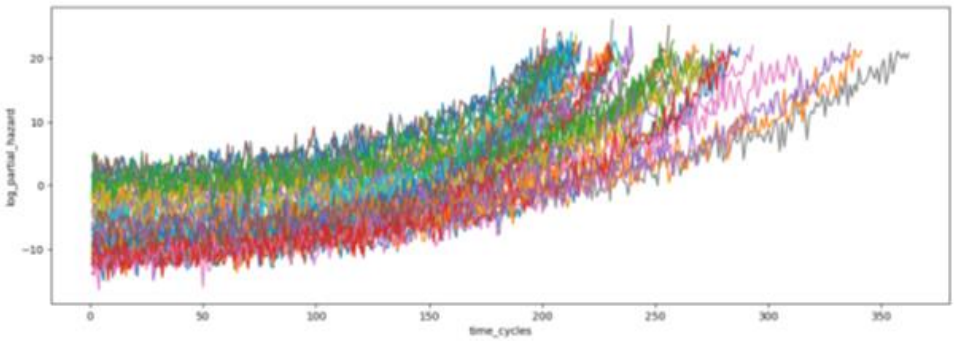


Figure 6.7: Turbofan dataset - log partial hazard computed over time

We list several python packages for survival analysis (in order of decreasing stars on GitHub):

- Lifelines
- Scikit survival
- Pysurvival

7. Conclusions

In this document we provided an overview of the final version of the Artificial Intelligence-based services developed within the IoTwins project. In this deliverable we cover services based on three main macro-areas within the Artificial Intelligence field: Machine Learning, Simulation, and Optimization. The listed services have been developed following the requirements expressed by the testbed partners (see D3.1). The services have been developed following a modular approach, aiming at decomposing the problems to be solved in their main components. Moreover, the services described here are mostly generic services which can be used adopted in different context where the same task needs to be tackled. When migrating from a context to

another, it is unavoidable to perform a tailoring action to specialize the service for the particular characteristics of the target system, such as pre-processing the data to make it compatible with ML services or compose the different subservices in the desired workflow (e.g., data split for training and testing of ML models, hyperparameter fine-tuning on a set of data preserved for validation, etc.). These operations must be performed by users which want to adapt these general services to their specific needs, and can be done by following the guidelines presented in this deliverable and in the series of meetings, seminars, and workshop which have been held during the course of the IoTwins project.

The services described in this document were deployed on the IoTwins platform and thoroughly tested on different data sets. Furthermore, the developed services have been widely used in the various testbeds, for instance anomaly detection (semi-supervised) in TB06 (large-scale data centre) and RUL estimation in TB04 (manufacturing), demonstrating the effectiveness of the proposed approach and services in the expected use-cases.