Master's Thesis : State segmentation and forecasting of production processes by machine learning

Auteur : Roekens, Joachim
Promoteur(s) : Wehenkel, Louis
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"
Année académique : 2019-2020
URI/URL : http://hdl.handle.net/2268.2/9058

Master thesis

State segmentation and forecasting of production processes by machine learning

Author

Joachim *Roekens*

Promoter

Louis *Wehenkel*

Institution

University of Liège - Faculty of Applied Sciences

In collaboration with

WideTech

Graduation Studies conducted for obtaining the Master's degree in Computer science and Engineering

Academic Year 2019-2020

Abstract

In this thesis, two different tasks concerned with time series of production processes are tackled. The first one is a time series segmentation into different classes for each timestep of the time series. An accurate timestep classification has multiple useful applications such as doing retrospective analyses of production processes based on the percentage of occurrence of each class or implementing an intelligent system to activate or deactivate alarms applied to processes based on their current state. Then, the second task is a multistep multivariate time series forecasting[a]. It can be used to forecast specific events in order to avoid them or to prepare for them.

This work focuses on the application of machine learning algorithms to those two problems with the objective to automate and generalize the solution to the broadest range of production datasets as possible. The end goal is to study the potential of those algorithms, rather than delivering a perfect solution.

For the time series segmentation, tree-based models are considered. In the final evaluation, they display an irregular performance alternating between very high and low accuracy depending on the classes. However, the lack of precision might be caused by an external bias in the data labelling. Still, its performance on the best classes reveals its high potential.

For the time series forecasting, the study focuses on deep learning algorithms which gave good results in this domain. Two state of the art level models are tested: DeepAR and Temporal Fusion Transformers (TFT). The evaluation demonstrated the difficulty encountered by the models and, by extension, the difficulty of an automated timestep forecasting of a wide range of datasets by deep learning.

[a]Multistep time series forecasting denotes the fact of predicting multiple timesteps of a time series while the multivariate term indicates that forecasts are done on more than one value for each timestep.

Contents

Acknowledgements

I would like to thank

- my promoter Dr L.Wehenkel for his general helpful guidance,
- my promoter's assistant L.Duchesne for her advice which has greatly improved the quality of the thesis redaction,
- my main contact person in WideTech O.Ghaye for our weekly talks during which we kept track of the thesis progress and for his help provided at multiple times,
- WideTech for his warm welcome and the interest you have all shown toward my work,
- my teacher Dr P.Geurts for his help in decrypting the inner working of the XGboost classifier,
- everyone who helped by proofreading this work.

Thank you all for your support, without you this work wouldn't have been the same.

Introduction

Large production systems are commonly seen in the industry sector. Those systems are monitored with sensors relaying their measurements. Then, by analysing it, a clear image of the state and behavior of the system can be built. However due to the size of production systems and their multiplicity, analyses are difficult and time consuming to tune to each process. The general goal of this work is to apply automated algorithms to solve this problem.

In this work, two specific tasks are tackled. The first one is a classification of the past actions of the system into production states such as low production, stable production, system shutdown, etc. Once the states have been assigned, it is trivial to collect useful information, for example, knowing the percentage of time when the system was running at full capacity. It can also be applied in real time as knowing the current production state opens the possibility of implementing real-time policies based on the production state. On the other hand, the second task is concerned with the future state of the system. It is a forecasting of key production signals which could be used to anticipate problems. One task explores the past and current behavior of the system while the other one predicts its future. Together they can provide a clear and automated view of the system.

The techniques considered are machine learning algorithms as they have shown high quality performances in similar problems. More specifically, tree-based models are applied in the first task while the second one is addressed with Temporal Fusion Transformer (TFT) and DeepAR, two deep learning models. The goal is to do a proof of concept for both tasks. All the code implemented for those two is available in the archive provided with the report.

To evaluate the proposed algorithms, data from real production processes is used. Their application should not be restricted to any particular process, thus two different types of production are observed in this study. The first one is an offshore oil extraction process while the second one is a fertilizer granulation.

This thesis is conducted in collaboration with WideTech. WideTech is a Belgian company founded in 2012 with the main purpose of providing software solutions and related engineering services for the process industry to gather data and assess its quality in near real-time mode, thereby enabling informed and better decisions. It provides services such as:

- *improving the maintenance of instrumentation and equipment :* which reduces risks of operations (failures, unwanted shutdowns, etc.),
- *monitoring process performance :* which reduces performance losses and find improvements,
- *providing quality data – both for production or energy accounting :* which reduces time spent on turning information into assets.

The thesis is organised in three chapters, one for each task and one for the final conclusion. The first two chapters are structured with, in order, the task goal, the data description, the explanation of the algorithm, the evaluation metrics, the evaluation of the algorithms and finally, the task conclusion.

Chapter I

First task: Time series segmentation

In this section, machine learning algorithms, and in particular tree-based methods, are applied to the task of time series segmentation. Its goal is to classify each timestep in the right class. The proposed method shows an irregular performance, alternating between high and low accuracy depending on the classes. However, this might not reflect its segmentation capability as the test does not generalize the possible applications of the solution. Other experiments might dispel the irregularity in its performance.

Contents

1 First task goal

Retrospective analysis is an important part of production management. It can provide information such as the number of unexpected shutdowns, the average time of shutdowns, the total time period of production at maximal capacity, etc. This analysis can be done by classifying past time periods into states: shutdown, recovering, full-capacity production, etc. By knowing the states, the task becomes trivial. On top of the state knowledge about the past, the one about the present is also useful. For example, with it, it becomes possible to activate alarms only when the process is in a particular state.

However, states can be difficult to obtain. For example, in the offshore oil extraction domain, the state is based on the oil flow. During the extraction, the pipe contains a mixture of oil, water and gas and a multi-phase flow meter is required to measure the flow. Then, hand-crafted rules can be written to classify timesteps based on those measurements, still, this process is time consuming and prone to human error as it is difficult to think of all possible cases. Instead, having an intelligent automated solution would reduce the time loss and increase the classification accuracy. Thus, the goal of this part of the thesis is to provide a solution, based on machine learning, capable of doing an automatic time series segmentation using time-series data provided by sensors.

This work considers a production system represented by measured signals arranged in time series with a constant time interval between timesteps. The objective is to automate the classification of the previous timesteps into production states (cf. Figure I.1). This would provide information about the production history which can be used for a retrospective analysis. In other words, this task is a classification of timesteps from multivariate time series.

More precisely, the algorithm should be able to :

- choose the meaningful signals for the classification,
- classify timesteps of the production based on the previously selected signals ,
- be interpreted by an engineer working in the production domain to bring helpful information on how the classification was made,
- handle missing data,
- be applicable to a wide range of production systems.

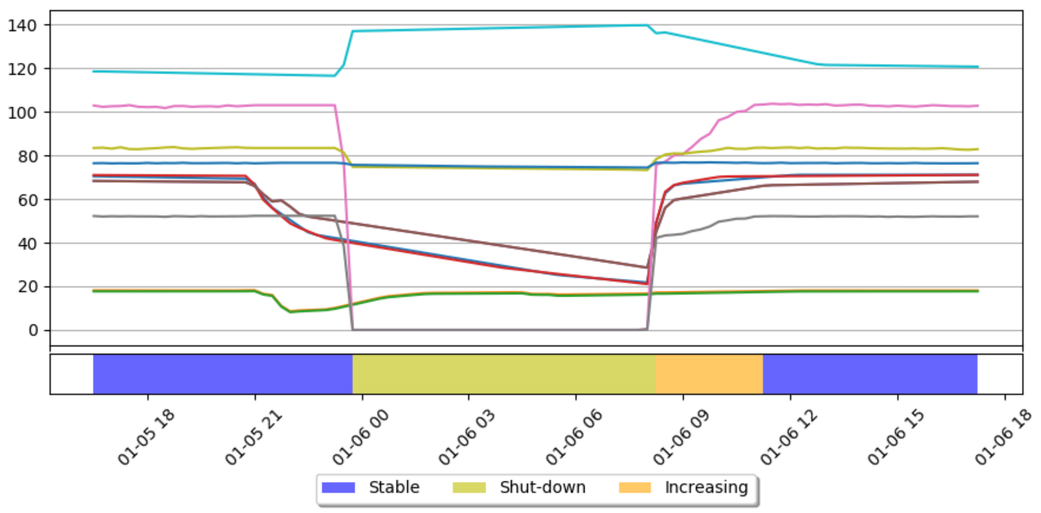


Figure I.1 – Example of time series segmentation with 3 classes. A time series composed of 8 signals is displayed on top, while its segmentation is shown below.

2 Data description

The data on which the time series classification will be applied is not set. As explained in Section 1, the goal is to be able to classify a wide range of production processes. The data is supposed to be a production data consisting of a series of measurements done at different time.

Each measurement point provides information about production signals. The production signals could be for example, the motor temperature, the flow in a pipe, the pump pressure, etc. These signals usually give one value each time they are measured. However, during the raw data preprocessing, multiple small timesteps are aggregated at a lower constant frequency (i.e. bigger timeteps); going from 1 data point every 1-2 minutes to 1 data point/hour for example. This can be used by the preprocessing to compute useful aggregations per timestep: standard deviation, minimum, maximum, gradient, average, etc.

In summary, the time series classification is applied on production data made of a series of points, spaced by a constant time difference, where each point contains computed values such as the standard deviation, the gradient, the average, etc., for each of the measured production signals. The computed values are referred to as features.

2.1 Data labeling

As the used methods are based on supervised machine learning, labeled data is required. Models will learn from the data and tune their parameters to fit it as best as possible. labeled data consists in the time series explained above with an additional feature per point: the production state. The new feature classifies each timestep into their corresponding production state.

Data labeling is one of the foundations of the supervised machine learning algorithms. Inconsistent and imprecise labels will mislead the models and thus reduce their accuracy. This should not be overlooked.

2.2 Detailed description of the experiments data

The algorithm solving this task will be tested on two datasets. The first one is used at the implementation phase to check the algorithm correctness and to choose the model and the imputation technique. The second dataset, however, is never used during the implementation but rather in the final tests for a performance evaluation. Both datasets are real anonymized production data of two different production systems.

The general characteristics of the datasets are given in Table I.1 and samples can be visualized in Figure I.3. Six features are computed by WideTech for each measured signal using the measurements to aggregate for each timestep: average, standard deviation, minimum, maximum, gradient, interpolation. For all aggregation dates d distanced by a constant interval t, the values are computed with:

$$average_d = \frac{\sum_{d \leq d_i < d+t} x_{d_i}}{\sum_{d \leq d_i < d+t} 1}, \tag{I.1}$$

$$standard\_deviation_d = \sqrt{\frac{1}{\sum_{d \leq d_i < d+t} 1} \sum_{d \leq d_i < d+t} (x_{d_i} - average_d)^2}, \tag{I.2}$$

$$minimum_d = \min_{d_i<d+t}(x_{d_i}), \tag{I.3}$$

$$maximum_d = \max_{d_i<d+t}(x_{d_i}), \tag{I.4}$$

$$gradient_d = x_{d_2} - x_{d1}, \quad with \begin{cases} d_1 = \min_{d\le d_i<d+t}(d_i) \\ d_2 = \max_{d\le d_i<d+t}(d_i) \end{cases}, \tag{I.5}$$

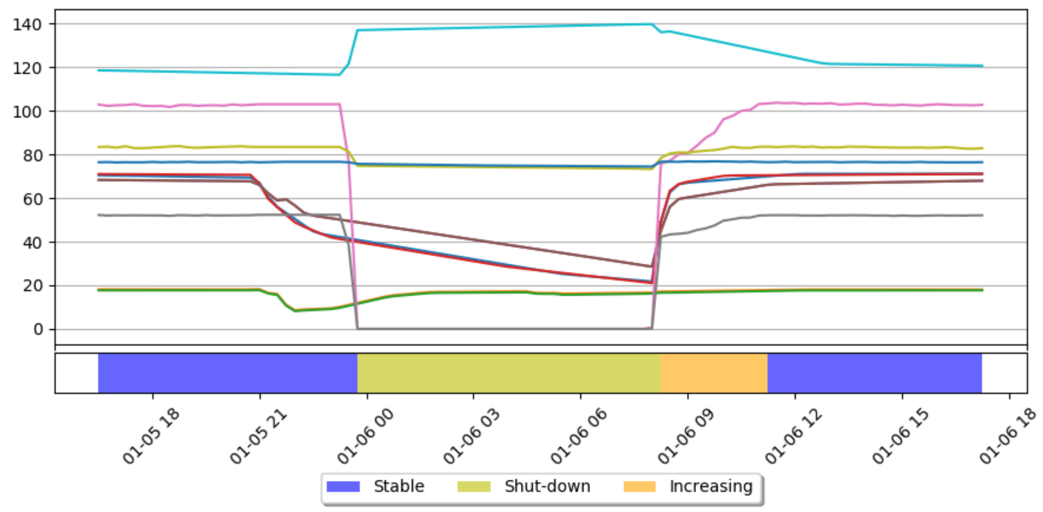
$$interpolation_d = \begin{cases} x_d & : \exists x_d \\ x_{d_1} + \frac{x_{d_2}-x_{d_1}}{d_2-d_1}(d-d_1) & : \nexists x_d \end{cases} \quad with \begin{cases} d_1 = \max_{d_i<d}(d_i) \\ d_2 = \min_{d_i>d}(d_i) \end{cases}, \tag{I.6}$$

where all the notation d_i refer to measurements dates and x_e is the measurement at date e.

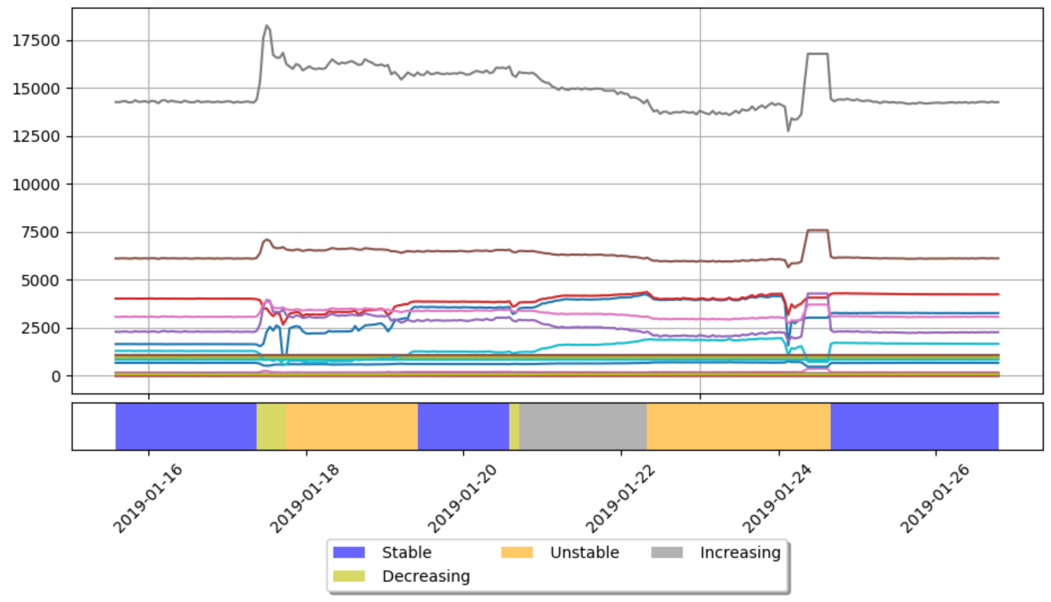
Both datasets are manually labeled by an engineer working on a daily basis on this kind of data.

Name	Classes	Time interval	Start date	End date	Points	Signals	Features
ds1	3	15 min	2020/01/01 00:00	2020/03/31 23:45	8736	11	66
ds2	6	1 hour	2019/01/01 00:00	2019/12/30 01:00	8714	49	294

Table I.1 – Datasets characteristics



(a) Sample of dataset ds1



(b) Sample of dataset ds2 (only 4 out of the 6 signals are visible here)

Figure I.3 – Subsamples of the datasets ds1 and ds2. Each displays the average of the signals on top of the classification of the timesteps of the subsamples.

Production process of segmentation dataset ds1

The process of ds1 is an offshore oil extraction (cf. Figure I.4) whose simplified explanation is the following. A mixture of gas, oil and water is extracted by one well then a pump increases their flows. Afterwards, a multiphase flow meter (MPFM) measures their flows individually. The signals found on this dataset correspond to temperatures, pressures, flows, etc. within the region A (cf. Figure I.4) but also to the temperature of the pump motor or to the current it uses.

The three states shutdown, increasing and stable respectively correspond to when the well has to be stopped (for example because of a failure in the pump), when the well restarts its activity and when the well has a normal activity (as opposed to being stopped or restarting).

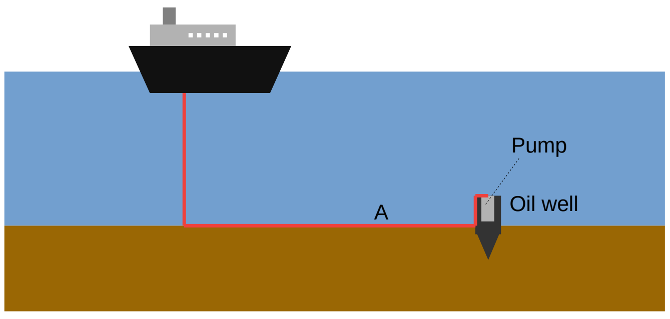


Figure I.4 – Simplified context of ds1: offshore oil well data focused on one well. A pump increases the flow of gas, oil and water extracted by the well while a multiphase flow meter (MPFM) is applied in the area A to give the gas, oil and water flow.

Production process of segmentation dataset ds2

The context for ds2 is a more complex offshore oil extraction (cf. Figure I.5) which can be simplified into the following. Multiple oil wells are placed in series. Then a multiphase pump (MPP) increases the flow inside the pipe which contains a mix of gas, oil and water extracted from the wells. Finally, a multiphase flow meter (MPFM) measures flows individually in terms of gas, oil and water. The main signals are measurements such as temperature, pressure, flow, etc. from sensors in regions A and B, the MPP motor speed and the individual gas, oil and water flows in A.

The time series is to be segmented into the following states:

- *stable :* stable production,
- *unstable :* unstable production,
- *decreasing :* production in decrease,
- *increasing :* production in increase,
- *shutdown :* stopped production,
- *flat :* unknown state caused by a sensor failure causing signals to stay constant.

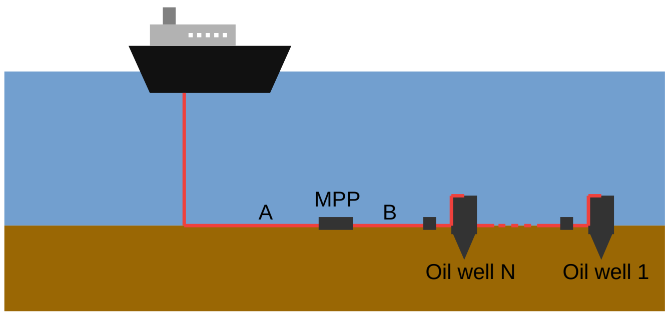


Figure I.5 – Simplified context of ds2: offshore oil well where N wells are put in series and a multiphase pump (MPP) is applied to increase the flow. The region A is where the multiphase flow meter (MPFM) is applied which gives the gas, oil and water flow.

2.3 Consideration of missing data

Missing data can be classified in three categories: missing completely at random (MCAR), missing at random (MAR) and missing not at random (MNAR) [Mack et al., 2018]. With MCAR data, being missing is independent of the observed and unobserved data. Thus the fact that data is missing brings no information. However with MAR and MNAR data, being missing is respectively dependent on the observed data and the unobserved data. In those cases, containing missing data in a sample provides information. For example, in MNAR, if a heat sensor is exposed to a heat higher than its maximal threshold, it might fail and cause the appearance of missing data. Thus, when missing data are observed, it could mean that the heat is above the threshold.

Handling the missing data can be done in three ways:

1. drop all incomplete samples,
2. replace the missing values by imputed ones,
3. use a model handling missing values by itself.

If the data is MNAR, the first solution would lead to a bias in the model. Indeed, with the previous example of the heat sensor, if the state of the production system was 'danger due to high heat level', dropping all samples containing the missing data would lead to a model unable of classifying this state. Moreover, such a model would be incapable of handling any incomplete sample, which usually goes against the task. The other two solutions are more common. However, in case of MNAR, imputing the data could lead to a loss of information and using a model handling the missing data could be preferred.

In this task, missing data is considered mostly MCAR with very rare cases of MAR and MNAR. It is usually caused by a faulty sensor or a faulty transmission. Moreover, as the data is merged in larger timesteps as explained earlier, some random missing data have already disappeared during the aggregation process. According to engineers working in the domain of the data, missing data should be rare. Preprocessings applied directly after the measurements already removes missing data even before the merging explained above. Thus, due to its rarity, dropping the samples with missing data in the dataset is an acceptable solution. Nevertheless, for the sake of generalizing the solution, data imputation and missing values handled by models are applied instead. Those two are later tested and compared (cf. Sections 4 and 8).

3 General description of the solution

The general idea of the solution is the following. Firstly, a manually labeled dataset is provided as input to the algorithm. Secondly, the algorithm learns from this dataset how to segment the data; in other words, it fits its model to the dataset. Finally, unlabeled datasets can be fed to the solution to be segmented and corresponding labeled datasets are outputted.

The proposed solution is a black box capable of:

- *selecting* automatically which features of the dataset are useful for the classification,
- *learning* a segmentation model based on a given labeled dataset,
- *classifying* each timestep of a given unlabeled dataset,
- *ploting* the segmentation,

where it is acceptable for datasets to contain missing value. This black box is implemented with object-oriented python codes. It is composed of three main components: a data imputation module, a feature selector and a model.

First, the data imputation replaces the missing data contained in a dataset by realistic values such that models using only complete datasets can be applied to sparse ones. Then, the feature selector retrieves the useful features out of all those available to improve and speed-up the classification. Finally, the model is the main component capable of learning how to classify the data based on the features chosen by the feature selector. The model is based on supervised machine learning techniques. This means that for it to learn how to segment the time series, data already segmented is required for training. Those three modules are respectively explained in Sections 4, 5, 6. An example showing how they interact with each other is displayed in Figure I.6.

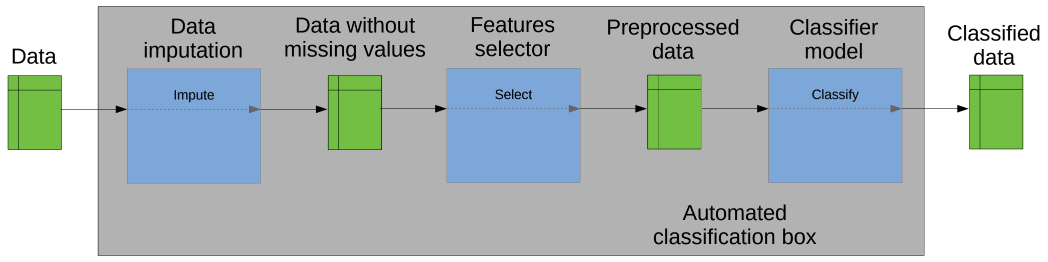


Figure I.6 – Interactions between the three modules of the classification box during the classification process. First, a dataset possibly containing missing data is provided. It is forwarded to the data imputation module which replaces the missing values by realistic ones. Then the imputed dataset is sent to the feature selector which selects the best set of features. Finally, the model takes the output of the features selector and classify each timestep.

4 First module: the data imputation module

The role of the data imputation module is to handle missing data. Based on a reference dataset, it can impute the missing values contained in a second dataset or in the reference one. The imputed values should be as close as possible to the real ones if those were available. The data imputation is an important part of the solution if the model used cannot handle missing data by itself. In this case, without this module the use of this solution would be restricted to complete datasets, as explained previously (cf. Section 2.3). However, this module is not required when the model handles missing data. As imputing missing data takes time, removing the module would increase the time efficiency. Furthermore, the presence of missing data might be a useful information for the task and then removing the module might improve the results given by the model.

The imputation module uses the iterative imputation algorithm (cf. Algorithm 1). Iterative imputation is very flexible and can be used in many applications [Azur et al., 2011].

Algorithm 1. Iterative Imputation

1. *First simple imputation :*

   Missing data are replaced by values found using a simple imputation method such as mean imputation. These values are referred to as imputed values.

2. *feature set initialization :*

   A feature set S is initialized: $S = \{feat$ for each feature $feat$ of the dataset$\}$.

3. *Feature selection :*

   One feature $feat$ of S is randomly chosen and removed from S. Every one of the imputed values of $feat$ are set back to missing.

4. *Fit :*

   A model is fitted on the regression of the non-missing values of $feat$ using the other features.

5. *Imputation :*

   Values are imputed by the newly fitted regression model to replace the missing values in $feat$.

6. *Iteration on features :*

   Steps 3–5 are repeated until S is empty.

7. *Iteration on cycles :*

   Steps 2-6 are repeated N times. N can be fixed or dynamically determined to wait until convergence.

Implementation in this study:

In this case, the regression models considered for the imputation are the k-nearest neighbors or the Bayesian Ridge regression. However, the final version of the module is implemented with the *IterativeImputer* of SKLEARN with a k-nearest neighbors. This method is preferred over the Bayesian Ridge regression as it showed better results in the experiment explained in the Section 8. Each time the classification box has to impute missing values, the last learning set is used as the reference data. The reference data is then used to compute the imputation functions, which are themselves used to impute realistic values in place of missing ones. A learning set should contain a wide range of samples allowing a model to generalize well to the data. Thus it is also perfect as the reference data for the imputation method and should lead to realistic imputed values.

However, as explained before, the data imputation method's usefulness peaks when the model cannot handle the missing data itself. During the same experiment comparing the Bayesian Ridge regression and the k-nearest neighbors as imputation method (cf. Section 8), two models are tested: one incapable of working with missing data and another one using the existence of missing data in its classification. The experiment shows that the second model, which does not use the imputation module, gives better results. In light of this, this imputation module, while still existing inside the code, is deactivated and is never called to reduce computation time and improve the accuracy.

5 Second module: the feature selector

The dataset of a production system consists in time series of multiple signals. Each signal has variables such as the mean, standard deviation, minimum, maximum, etc. All those features are then used by the model to classify the timesteps of the time series. This can lead to a large number of features. Furthermore, some time series require features from the previous and/or the next timesteps to classify the current one. Indeed, let a dataset be made of one time series corresponding to the water flow in a river, and let a timestep be annotated as 'unstable' if during the last 10 timesteps the flow switched between 'high' to 'low' at least 7 times, then for the model to be able to classify the 'unstable' timesteps correctly, it requires to know previous ones. Another example would be a 'peak' state when the current timestep is 'high' and the next and previous ones are 'low'. In this case, the previous and next timesteps are indispensable for the classification.

Thus, many features are provided from the current timestep but also potentially from previous or next ones. Out of these features many can be irrelevant or redundant and it is up to the model to avoid getting entangled with useless ones. By selecting a relevant subset of them before providing them to the model, it can increase the time efficiency as it reduces the complexity of the task. It can also increase the model accuracy by removing features which would have lead it astray. However which features to take and how many previous and next timesteps to use can be unclear to engineers. Thus an automated feature selector can be interesting for this solution. The goal of the feature selector is to provide a small subset of the features well defining the data in order to ease the work of the model. Subsequently, it also helps deciding how many previous and next timesteps to take to obtain the best classification.

Implementation in this study:

The implementation is based on one simplistic handcrafted algorithm (cf. Algorithm 2), whose main idea is the following.

First, Pr and Ne are fixed, they are respectively the number of previous and next timesteps. From the dataset given to the feature selector, $d1$, a new dataset is built, $d2$. Each timestep of $d2$ consists of the concatenation of $Pr+1+Ne$ timesteps of $d1$, each classified by the classification of the $(Pr+1)$-th timestep. This way, each timestep of $d2$ contains all features required for its classification. Using $d2$ instead of $d1$ is necessary in the next step to divide the dataset into 3 sets with a balanced class proportion.

Secondly, the dataset is divided in three: the learning set (LS), the validation set (VS) and the testing set (TS) with a balanced class proportion close to $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{4}$. To insure the class proportion, the total number of instances per class is computed and LS, VS and TS are initialized with empty sets. Then, the algorithm iterates through each timestep of the dataset. Let c be the class of the current timestep, if less than $\frac{1}{2}$ of the c instances are already in LS, the timestep is added to LS. Else, if less than $\frac{1}{4}$ of the c instances are already in VS, the timestep is added to VS and, if not, it is added to TS. The chronological order of timestep is kept as much as possible as it is unlikely for the model to be used only on past values. Thus the testing set contains timesteps ensuing those of the learning set as much as possible, to be tested in conditions closer to the reality.

Thirdly, the LS is used to train a first model capable of giving the feature importance. Its classification error on VS and TS is computed and stored. Using the feature importance given by the model, a new set of features is selected and a new model is trained on LS using only the features from the new set. Then, the algorithm loops over N selection steps. At each selection steps, its classification error on VS and TS is computed and stored. Then, if the classification error on VS done by the model is higher than the previous error on VS, the previous feature set is selected. If not, using the feature importance a new feature set is selected. After a new set has been selected, a model is trained on it. After N selection steps, the lowest error on TS obtained during the selection designates the best feature set.

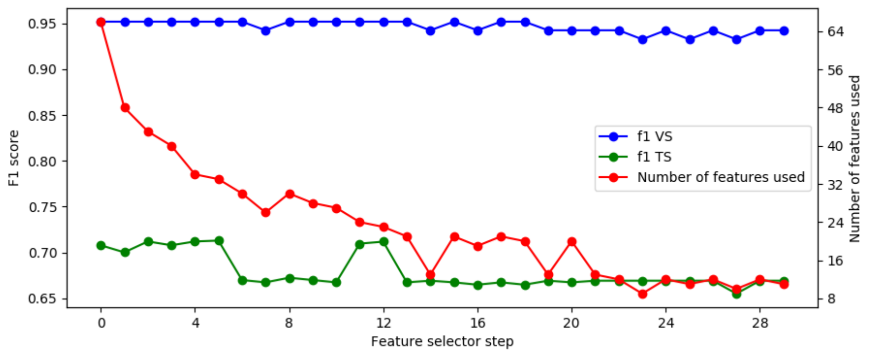
Finally, the process is repeated M times, each fixing a different pair of Pr and Ne. The lowest error designates the best Pr and Ne as well as their corresponding best feature set. An example of the feature selection process can be seen in Figure I.7. From it, a big change of value between the classification error on VS and TS is visible. It is caused by the time difference between the two, as the timesteps in TS are further from LS than VS is to LS, making the values in TS more different. For example, imagining that 2 windows of 3 timesteps of a class C are present in the dataset, then LS contains 2 timesteps of C from the first window, LS 1 timesteps of each of the C windows while TS only has the timesteps from the second C windows. If the two windows differed a lot in terms of value, it is understandable that the classification error on TS will be greater than the one on VS.

In the implementation, it is possible to directly give the feature set instead of activating the automated selection. One can also choose the maximal number of past and future timesteps to search from, before running the feature selector.

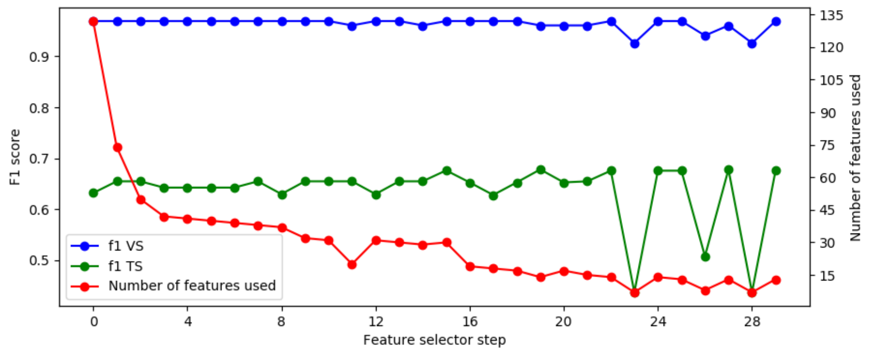
The features selector is not a requirement for the solution neither is it the main part of the task. Taking this into consideration, the feature selector uses a simplistic and fast to implement algorithm. However, more complex and efficient algorithms exist and could be implemented to obtain better selections: Laplacian score [He et al., 2005], Multi-Cluster Feature Selection [Cai et al., 2010], nonnegative spectral analysis [Li and Tang, 2015], etc.

Algorithm 2. Feature selection for a fixed number of previous and next timesteps

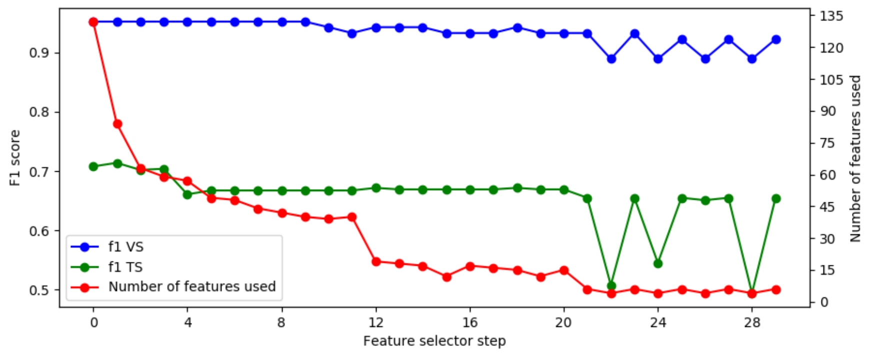
```
1  $f \leftarrow$ set of all features;
2  $f_{prev} \leftarrow f$;
3  $\{score_{prev}, len_{prev}, len_{prev\_rem}\} \leftarrow \{0, 0, 0\}$;
4  $num\_rem \leftarrow 1/4$;
   // Loop over N selection steps
5  for $steps \leftarrow 0$ ; $steps < N$; $steps \leftarrow steps + 1$ do
6      $data_{lsx}, data_{lsy}, data_{vsx}, data_{vsy} \leftarrow$ preprocess($data$, $f$);
7      clf $\leftarrow$ tree_based_model().fit($data_{lsx}$, $data_{lsy}$);
8      $score \leftarrow$ clf.$score(data_{vsx}, data_{vsy})$
9      if $score_{prev} > score$ then
           // Go back to previous features
10         $f \leftarrow f_{prev}$;
11         if $len_{prev} == length(f)$ then
12             $thresh\_i \leftarrow thresh\_i/2$;          // Reduce threshold to avoid deadlock
13         else
14             $len_{prev} = length(f)$;
15         end
16     else
           // Try new features
17         $cnt_{prev} \leftarrow 0$;
18         $f_{prev} \leftarrow f$;
19         $score_{prev} \leftarrow score$;
20         $thresh \leftarrow thresh\_i$;
21         while $f == f_{prev}$ do
22             $f \leftarrow$ feat_above_ threshold(clf.feature_importance, $f$, $thresh$,
                max_removed $= num\_rem * length(f_{prev})$);
23             $thresh \leftarrow 2\ thresh$;
24         end
25         if $num\_rem * length(f_{prev})$ has been removed then
26             if $len_{prev\_rem} == length(f_{prev})$ then
27                 $num\_rem \leftarrow 1/(1/num\_rem + 1)$;  // Reduce max. number to avoid
                    deadlock
28             end
29             $len_{prev\_rem} \leftarrow length(f_{prev})$;
30         end
31     end
32 end
33 $f_{final}$ $\leftarrow$feat_lowest_TS_score();
```



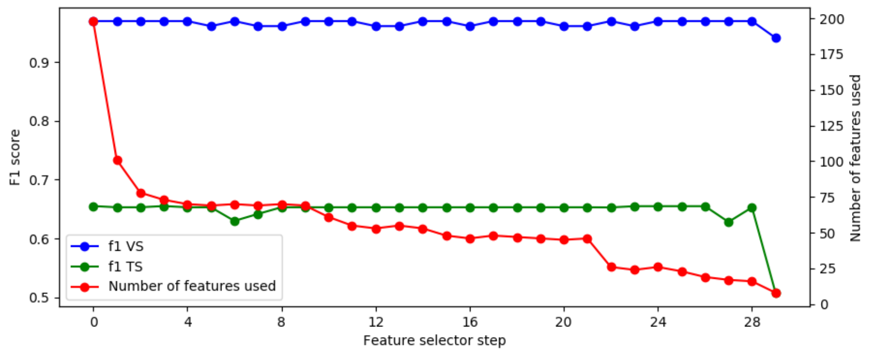
(a) $\{Previous, Current, Next\} = \{0, 1, 0\}$: best score at step 12 with F1 = 0.72 and 24 features



(b) $\{Previous, Current, Next\} = \{1, 1, 0\}$: best score at step 27 with F1 = 0.68 and 12 features



(c) $\{Previous, Current, Next\} = \{0, 1, 1\}$: best score at step 1 with F1 = 0.72 and 135 features



(d) $\{Previous, Current, Next\} = \{1, 1, 1\}$: best score at step 28 with F1 = 0.66 and 16 features

Figure I.7 – Example of a feature selection process following the Algorithm 2 using n numbers of previous and next timesteps for the classification, with $n \in \{0, 1\}$. The F1 score is a metric computing the classification accuracy (cf. Section 7).

6 Last module: the model

The model is the main module of the solution for the time series segmentation into classes for each timesteps. Its role is to take a production dataset, which can potentially contain missing data, if the model can handle it, and classify each of the timestep into one of the available classes. The model has to be flexible to be applicable to a wide range of datasets. It should also be easily interpretable to allow engineers to understand the model and gain insights of which signals can lead to which states and how.

Two tree-based models are considered for this task: the random forest classifier and the XGBoost classifier. Both profit from the easy interpretation of tree-based models, which is the main reason why they are considered. Moreover, the idea of the handcrafted rules method used by WideTech (cf. Section 6.1) is close to decision trees whose main idea is to have a decision at each leaf and a predicate[1] at each node allowing the decision process to go to the left or right branch depending on the value of the predicate. By going through the tree, it leads to the decision to make. They also have the advantage of providing the features importance which can be used directly by the feature selector algorithm explained earlier (cf. Section 5). A feature importance is a value, usually between 0 and 1, assigned to each feature which indicates how useful the feature is for the classification. The random forest classifier and the XGBoost classifier are explained in this section and are later evaluated to choose the most accurate one (cf. Section 8).

6.1 WideTech classification model

The classification method used by WideTech is based on boosting, i.e. using many weak estimators to create one strong one, and handcrafted rules. For each sensor of the production process, one estimator is created. The latter is limited to the signals measured by its sensor. If required, other estimators combining signals from different sensors are also constructed. Then each estimator has one classification vote. When the model has to classify a timestep, each estimator gives its vote, then all the votes are weighted to obtain a score for each class[2]. Finally, the classification follows the class with the highest score. However, if the highest score is below a certain threshold, the classification is replaced by an unknown state.

This voting system has two main advantages. Firstly, it is robust to the loss of one sensor. Secondly, it brings a notion of indecisiveness, when estimators vote for different classes.

Each estimator is built by assigning to each class a set of rules such as $A < B < C$ where A and C are fixed values and B is the average, gradient or standard deviation of one of the signals measured by the sensor. If a set of rules is verified then the corresponding class is the one the estimator will vote for. The disadvantages of this solution are the time required to manually tune the rules for each class and the fact that it is difficult for a person to perfectly think of all situations when redacting rules.

6.2 Random forest classifier

This section describes the random forest model and is based on [Hastie, 2016]. Random forests can be used for classification or regression, however, as this work's task is classification, the description below only concerns the classification.

[1]A predicate is an affirmation which can be either true or false.
[2]The weighting system is intentionally not explained to preserve its secrecy.

Classification tree: Classification trees are the foundation of random forests. The idea is to have a tree structure where each leaf corresponds to a class and each internal node to a splitting criteria. For example, on the task of classifying triangles, equilateral triangles and squares based on three variables: the area, the number of sides and whether all sides are equal (boolean equal to 1 if true or 0 if false), a simple tree of depth 2 can be applied (cf. Figure I.8). The classes are exclusive, an equilateral triangle, while still a triangle should only be classified as being an equilateral triangle in this example. With three leaves corresponding to the three classes and two internal nodes respectively with the predicates 'number of sides $<$ 4' and 'all sides are equal $<$ 1' shown in the figure, each sample is perfectly classified.

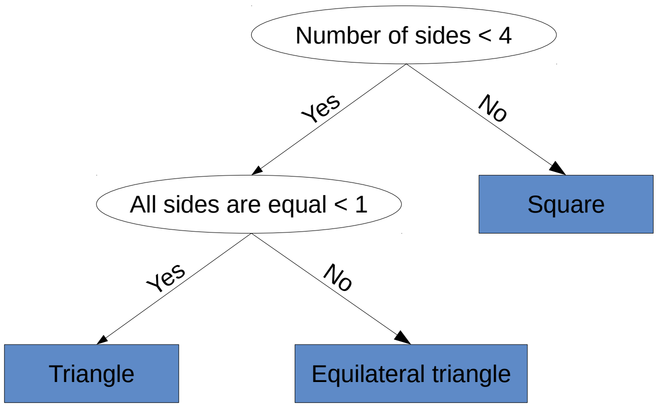


Figure I.8 – Simple classification tree for the classes 'Square', 'Triangle' and 'Equilateral triangle' based on the features 'area', 'number of sides' and 'all sides are equal'. The classes are exclusive, an equilateral triangle, while still a triangle should only be classified as 'Equilateral triangle'.

To grow a tree, the algorithm has to decide the splitting features, their split points and the shape of the tree. Let the data consists of N observation each with p features and one class: (x_i, y_i) for $i = 0, 1, 2, ..., N-1$, with $x_i = (x_{i,0}, x_{i,1}, ..., x_{i,p-1})$ being the features of the i-th observation and y_i being its class. Let T be a tree dividing the dataset into M regions (with M leaves); the method to construct the tree is explained later. The goal is to label each leaf to minimize the function:

$$p(L, y_L) = \frac{1}{N} \sum_{i=0}^{N-1} 1_{(x_i \in L \ \& \ y_i = y_L)}, \tag{I.7}$$

$$loss = 1 - \sum_{j=0}^{M-1} p(L_j, y(L_j)), \tag{I.8}$$

with L_i, $i \in [1, M]$, being the i-th leaf and $y(L)$ being the value of the leaf L. To label each region, it takes the class in majority among the ones contained in the corresponding leaf. This indeed minimize the loss function (I.8).

Now to grow the tree and find the partition, it is often computationally infeasible to find the one minimizing the loss. Instead a greedy algorithm is used. Starting from all the data, for each feature f and split point s, it defines:

$$R_1(f, s) = \{X | X_f \leq s\} \quad \text{and} \quad R_2 = \{X | X_f > s\}. \tag{I.9}$$

Then the best feature f and split point s is found by maximizing:

$$\max_{f,s}\Big(\max_{y_1}(loss\_leaf(R_1, y_1)) + \max_{y_2}(loss\_leaf(R_2, y_2))\Big). \tag{I.10}$$

Once the splitting criterion is determined, the data is divided into the new regions and the same process is applied on them. Usually the process per node is stopped when some minimum node size is reached and this large tree is then pruned using cost-complexity pruning. Pruning a tree is the action of replacing internal nodes and their daughters by leaf nodes to reduce the complexity of the tree. Let $T \subset T_0$ be the relation such that T is a subtree of T_0 which can be obtained by pruning T_0, and let $|T|$ be the number of leaf nodes in T. Then, with a node m representing a region R_m, the cost complexity criterion is:

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T| \tag{I.11}$$

where $N_m = \#\{x_i \in R_m\}$, $\alpha \geq 0$ and Q_m is the impurity measure. The goal is to find the subtree $T_\alpha \subseteq T_0$ to minimize $C_\alpha(T_0)$. The tradeoff between how well the tree fits the data and tree size is controlled by α; if $\alpha = 0$ the pruning returns the non-pruned tree. The subtree T_α is constructed by producing an ensemble of subtrees from T to a single node tree by iteratively collapsing the internal node which produces the smallest increase in $\sum_{m=1}^{|T|} N_m Q_m(T)$. Then, the tree minimizing (I.11) out of those in the ensemble is T_α. Let $y(R_m)$ be the class of node m and $\hat{p}_{m,k} = \frac{1}{N_m}\sum_{i=0}^{N-1} I(x_i \in R_m \;\&\; y_i = k)$, then the impurity measures for classification could be:

$$\text{Misclassification error:}\quad Q_m(T) = 1 - \hat{p}_{m,y(R_m)}, \tag{I.12}$$

$$\text{Gini index:}\quad Q_m(T) = \textstyle\sum_{k\neq k'} \hat{p}_{m,k}\hat{p}_{m,k'}, \tag{I.13}$$

$$\text{Cross-entropy:}\quad Q_m(T) = -\textstyle\sum_{k=1}^{K} \hat{p}_{m,k}\log\hat{p}_{m,k}. \tag{I.14}$$

Classification random forest: The random forest model idea is to build a large number of non-correlated trees and average them for predictions. The averaging improves the results as the trees are noisy.

The implementation of random forest and its application to the classification task are defined in Algorithms (3, 4).

Algorithm 3. Random forest instantiating

```
1  Create an empty ensemble of trees {};
2  for $b = 1$ to $B$ do
3      Draw a bootstrap sample[a] of fixed size from the learning
        set;
4      Initialize the tree $T_b$ with a unique node;
       // Grow a tree $T_b$ fitted to the previously sampled data
5      for Each terminal node of $T_b$ do
6          if Node size n_min is not reached then
7              Select $m$ features at random from the $p$ ones;
8              Pick the best split-point among the $m$;
9              Split the node into two daughter nodes;
10         end
11     end
12     Add $T_b$ to the ensemble of trees;
13 end
14 Output the ensemble of trees $\{T_b\}_1^B$;
```

[a] A bootstrap sample from a set is a set of observations repeatedly drawn with replacement from the original set.

Algorithm 4. Random forest classification

```
1  Instantiate a random forest $\{T_b\}_1^B$;
   // Let $\hat{C}_b(x)$ be the class prediction of the $b$th tree
2  $\hat{C}(x) = \text{majority vote}\{\hat{C}_b(x)\}_1^B$;
```

In this study: The random forest tested in this solution is implemented using the RandomForestClassifier of SKLEARN. It has 10000 estimators, no maximal depth, at least 2 samples per leaf and no pruning.

6.3 XGBoost classifier

This section describes the XGBoost model and is based on [Chen, 2017]. As for the random forest, only the classification matters in this task. However, the classification is not explained in the released paper. Thus the regression is first explained in this section, then a deduction about the inner working of the classification is done based on the papers referenced in [Chen, 2017]: [Friedman et al., 2000] and [Friedman, 2001].

Regression: The data stays the same as in Section 6.2, thus N observations each with p features and one class: (x_i, y_i) for $i = 0, 1, 2, ..., N-1$, with $x_i = (x_{i,0}, x_{i,1}, ..., x_{i,p-1})$

being the features of the i-th observation and y_i being its class.

The model uses K additive functions to predict the output:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i) \tag{I.15}$$

where function $f_k(x)$ is the prediction of x done by the k-th regression tree from the ensemble. Unlike in classification trees, the leaves of regression trees contain a continuous score instead of a class label.

Then to learn the set of functions, the following equation is minimized:

$$loss = \sum_{i=0}^{N-1} l(\hat{y}_i, y_i) + \sum_{k}(\alpha|T_k| + \frac{1}{2}\beta||w_k||^2) \tag{I.16}$$

where l is a differentiable convex loss function, $|T_k|$ is the number of leaves in the tree of function f_k and w_k is the score of each leaf of the tree T_k. While the first term of the equation penalizes the accuracy of the regression, the second one penalizes the model complexity. To minimize the equation (I.16), instead of using traditional optimization methods, the model is trained in an additive greedy manner. Formally, at the t-th iteration, a new function f_t is added such as to minimize:

$$loss^{(t)} = \sum_{i=0}^{N-1} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \alpha|T_t| + \frac{1}{2}\beta||w_t||^2 \tag{I.17}$$

where $\hat{y}_i^{(t)}$ is the prediction for the regression on x_i at the t-th iteration. Let $g_i = \delta_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ and $h_i = \delta^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$, then to optimize the objective, second-order approximation can be used

$$loss^{(t)} \simeq \sum_{i=0}^{N-1} \Big(l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)\Big) + \alpha|T_t| + \frac{1}{2}\beta||w_t||^2 \tag{I.18}$$

Removing the constant term, the objective becomes:

$$\tilde{loss}^{(t)} = \sum_{i=0}^{N-1} \Big(g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)\Big) + \alpha|T_t| + \frac{1}{2}\beta||w_t||^2 \tag{I.19}$$

Let q be the decision rule in the tree such that the instance set of leaf j is $I_j = \{i|q(x_i) = j\}$. As $f(x) = w_{q(x)}$, the equation (I.19) can now be rewritten:

$$\tilde{loss}^{(t)} = \sum_{i=0}^{N-1} \Big(g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)\Big) + \alpha|T_t| + \frac{1}{2}\beta \sum_{j=0}^{|T_t|-1} w_j^2 \tag{I.20}$$

$$= \sum_{j=0}^{|T_t|-1} \Big((\sum_{i\in I_j} g_i)w_j + \frac{1}{2}(\beta + \sum_{i\in I_j} h_i)w_j^2\Big) + \alpha|T_t| \tag{I.21}$$

Thus for a fixed tree, $q(x)$ is fixed too and the optimal weight w_j^* of leaf j is:

$$w_j^* = -\frac{\sum_{i\in I_j} g_i}{\beta + \sum_{i\in I_j} h_i} \tag{I.22}$$

Using this, the loss for a given tree can be computed:

$$\tilde{loss}^{(t)}(q) = \alpha|T_t| - \frac{1}{2}\sum_{j=0}^{|T_t|-1} \frac{(\sum_{i\in I_j} g_i)^2}{\beta + \sum_{i\in I_j} h_i} \tag{I.23}$$

The equation (I.23) can measure the quality of a tree structure and is like the impurity score for evaluating decision trees. As it is often impossible to carry out an exhaustive search of tree decision rule q, a greedy algorithm is used instead. It starts from a single leaf and iteratively adds branches and evaluates them with equation (I.23).

Classification: The classification is not explained in the XGBoost paper, however, based on its references [Friedman et al., 2000] and [Friedman, 2001], it is possible to deduce a probable implementation.

First, in a two classes problem, the model still uses K additives functions but instead of predicting the output as in equation (I.15), it approximates the following:

$$F(x_i) = \sum_{k=1}^{K} f_k(x_i) \simeq \log\frac{p(y_i = 1|x_i)}{p(y_i = 0|x_i)} \tag{I.24}$$

With $F(x)$ it is now possible to approximate the $p(y = 1|x)$ and $p(y = 1|x)$.

$$\begin{cases} p(y_i = 1|x_i) = \frac{\exp F(x_i)}{1+\exp F(x_i)} \\ p(y_i = 0|x_i) = 1 - p(y_i = 1|x_i) = \frac{1}{1+\exp F(x_i)} \end{cases} \tag{I.25}$$

The predicted class is the one with the highest probability.

Then, in the multi-class problem, each class c has its own F_c and thus its own estimators. So instead of having K trees constructed, there are numbered $K \times C$ with C the total number of classes. The equations become:

$$F_c(x_i) = \sum_{k=1}^{K} f_{c,k}(x_i) \tag{I.26}$$

$$p(y_i = c|x_i) = \frac{\exp F_c(x_i)}{\sum_{c'=1}^{C} \exp F_{c'}(x_i)} \tag{I.27}$$

The probabilities in equation I.27 are used to predict the class. Only the loss function is left to explain. It is the cross-entropy:

$$l(y_i, \hat{y}_i) = -\sum_{c=1}^{K} 1_{(y=c)} \log(p(y_i = c|x_i)) \tag{I.28}$$

XGBoost approximate split finding: As explained earlier, XGBoost makes an iterative greedy search to find the tree structure. It consists in enumerating all the possible splits on every possible feature. This easily becomes computationally demanding, especially in case of contiguous features. Thus, instead of an exact greedy algorithm, XGBoost also proposes to use a faster approximate algorithm.

To summarize, the idea is to use percentiles of feature distribution to get candidate splitting points. Then those points split the continuous features into buckets, the algorithm aggregates the statistics and uses it to find the best solution.

XGBoost missing data handling: In many real-world problems, missing data appears in the dataset. XGBoost algorithm is aware of the missing pattern in the data. This is done by adding a default direction in each tree node. If the features used to split the node are missing in a sample, the sample goes to the default direction. The default direction for each node is chosen by the algorithm depending on the data. This allows the algorithm to use the fact of a feature being missing as information itself and it has the advantage of not having to impute missing data.

In this study: The model tested comes from the XGBoost python package. It is a XGBoost classifier with an approximate greedy algorithm as production datasets are usually large. It has 20 prediction functions at most, 10000 estimators, a tree depth of $\log_2(n_c)$ with n_c the number of classes, at least 1 sample per leaf, 80% of the features used per tree and 80% out of those used per node.

6.4 Module final implementation in this study

The models considered are the random forest classifier and the XGBoost classifier. However, the final version of the module is implemented with the XGBoost classifier. This model is preferred over the random forest as it showed better results in the experiment explained in Section 8.

7 Imputation, feature selection and classification evaluation metrics

This section describes the metrics used during this task, for the data imputation, the feature selection and the classification. A regression metric evaluates the quality of the missing values imputation, whereas a classification metric is used for the classification models. The root mean square error and F1 score are respectively chosen as regression and classification metrics.

F1 score:

The multi-class F1 score is based on the F1 scores of two classes. It is thus easier to start with it.

Let *positive* and *negative* be the two classes. The notions of true positive (TP), false positive (FP), true negative (TN) and false negative (FN) are defined in Table I.2.

		True class	
	Total number	number of *positive*	number of *negative*
Classification	number of *positive*	TP	FP
	number of *negative*	FN	TN

Table I.2 – Definition of true positive (TP), false positive (FP), true negative (TN) and false negative (FN).

With the help of the previous notions, it is possible to define two metrics: precision and recall. *Precision* of the *positive* class evaluates how often an object classified *positive* has the correct class while *recall* of the *positive* class evaluates how often a *positive* object has been classified *positive*. Formally:

$$precision = \frac{TP}{TP + FP}. \tag{I.29}$$

$$recall = \frac{TP}{TP + FN}. \tag{I.30}$$

The F1 score is a combination of *precision* and *recall* and its measure for a class A benefits from penalizing how often an A object has been wrongly classified and also how often the A classification has been wrong. The F1 score is defined as follows:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}. \tag{I.31}$$

As $precision, recall \in [0, 1]$, $F_1 \in [0, 1]$ with 1 being a perfect classification of the *positive* and *negative* classes.

In the multi-class case, for each class c, a F1 score is computed considering 2 classes only: c and 'anything but c'. Then the final F1 score is the weighted (by the number of true instances for each label) average of the score of each class.

Root mean square error: The root mean square error ($RMSE$) is a regression metric computed with equation:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{T}(y_i - \hat{y}_i)^2}{T}} \tag{I.32}$$

The root mean square error is such that $RMSE \in [0 + \infty[$, with 0 corresponding to no error.

8 Testing and choosing the model and imputation module

Previously in Sections 4 and 6, two methods and two models were introduced respectively as imputation methods and classification models: the k-nearest neighbors, the Bayesian Ridge regression, the random forest and XGBoost. A choice has to be made for the general automated solution. To do this, an experiment is conducted to evaluate the alternatives.

As XGBoost handles missing data, there are three possible configurations: a random forest with an iterative imputation by k-nearest neighbors, a random forest with iterative imputation by Bayesian Rigde regression or a XGBoost. The evaluation is done on the F1 score of the classification (cf. Section 7) and the RMSE of the imputed data (cf. Section 7).

8.1 Experimental setup description

Data: The data is the dataset ds1 (cf. Section 2.2). There is a total of 11 signals each having 6 measures: average, standard deviation, gradient, interpolation, minimum and maximum. All features are used by the model. Each feature is computed on the raw data observed during periods of 15 minutes, thus making time series with a 15-minute-long interval. Part of the dataset can be seen in Figure I.9.

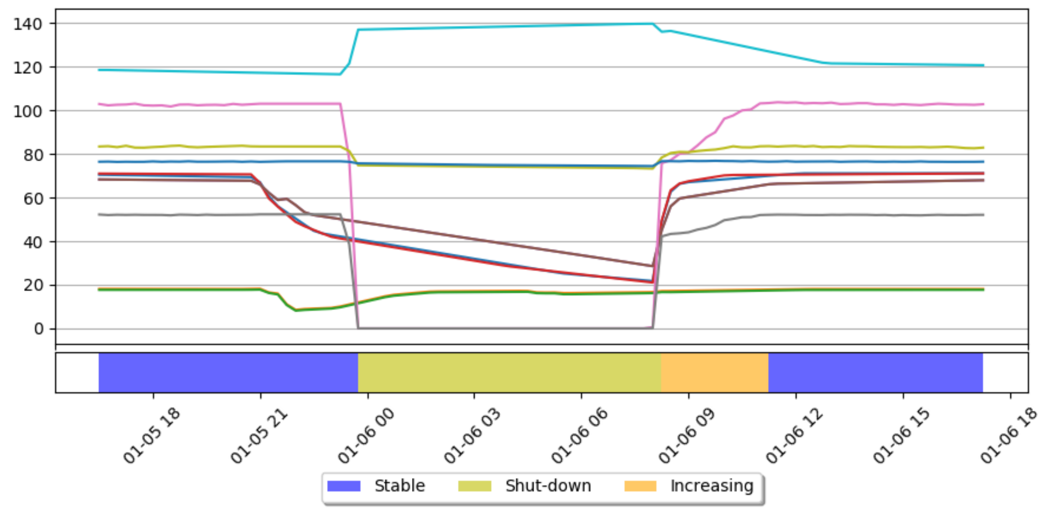


Figure I.9 – Subsample of the datasets ds1. It displays the average of the signals on top of the classification.

There is a total of 8734 timesteps. The division in LS, VS and TS is done with a $(\frac{1}{2}, \frac{1}{4}, \frac{1}{4})$ ratio by dividing the time series at its half and its three fourth length. Each timestep is classified by one out of three classes: stable, increasing or shutdown (cf. Table I.3 for the class distribution).

	LS	VS	TS	
Stable	4232	2026	2172	8430
Shutdown	88	141	5	234
Increasing	47	16	7	70
	4367	2183	2184	8734

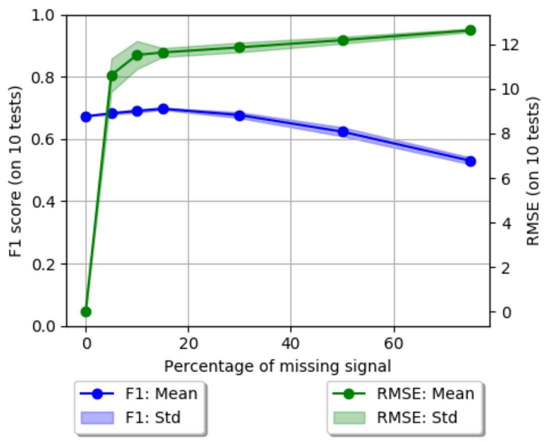
Table I.3 – Class distribution within the datasets

Models: Two models are tested: a random forest with a data imputation and a XGBoost classifier. On the one hand, the random forest is implemented using the RandomForest-Classifier of SKLEARN. It is composed of 10000 estimators and has no maximal depth. This model handles missing data with a data imputation by iterative imputation based on a k-nearest neighbors or a Bayes Ridge regression. The number of iterations is limited to $20 + n$ with n being the percentage of missing data (cf. Experimental setup below). On the other hand, a classifier from the XGBOOST python package is used. It has an approximate greedy algorithm, 20 parallel trees at most, 10000 estimators, a tree depth of $\log_2(n_c)$ with n_c the number of classes, 0.8 of the features used per tree and 0.8 out of those used per node.

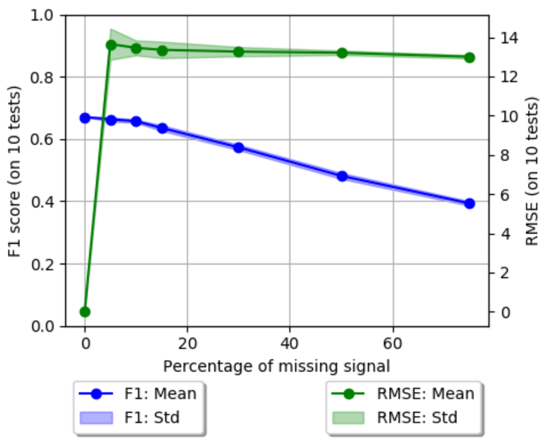
Experimental setup: For each model, a classifier is trained on the LS containing no missing data. In the case of the random forest, the data imputation is also based on this set. Then for each number n in [0, 5, 10, 15, 30, 50, 75], n percentage of missing signals are artificially and randomly induced in VS. VS is later used by the model to evaluate its classification (after having imputed the missing data for the random forest). This evaluation is based on the root mean square error (RMSE) of the imputed data but also on the F1-score of the classifier. This process is Finally, reproduced 10 times and the means and standard deviations are drawn (cf. Figure I.10).

8.2 Experiment results

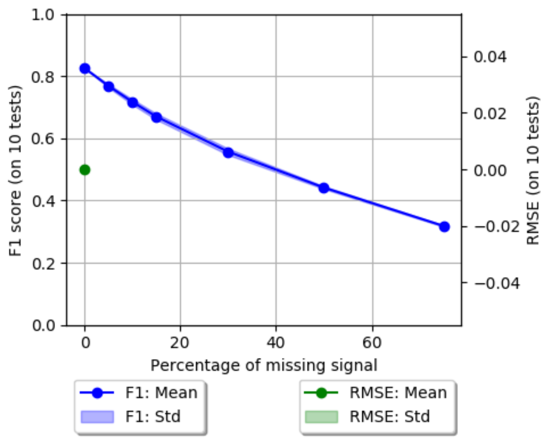
The results on Figure I.10 show the root mean square error (RMSE) of the imputation, if required by the model, and the F1 score of the classification on the data, potentially after imputation, as the number of missing data increases.



(a) Iterative imputation: k-nearest neighbors ($k = 15$)



(b) Iterative imputation: Bayesian Ridge regression



(c) No imputation: XGBoost classification

Figure I.10 – RMSE on the task of data imputation and F1-score weighted uniformly on each class on the task of classification of timestep of time series with missing data. The input data for a timestep is composed of signals (temperature, flow, density, etc.) with multiple measurements (average, gradient, minimum, max, etc.). If a signal is missing at a timestep t then all measurements of this signal are missing in t. There are no missing data in the set used for learning the imputation rules and the model. (a) Missing data is imputed by iterative imputation with k-nearest neighbors. (b) Missing data is imputed by iterative imputation with Bayes Ridge regression. (c) Missing data is not imputed and is handled by the XGBoost classifier.

Firstly, comparing the two regression methods k-nearest neighbors and Bayesian Ridge regression, the latter displays the worst score. Indeed, even if the error does not increase with the increment of missing data percentage, its lowest value is worse than the highest error of the k-nearest neighbors. Especially at a low level of missing data (5%), the k-nearest neighbors performs about 15% better than the Bayesian Ridge regression. Furthermore, the comparison of the F1 score for those two settings corroborates this analysis as the score of the k-nearest neighbors stays higher than its counterpart at every point.

Secondly, the two models, the random forest and XGBoost classifier, can also be compared using the F1 score. The random forest classifier with data imputation presents a higher robustness to the increase in missing data. The F1 score worsens only by 0.2 for the random forest while the XGBoost decreases by 0.5 when the missing data percentage goes from 0 to 70%. Nevertheless, as explained earlier, the expected number of missing data is low and if the comparison is focused on a percentage lower than 20%, XGboost consistently shows a better score. Moreover, it can be expected for the score to increase in XGBoost if the missing data was missing at random or missing not at random instead of missing completely at random.

8.3 Model choice based on experiment results

The k-nearest neighbors imputation method has got a better evaluation than the Bayesian Ridge regression on this experiment. However, XGBoost proved itself to be more accurate than random forest. XGBoost is thus chosen as the final model for the task. In this regard, the data imputation module becomes useless and is deactivated.

9 Evaluation of the solution

With the experiment from the previous section, the final architecture of the algorithm has been decided. However, its capabilities are yet to be evaluated. This section tests the algorithm in the task of time series classification. There is no specific goal value that should be reached. Instead, this section provides insights on its accuracy and its requirements in terms of data annotation. Those two aspects can then be used to determine whether this solution provides results worth of the time spent on the data annotation.

9.1 Experimental setup description

Generalities: Two different configurations for the algorithm are evaluated: one using the feature selector and one without the feature selector but with a set of features manually chosen by an engineer. It is important to note that the engineer who made the selection is also the one who annotated the data. Thus the set of features can be considered as the best possible one from an engineer point of view.

Data: The data is the dataset ds2 (cf. Section 2.2). There is a total of 49 signals each having 6 aggregations: average, standard deviation, gradient, interpolation, minimum and maximum. Each feature is computed on the raw data observed during periods of 1 hour, thus making time series with a 1 hour long interval. Part of the dataset can be seen in Figure I.11.

In the configuration without the features selector, only a fixed set of 6 signals is used out of the 49 existing ones.

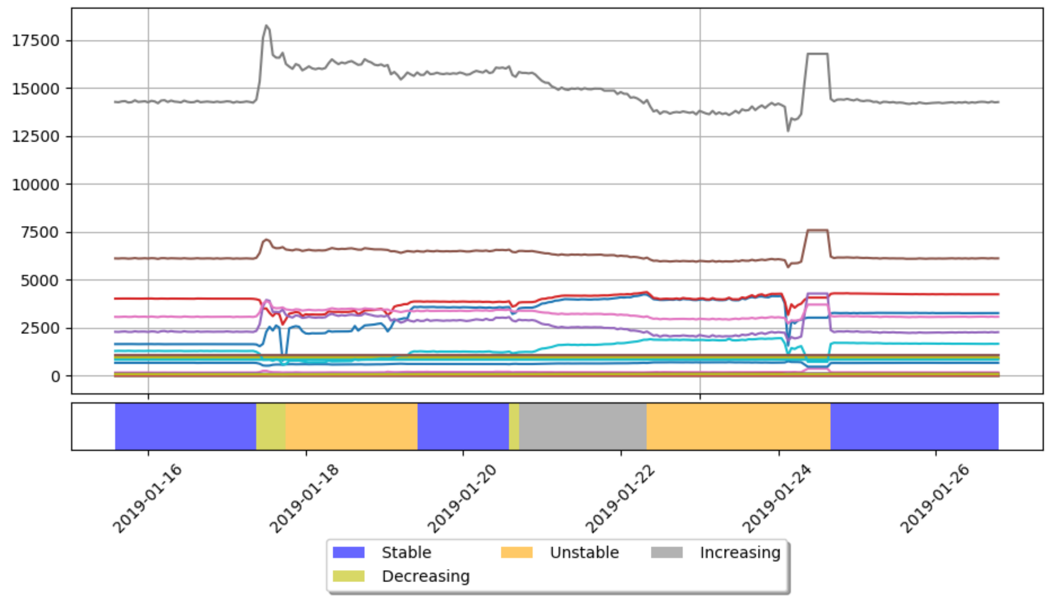


Figure I.11 – Subsample of the datasets ds2. It displays the average of the signals on top of the classification.

There is a total of 8714 timesteps. The division in LS and VS is done with a $(\frac{1}{4}, \frac{3}{4})$ ratio by dividing the time series at its one fourth length. Each timestep is classified by one of six classes: stable, unstable, increasing, decreasing, flat or shutdown (cf. Table I.4 for the class distribution).

	LS	VS	
Stable	1503	4504	6007
Unstable	520	334	854
Increasing	99	255	354
Decreasing	35	84	119
Flat	0	70	70
Shutdown	21	1289	1310
	2178	6536	8714

Table I.4 – Class distribution within the datasets

Configurations: As explained before, two configurations are tested: one with a feature selector and one with a manually fixed set of features. However, each configuration is trained in two different ways. One with weighted samples to compensate the unbalanced class distribution in the dataset (cf. Table I.4) and one without weights. Both have advantages on their own as while weighting samples should improve the average F1 score, using raw samples should improve the weighted average F1 score. They will be referred by four configuration tags:

- *Configuration 1 :* manual features selection and training with raw samples,
- *Configuration 2 :* automatic selection with the features selector and training with raw samples,
- *Configuration 3 :* manual features selection and training with weighted samples,
- *Configuration 4 :* automatic selection with the features selector and training with weighted samples.

Configurations 2 and 4 also have to decide how many previous and next timesteps to use for the classification. The maximal number of previous and next timesteps has been arbitrarily fixed to 1. On the other hand, configurations 1 and 3 only use the current timestep as per the recommendation of the engineer who gave the features.

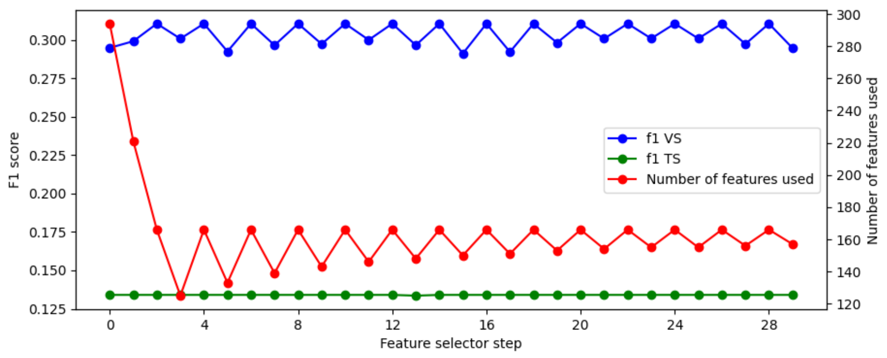
The model used is a classifier from the XGBoost python package. It has an approximate greedy algorithm, 20 parallel trees at most, 10000 estimators, a tree depth of $\log_2(n_c)$ with n_c the number of classes, a minimal of 1 sample per leaf, 80% of the features used per tree and 80% out of those used per node. During training, only $\frac{3}{4}$ of LS is used, what is left is used as a validation set for early stopping. After 10 iterations without improvement on the accuracy on the validation set for early stopping, the training stops. Early stopping reduces the chance of overfitting LS.

Experimental setup: The evaluation is done in two ways. The first one is a classic division of the dataset in two: LS and VS. Then the algorithm is trained on LS and is evaluated on its classification of VS using the F1 score as metric. For the second test, a cross-validation is applied on the VS with an increasing number of blocks, from 2 to 20. The LS is left untouched by configurations 1 and 3 during the cross-validation but it is used by configurations 2 and 4 to select their features.

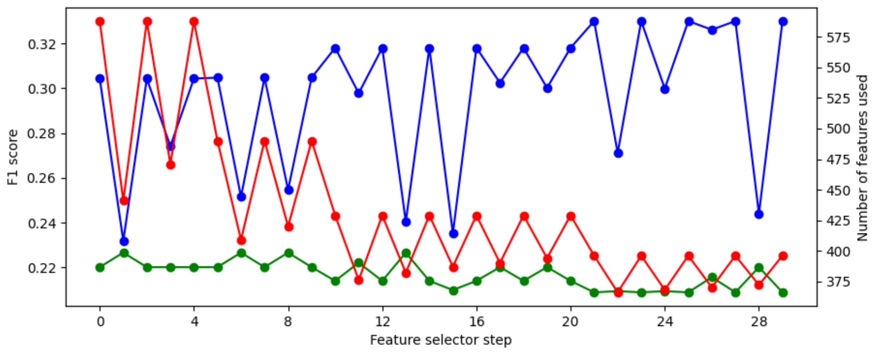
9.2 Experiment results

Firstly, configurations 2 and 4 need to select their features. As they use LS both for the simple validation test and the cross-validation, the selection is only executed once per configuration. The selectors take the dataset given (LS) and divide it in three to construct their own learning set, validation set and testing set. For each combination of features from the current, previous and next timesteps, the selectors iteratively remove features to improve the score on their validation set. At the end, the score on their own testing set is compared to find the best feature set (cf. Section 5).

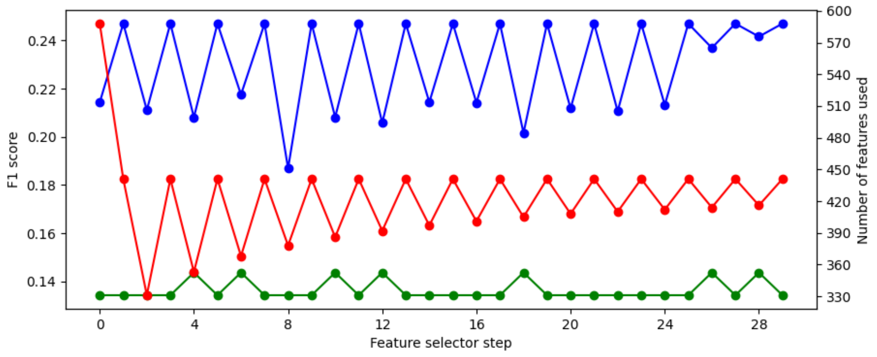
The selection process is displayed on Figures I.12, I.13 and I.14. The best F1 score is obtained with features from the current and the next timesteps for both configurations 2 and 4. Thus the sets of features used to reach this score is selected and they respectively end up with 382 and 471 features. Nevertheless, the scores reached using other combinations of features from previous, current and next timesteps are not far from their best one. In this situation, it is difficult to affirm that their respectively selected set is the best. Moreover, the F1 scores on the validation and testing set are low: respectively around 0.3 and 0.15 for configuration 2 whereas configuration 4 is around 0.3 and 0.25. It means the models fitted on the feature sets are incapable of accurately classifying data other than their own learning set. It can be caused by overfitting or by the lack of data in their own learning set, for example. In any case, the selection process does not seem very trustworthy in both configurations. The selection process might be too simplistic and a more complex and robust method might be required in this case.



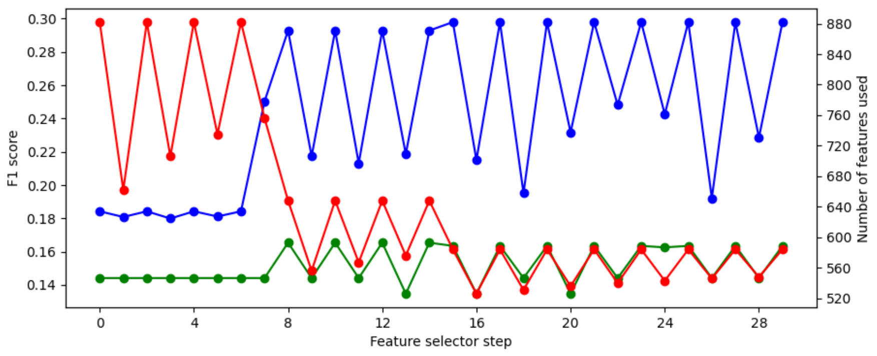
(a) $\{Previous, Current, Next\} = \{0, 1, 0\}$: best score at step 29 with F1 = 0.134 and 157 features



(b) $\{Previous, Current, Next\} = \{0, 1, 1\}$: best score at step 13 with F1 = 0.227 and 382 features

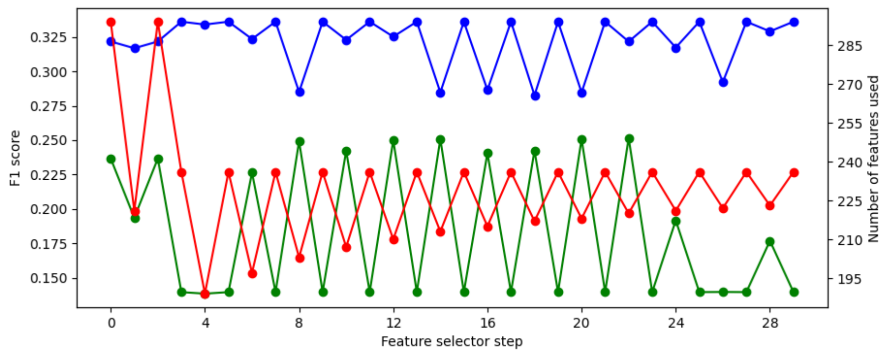


(c) $\{Previous, Current, Next\} = \{1, 1, 0\}$: best score at step 28 with F1 = 0.144 and 416 features

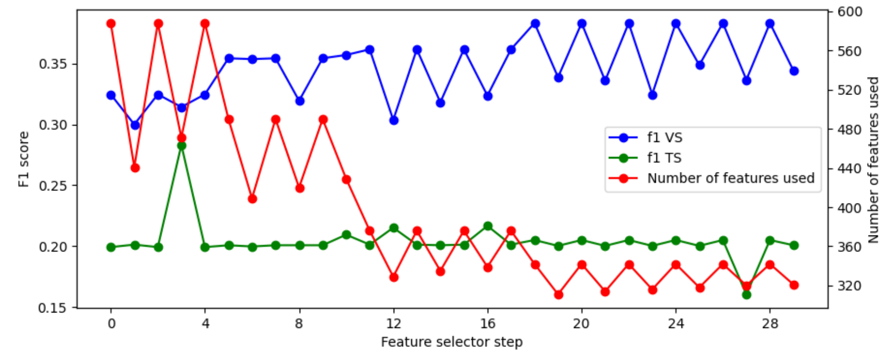


(d) $\{Previous, Current, Next\} = \{1, 1, 1\}$: best score at step 14 with F1 = 0.165 and 648 features

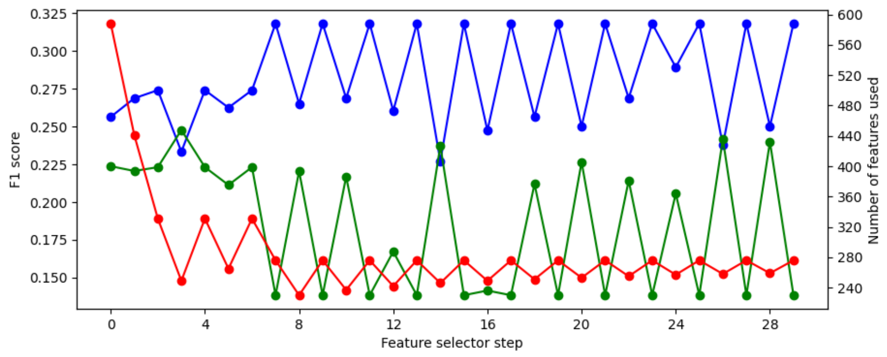
Figure I.12 – Configuration 2: Feature selection process on the learning set. The LS is itself divided in three parts: its own learning set LS, validation set VS and testing set TS. Each plot corresponds to the features selection using a different number of previous and next timesteps. The highest F1 score on the TS is obtained with 0 previous and 1 next timestep.



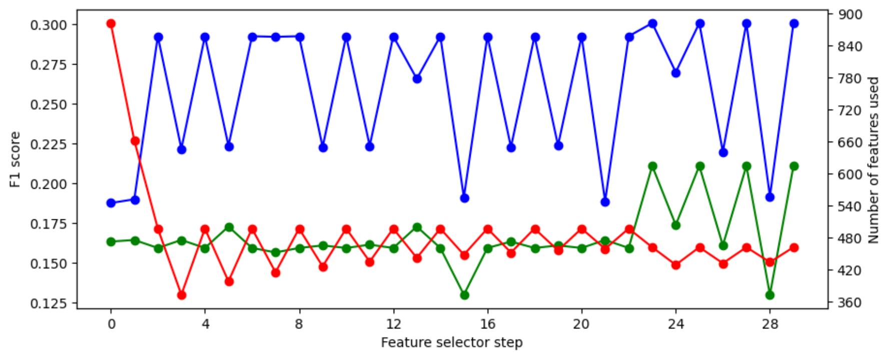
(a) $\{Previous, Current, Next\} = \{0, 1, 0\}$: best score at step 22 with F1 = 0.252 and 220 features



(b) $\{Previous, Current, Next\} = \{0, 1, 1\}$: best score at step 3 with F1 = 0.283 and 471 features

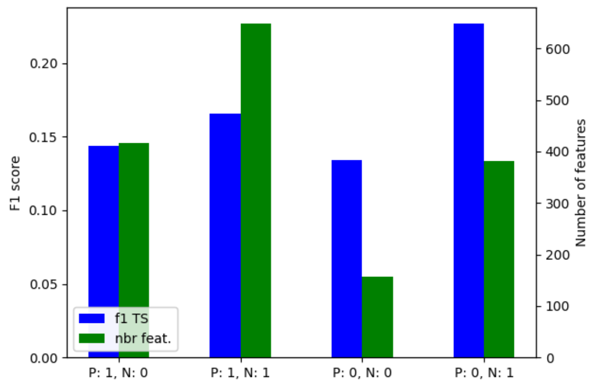


(c) $\{Previous, Current, Next\} = \{1, 1, 0\}$: best score at step 3 with F1 = 0.248 and 249 features

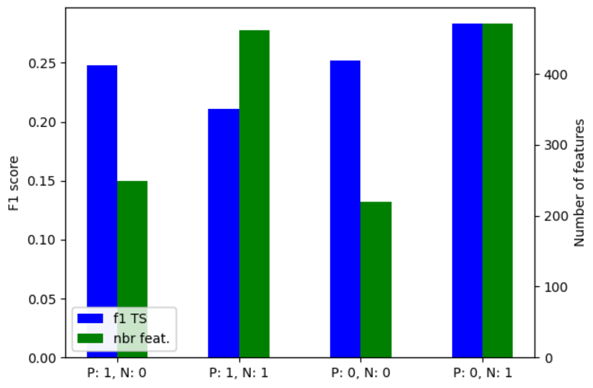


(d) $\{Previous, Current, Next\} = \{1, 1, 1\}$: best score at step 29 with F1 = 0.211 and 462 features

Figure I.13 – Configuration 4: Feature selection process on the learning set. The LS is itself divided in three parts: its own learning set LS, validation set VS and testing set TS. Each plot corresponds to the features selection using a different number of previous and next timesteps. The highest F1 score on the TS is obtained with 0 previous and 1 next timestep.



(a) Configuration 2

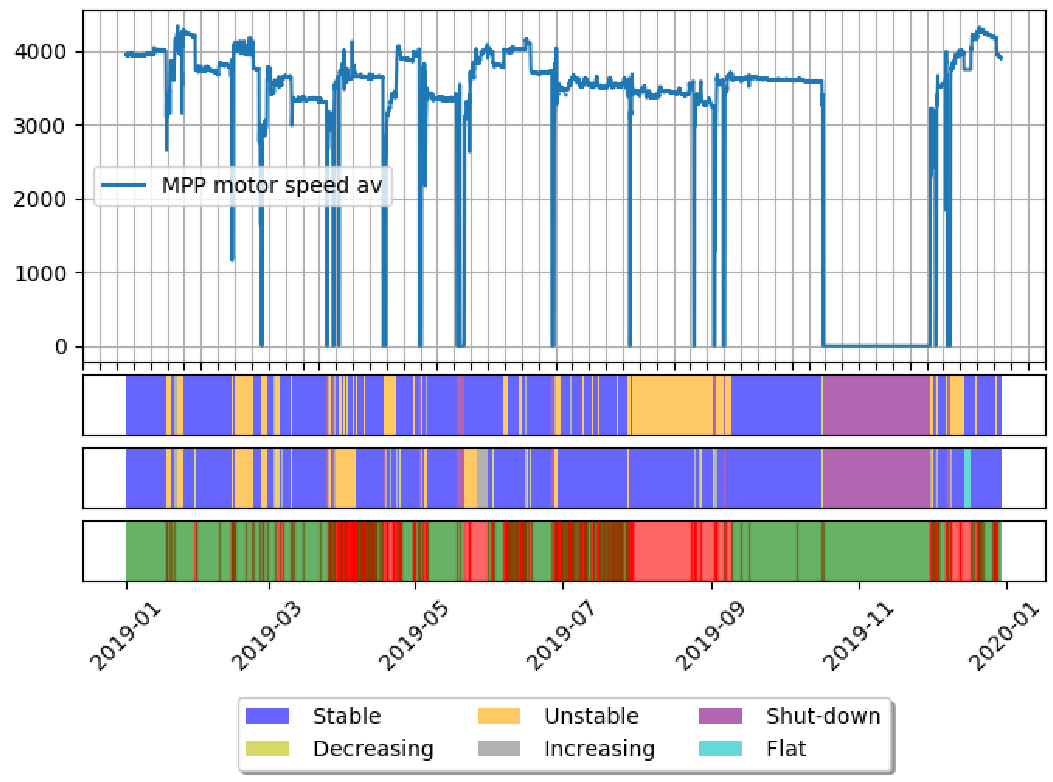


(b) Configuration 4

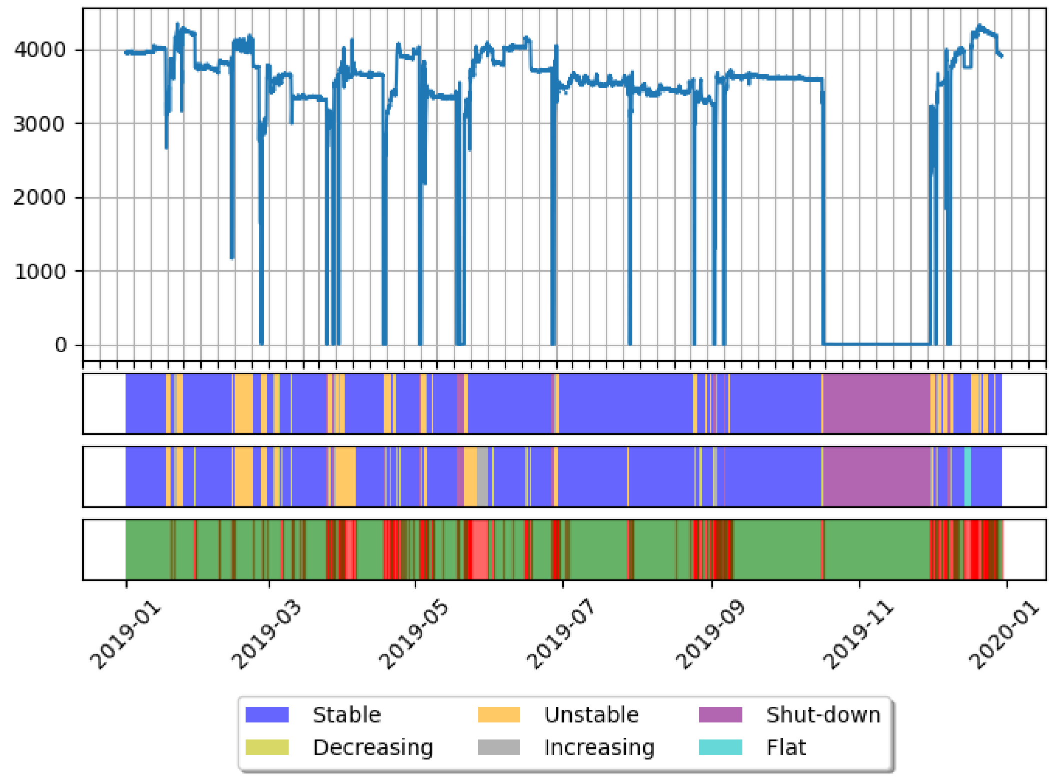
Figure I.14 – Comparison of the best scores for configurations 2 and 4 on the feature selection process on the learning set. The LS is itself divided in three parts: its own learning set LS, validation set VS and testing set TS. In each graphic, 'P' and 'N' stands for the number of previous and next timesteps from which features can be taken (in addition to the one of the current timestep). The pair of F1 score on its own TS and number of features displayed are the one with the highest F1 score.

Secondly, the configurations are tested with a validation test (cf. Figure I.15 and Table I.5). Each configuration is trained on the LS and tested on the VS. Moreover, configurations 2 and 4 use the LS to select their feature set as explained previously. Then each configuration is evaluated on its classification. First of all, the 'Flat' class is a particular case which only appears in December 2019 in the data. Consequently, it is not present in the LS , it is unknown to the algorithm and no timestep will be classified with this class. Now, comparing configurations 1 and 2, the F1 scores in the table shows very close scores between the configurations. This indicates that the selection of features didn't play a big role in the final score as the difference between the two configurations lays in their feature set: configuration 2 with 382 features (taken from the current and next timesteps) and configuration 1 with 36 features (taken from the current timestep). One big difference can be observed in the score for the 'stable' class. It is predominantly caused by the wrong classification done by configuration 1 during Augustus 2019 which can be seen in the figure. This error is reflected in the weighted average. Indeed, as the 'stable' class represents almost $\frac{7}{10}$ of the points (cf. Table I.4), the weighted average score is mainly determined by the score of the 'stable' class alone. It explains why the weighted average is so high while 4 out of the 6 classes are classified with a low accuracy. The average F1 score, however, is low for both configurations with a slightly better score in configuration 2. Thus the automatic features selection allowed to reach a higher score in this case.

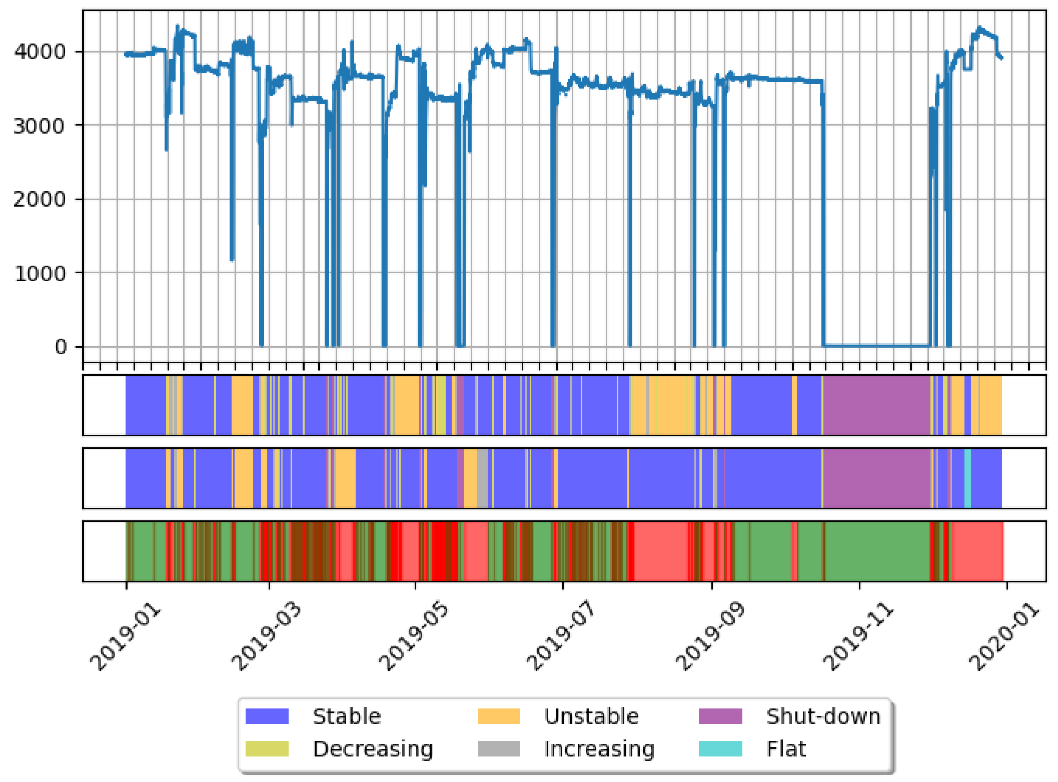
Then, configurations 3 and 4 can be compared. Configuration 3 obtains higher average and weighted average score since, even if it didn't classify the 'Stable' class as accurately as configuration 4, it reached an almost perfect score on the 'Shut-down' class. Again, the weighted average stays relatively high as it depends highly on the score of the 'Stable' and 'Shut-down' classes. Finally, comparing training with raw samples or with weighted samples, with configurations 2 and 3, the latter is more accurate. It was expected in terms of weighted average but not for the average F1 score. This can indicate that rather than the model focusing on the 'Stable' and 'Shut-down' classes because they are more present in the data, the model obtained low accuracy for the other classes because it is more difficult to classify them. All things considered, results highlight a better classification with raw samples with the features automatically selected and a high accuracy for two classes which contrasts with the low accuracy of the other classes.



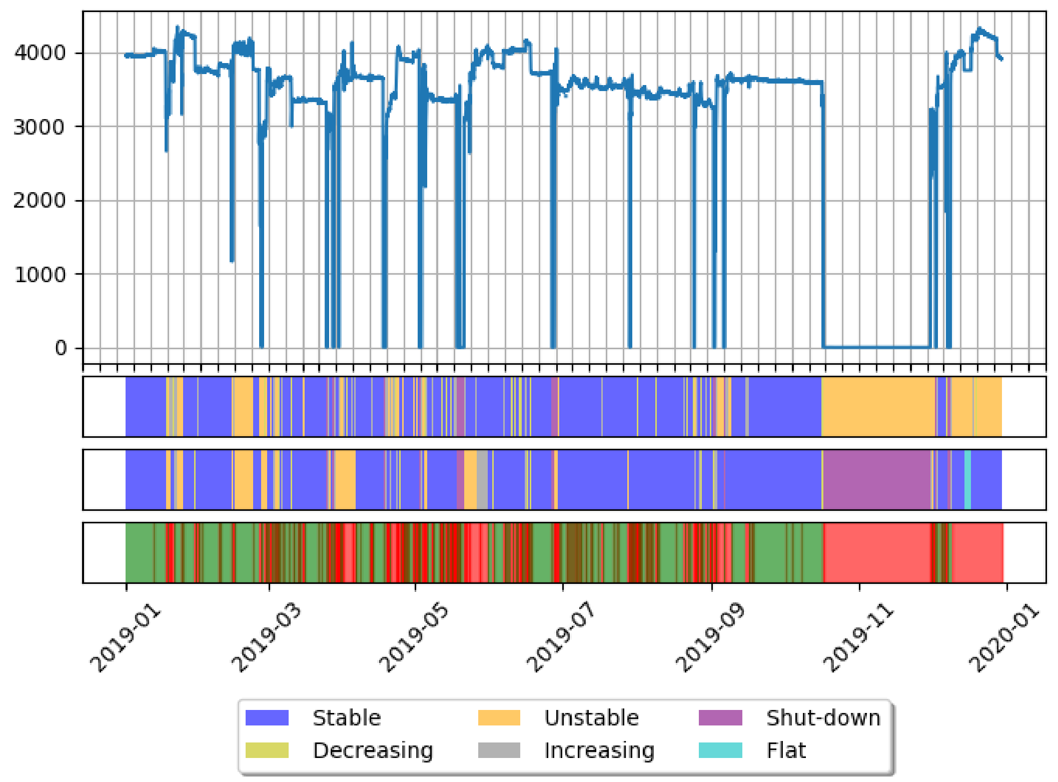
(a) Configuration 1: Model using features manually given by an engineer, trained with raw samples



(b) Configuration 2: Model using features automatically selected, trained with raw samples



(c) Configuration 3: Model using features manually given by an engineer, trained with weighted samples



(d) Configuration 4: Model using features automatically selected, trained with weighted samples

Figure I.15 – Classification of the validation set in two configurations: with or without a feature selector. The classification is done on the entire dataset (thus on LS and VS). LS is composed of 2178 points, and thus ends at the end of March 2019. Each plot is composed of 4 parts. From top to bottom, the first one shows the average of one signal (MPP motor speed) which is highly representative of the state. The other signals aren't shown to ease the reading, as there are too many signals to plot them all. The second part is the classification done by the model, while the third is the ground truth classification. Finally, the last part is a comparison between the ground truth and the classification: green if they are equal and red if they are not.

	F1 score on the validation set			
Weighting samples during training	false		true	
Features selection	manual	auto.	manual	auto.
Configuration	1	2	3	4
Class 'Stable'	0.769	**0.894**	0.698	0.837
Class 'Decreasing'	0.131	0.059	0.141	**0.172**
Class 'Unstable'	0.086	**0.183**	0.020	0.029
Class 'Increasing'	0.201	0.251	**0.342**	0.322
Class 'Shut-down'	0.984	0.981	**0.995**	0.286
Class 'Flat'	0	0	0	0
Average	0.362	**0.394**	0.366	0.274
Weighted average	0.738	**0.829**	0.694	0.649

Table I.5 – F1 score on the validation set.

Thirdly, a series of 19 cross-validations is applied on the validation set for each configuration (cf. Figure I.16). During the k-th cross-validation, the set is divided in $k + 1$ blocks. Then each configuration is trained in one block and tested on the others with a weighted average and average F1 score. This is repeated until the configurations have been trained on each block. Once it is done, the $k + 1$ results are averaged to give the scores for the k-th cross-validation. This is more reliable than the previous test, especially when the learning set is small.

In the cross-validation, configurations 1 and 3 whose feature sets are manually constructed are consistently better or equal to the other configurations. This aligns with the previous analysis of the features selection saying that the features selector was ill suited to this dataset and would not give the best set. The same two configurations have a close accuracy which indicates that weighting samples didn't have much effect. Similarly to the previous test, the weighted average F1 score is satisfactory while the average is disappointing. However, new information can be extracted from this figure. After reaching a size of 1100 timesteps, the model does not gain much from a growth in learning set. Nevertheless, as the scores are highly depending on the score on the 'Stable' class, this learning set length might be only sufficient to learn this particular class.

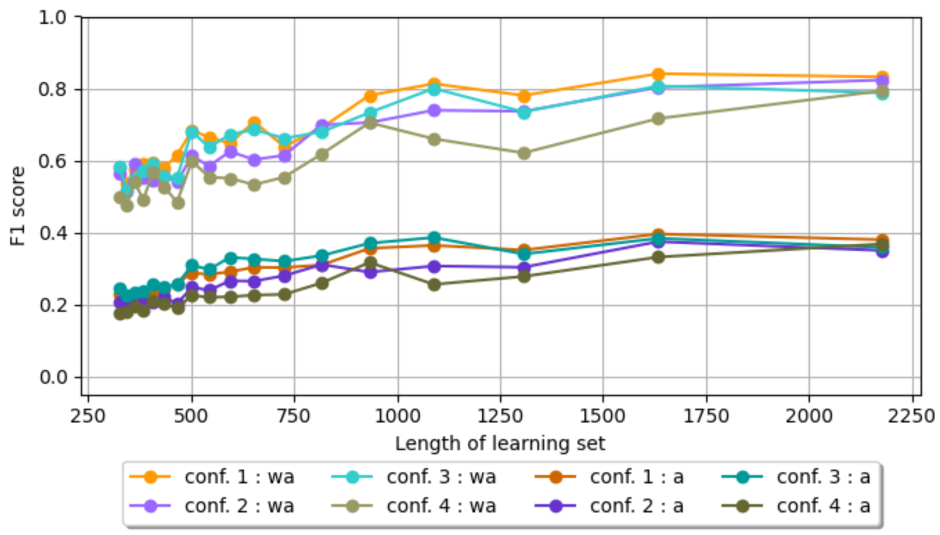


Figure I.16 – F1 score on a cross-validation, on the validation set, of decreasing number of blocks: from 20 blocks (of length 326) to 2 (of length 2178). The weighted average (wa) and average (a) F1 score are displayed for the four configurations.

In summary, the evaluation showed a high accuracy on two particular classes 'Stable' and 'Shut-down' however the other classes are not accurately classified. Still, while annotating the data, the engineer in charge of the task, struggled to classify some timesteps. For example, differentiating from an 'Increasing' state and an 'Unstable' state was sometimes difficult. Moreover, the misclassification between some classes are acceptable, like 'Increasing' and 'Unstable', as some points could have been belonged to both classes. Thus the low accuracy on the classes other than the 'Stable' and 'Shut-down' classes might be a reflect of the impossibility to categorically classify timesteps as one class rather than another. Additionally, the features selector didn't show satisfactory result in this test, however, as explained before, it stays a simplistic implementation which might not be sufficient in difficult cases such as this one. Finally, the evaluation was done on one particular dataset, while the goal is to have a general solution that could be applied to a large range of dataset. Thus the evaluation is nothing but an indicator of the performance of the model. To correctly assess the model capability, it should be tested on a larger range of datasets.

10 Conclusion

In the task of classification of timesteps of a multivariate time series representing a production system, variations of the algorithm have been tested:

- *On the model :* A random forest classifier with data imputation was opposed to a XGBoost classifier. The test demonstrates a higher accuracy for the XGBoost classifier (cf. Section 8). The experiment was only conducted on one dataset but the XGBoost classifier also benefits from the time efficiency acquired by removing the data imputation from the solution and a possible increase in accuracy if the missing data follows a pattern useful for the classification. Thus it was chosen as the machine learning model for this task.
- *On the features selection :* A manual selection was opposed to an automated tree-based features selection. Despite a not trustworthy selection made by the automatic

selector, the classification done with its feature set was more accurate than on the handcrafted one (cf. Section 9). However, the test does not generalize to a wide range of dataset, as only one was used. Thus, this result can only be taken as a performance indicator, rather than a statement. Besides, tree-based methods are robust to irrelevant/redundant features thus it was to be expected not to have a huge increase in the model accuracy.

- *On the use of weighted samples :* The unbalanced class distribution within the learning set can be compensated by weighting the samples. However doing so would decrease the average classification score weighted by the number of samples per class. Thus depending on the goal, whether an average or a weighted average score is more important, using one should be prioritized over the other. Nevertheless, the evaluation didn't reflect this as the average score didn't increase with the use of weighted samples (cf. Section 9). Higher accuracy was reached when the training was not weighted. Yet, as for the features selection, the test only covers one particular dataset, thus it cannot be fully trusted.

Firstly, in terms of accuracy, the evaluation highlighted an irregular performance. Out of the 6 classes to classify, one was impossible to predict as it never appeared in the learning set, two were classified with high precision (85-99%) and the last three were badly classified (5-25% of precision). As it was more accurate with the two classes making most of the timesteps ($> \frac{8}{10}$ of the dataset), the weighted average score is high but the average stays low. However, the three classes for which the algorithm struggles were also difficult to distinguish by the engineer making the annotation. Some timesteps could even have belonged to multiple classes. This could explain the low accuracy. Nevertheless, the goal is to have an automatic algorithm classifying a wide range of datasets. As the test is conducted on only one, it is not enough for any strong affirmations, but the test highlights the high potential of tree-based models for this task.

Secondly, for the solution to be applicable to a wide range of datasets, an hyperparameter optimization, which wasn't applied here, should be done beforehand for each dataset to segment to find the best values for the depth, number of features per node, minimal number of samples per leaf, etc.

Finally, as the tree-based methods are close to the estimators system of WideTech (cf. Sections 6.1 and 6.2), a hybrid method between the two can be imagined. For example, one could replace each sensor estimator by an automated tree or use the automatically built trees to reduce the time required to write the estimator rules. Many things are possible, which could make use of the advantage of the automation of the estimator construction from the tree-based model and the human input from the WideTech system.

Chapter II

Second task: Time series forecasting

The second task is concerned with simultaneously forecasting multiple steps of multiple time series from production data. In other words, it is a multivariate multistep time series forecasting task. State-of-the-art level deep learning methods are considered to solve this problem. In particular, two models are tested: DeepAR [Salinas et al., 2017] and Temporal Fusion Transformers (TFT) [Lim et al., 2019]. The evaluation is based on their forecasts of three different datasets, each being data measured from real production systems. However, both models show poor forecasting performance in the tests.

Contents

1 Second task goal

In this task, time series are predicted on multiple timesteps in the future (cf. Figure II.1). It is a useful tool to manage processes as it allows to know in advance when specific events are happening. With this knowledge, it becomes possible to prepare for those events or even to avoid them sometimes. When some incidents can force processes to stop all activities for a long period of time, from useful, this tool becomes indispensable.

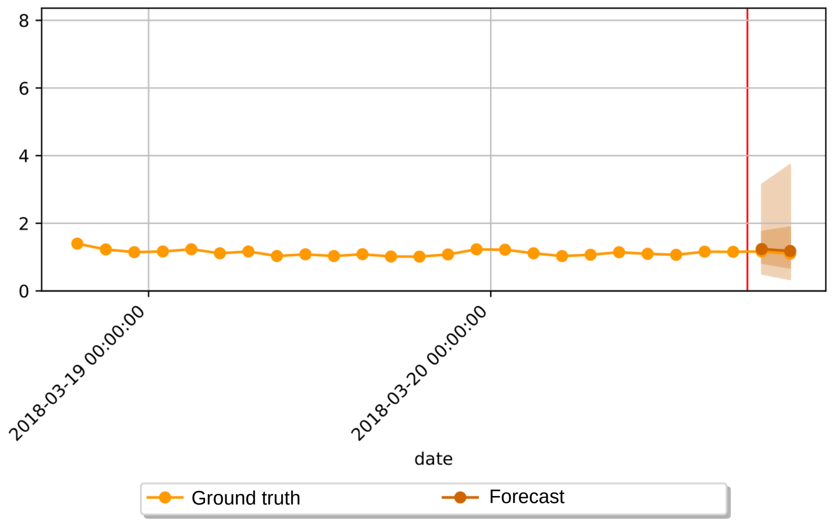


Figure II.1 – Example of multistep univariate time series forecasting where the time series has intervals of 2 hours, the context window, at the left of the vertical red line, contains 24 timesteps (represented by the points) and the prediction window, at the right of the vertical red line, contains 2 points. The time series has got only one feature, which is shown by the 'Ground truth' line. The model is given as input the 24 values of the context window (1 feature/timestep) and outputs the 2 next values of the target feature which happens to be the only feature of the dataset. On top of the point forecasts, it also gives 2 confidence intervals, a 90% confidence interval and a 50% confidence interval, illustrated by the intervals encompassing the point forecasts. The darker interval is the 50% interval.

The number of forecasting steps is not fixed as the required number for a 'useful prediction' depends directly on the time series. Instead, the problem is approached by increasing the number of steps until the accuracy lessen too much. In addition to point forecasts, confidence intervals are also predicted to give an idea of the point forecast uncertainty. Indeed, if the confidence interval is large, the point forecast should be less trusted than if it was tightly surrounding the point. In general, a confidence interval has more practical use than a simple point forecast.

Besides being a multistep forecasting, the task is also multivariate. This means that instead of predicting one value per forecasting horizon[1], it forecasts simultaneously multiple ones per horizon. Moreover, it should not be tuned for a particular kind of data but should be a general model capable of handling a wide range of production datasets. While the setup might change depending on the dataset used, usually, it consists of many years of measurements of the production process. Those measurements are done at irregular time intervals but are aggregated into points of constant time intervals. This aggregation, processed by WideTech, computes statistics for each aggregated point such as the mean, the standard deviation, the minimum, etc. Thus, time series contain signals like the temperature, the pressure or the flow monitored by sensors in the production process. Then

[1]The forecast horizon is the length of time into the future for which forecasts are to be prepared [Eurostat, 2014].

at each timestep, these signals are given in terms of mean, standard deviation, minimum, etc.; those statistics are referred to as feature in this task. Finally, out of all features, a few are chosen as the ones to forecast, the target features. The model is trained to take as input all features of C consecutive timesteps, the context window, and to forecast the target features of the P next timesteps, the prediction window. When the model allows it, some features of the prediction window are also given as input. Those correspond to features whose future values can be known, for example, a time feature such as the day of the week.

In summary, the goal of the task is to produce a model capable of accurate multivariate multistep forecasting for a wide range of production datasets.

2 Data description

As explained before, the goal is not to forecast a particular dataset but to be able to predict any production dataset after training. This section explains how missing data are handled, what preprocessing is applied and, finally, what are the data for the experiment.

Datasets are provided by WideTech. Originally, they contains measurements collected by sensors at irregular time intervals on a production process. Then, WideTech aggregates values together to built a timeseries with constant time intervals and multiple statistics per signal. For example, signals could be the temperature and the pressure in a room and, considering the aggregated statistics as being the average and the standard deviation, each timestep of the time series would have the mean and standard deviation of the temperature and the mean and the standard deviation of the pressure. The statistics are referred to as features. In the previous example, the time series has got four features.

2.1 Consideration of missing data

The missing data is considered the same way as in the previous task, i.e. mostly missing completely at random (MCAR) with very rare cases of missing at random (MAR) and missing not at random (MNAR) (cf. Chapter I Section 2.3). It is handled by imputing data by iterative imputation with the k-nearest neighbors method (cf. Chapter I Section 4).

2.2 Data preprocessing

Datasets are preprocessed in two ways. The first one detects and removes the outliers while the second one rescales the data in preparation to feeding it to neural networks.

Detection and removal of outliers

Outliers are observations with abnormal values compared to other observations of the population. While some might be real data, others can be erroneous measurements, caused, for example, by a sensor failure. In the presence of outliers from the latter case in the target signal, neural network models will see their performance weaken. Indeed, from the model point of view those points correspond to random abnormal values whose inputs could potentially not differ from other points with ‘normal’ values. For example, let a model be trained for a 1 step forecast with a dataset with time series of constant value with some random outliers of huge values. Then during training and testing, for the model to achieve a good score, it needs to be able to forecast the random outliers, which is by definition impossible. Thus, the model could either try to predict, wrongly, when those outliers could happen, or predict a value between the constant one and the outliers, or always predict the outliers or the constant value. Either way, the training and testing will be negatively affected by these outliers, which is why outliers are removed from the dataset during the preprocessing.

The interquartile range (IQR) method [Zwillinger and Kokoska, 2000] is applied to each feature of the dataset to detect the outliers:

$$IQR = Q_{75} - Q_{25}, \tag{II.1}$$

with Q_i designing the i-th quantile. An outlier is defined by values respectively above or below $Q_{75} + 1.5 \cdot IQR$ and $Q_{25} - 1.5 \cdot IQR$. Depending on the data, however, the outlier definition might be slightly different (for example when observations with known abnormal values shouldn't be removed). Once outliers are detected, each of them is removed and replaced by missing data, later handled by the missing data handling (cf. Section 2.1).

Data rescaling

Data rescaling might be applied to the datasets after the outliers removal for two different reasons, in two different ways. First, data rescaling during preprocessing and postprocessing can force the forecast to be within fixed bounds. It is done by applying a function $f : \mathcal{D} \to \mathbb{R}$ to the each feature of the datasets before training, where $\mathcal{D}$ is the domain of the data, and a function $g : \mathbb{R} \to \mathcal{D}$ during the postprocessing of the model's outputs. Doing so easily insures that the output of the model stays in the domain $\mathcal{D}$. It is called normalization.

The functions f and g are divided in two sub-functions. The first set of functions is responsible to map values from $\mathcal{D}$ to $[0, 1]$ and inversely, with a min-max rescaling. Then, the second set of function map $[0, 1]$ to $\mathbb{R}$ and inversely. This process is applied to each feature of the dataset when the latter is normalized. For each feature, the minimum and maximum values, x_{min} and x_{max}, are set (either be using a prior knowledge or be observing the values in a training set), then, the functions are defined such that $x = g(f(x))$ by:

$$f(x) = f_2(f_1(x)), \tag{II.2}$$

$$f_1(x) = \frac{x - x_{min}}{x_{max} - x_{min}}, \qquad f_2(x) = \log_e(\frac{1}{x} - 1), \tag{II.3}$$

$$g(x) = g_1(g_2(x)), \tag{II.4}$$

$$g_1(x) = x(x_{max} - x_{min}) + x_{min}, \qquad g_2(x) = \frac{1}{e^x + 1}, \tag{II.5}$$

The process to force the output values in a particular domain $\mathcal{D}$, using data rescaling is illustrated in Figure II.2.

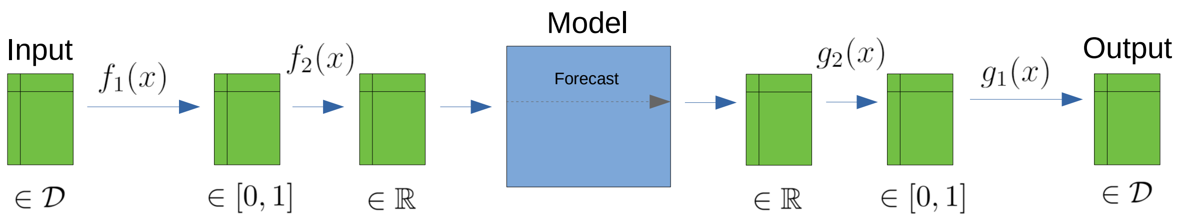


Figure II.2 – Data rescaling by normalization from the input to the output to force the output values in domain $\mathcal{D}$. The model is considered to have been trained with a dataset having received been preprocessed by $f(.)$.

The other reason why rescaling is applied is to standardize the data. When applied, this process rescales the data such that all features have a zero mean and a standard deviation of one:

$$\boldsymbol{feat}_{standardized} = \frac{\boldsymbol{feat} - \mu}{\sigma}, \tag{II.6}$$

where $\boldsymbol{feat}$ is a feature, μ is its mean and σ is its standard deviation. Again, when applied, it is used as pre-processing but also as post-processing to rescale the outputs in the initial input dimension.

The standardization is used in deep learning to improve neural networks convergence. Many weight initialization strategies for networks, such as LeCun's uniform initialization, Xavier initialization or normalized initialization, assume the variance of each input features are the same. Often not verified, this assumption can be imposed by standardization. However, in non-convex optimization problems[2], initialization is crucial [Louppe, 2020], which by extent makes standardization important too. Furthermore, the convergence of neural network models is faster when features means are close to zero [LeCun et al., 2012] which is also verified after standardization.

2.3 Detailed description of experiments data

This section describes the three datasets used for the evaluation: ds3, ds4 and ds5. Each dataset consists in anonymized data from a different real production process.

Each dataset is a timeseries of points distanced by regular time intervals. Each point contains statistics made by WideTech by aggregating multiple sensor measurements retrieved at a higher frequency than the point frequency in the time series. Thus, each signal monitored (e.g. the temperature) appears at each timestep in terms of statistics (e.g. the mean, the standard deviation, the minimum, etc.). The latter ones are called features. Moreover, time features are added to each timestep to describe the date (cf. specific description of each dataset below). Out of all features, a few are chosen as the ones to forecast: the target features. Later, forecasting models are described (cf. Section 3), they take as input all features from certain number of previous timesteps and the time features of each timestep to forecast, then, they output the target features of a certain number of timesteps to forecast.

First forecasting dataset ds3

The dataset ds3 contains time series of the same production process as ds2 from Chapter I Section 2.2. However, the length of the time series is much longer and only the average

[2]A non-convex optimization problem is a problem with multiple locally optimal points.

values of each signal are available (cf. Table II.1). The average values are computed by WideTech by aggregating measurements with smaller and inconsistent time intervals into larger constant time interval. For all aggregation dates d distanced by a constant interval t, the values are computed by WideTech with:

$$average_d = \frac{\sum_{d \leq d_i < d+t} x_{d_i}}{\sum_{d \leq d_i < d+t} 1}, \tag{II.7}$$

where all the notation d_i refer to measurements dates and x_e is the measurement at date e. The equation (II.7) is repeated for all signals of the time series. The rest of them which were used in the previous task, minimum, maximum, gradient and standard deviation, couldn't be exported in this case. This is a restriction coming from WideTech which is responsible from the dataset extraction.

Name	Time interval	Start date	End date	Points	Features
ds3	1 h	2015/12/01 00:00	2020/04/17 00:00	38350	55

Table II.1 – Dataset ds3 characteristics

The context for ds3 is the same as ds2 (cf. Figure II.3). It can be simplified into the following. Multiple oil wells are placed in series. Then, a multiphase pump (MPP) increases the flow inside the pipe which contains a mix of gas, oil and water extracted from the wells. Finally, a multiphase flow meter (MPFM) measures flows individually in terms of gas, oil and water. The main signals are measurements such as temperature, pressure, flow, etc. from sensors in regions A and B, the MPP motor speed and the individual gas, oil and water flows in A.

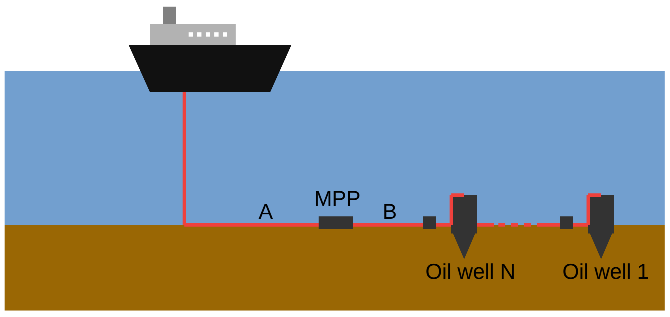


Figure II.3 – Simplified context of ds3: offshore oil well where N wells are put in series and a multiphase pump (MPP) is applied to increase the flow. The region A is where the multiphase flow meter (MPFM) is applied which gives the gas, oil and water flow.

The target signals to forecast are the gas, oil and water flow (Qgas, Qoil and Qwat). The goal is to determine how those are going to evolve in the next future in order to determine the future production. Moreover, when the MPP fails, those flows are greatly reduced. Thus it also allows to anticipate the failure of the MPP and prepare for it.

To preprocess the data, outliers are removed (cf. Section 2.2), missing data is replaced (cf. Section 2.1). Then, the date of each timestep is embedded by adding the following features:

- the year as a number,
- the month as a number,
- the day as a number,

- the day of the week as a number,
- the hour as a number,
- the difference in day between the date and a fixed date in the past chosen such that the difference will stay positive.

The Figures II.4, II.5 and II.6 respectively display the target features Qgas, Qoil and Qwat throughout ds3 and indicate when values were imputed.

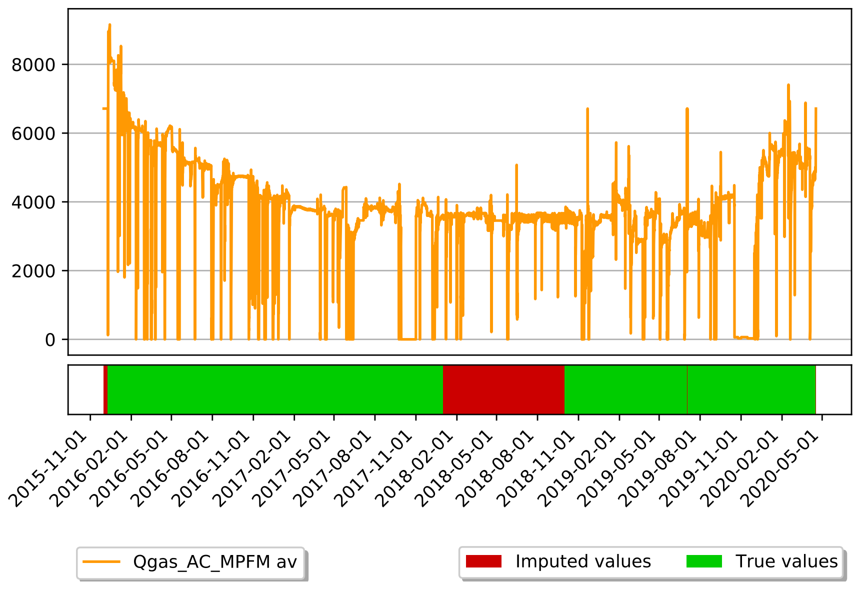


Figure II.4 – Gas flow feature Qgas throughout ds3

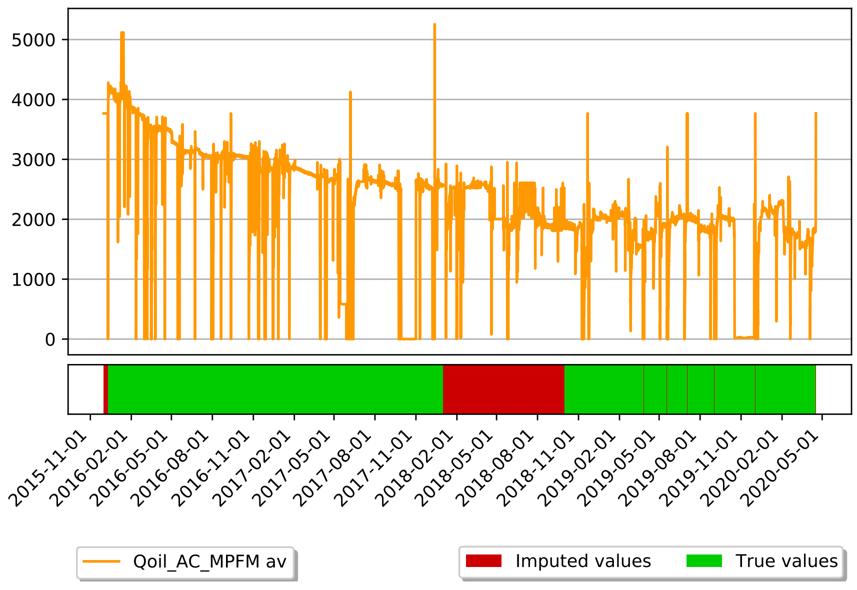


Figure II.5 – Oil flow feature Qoil throughout ds3

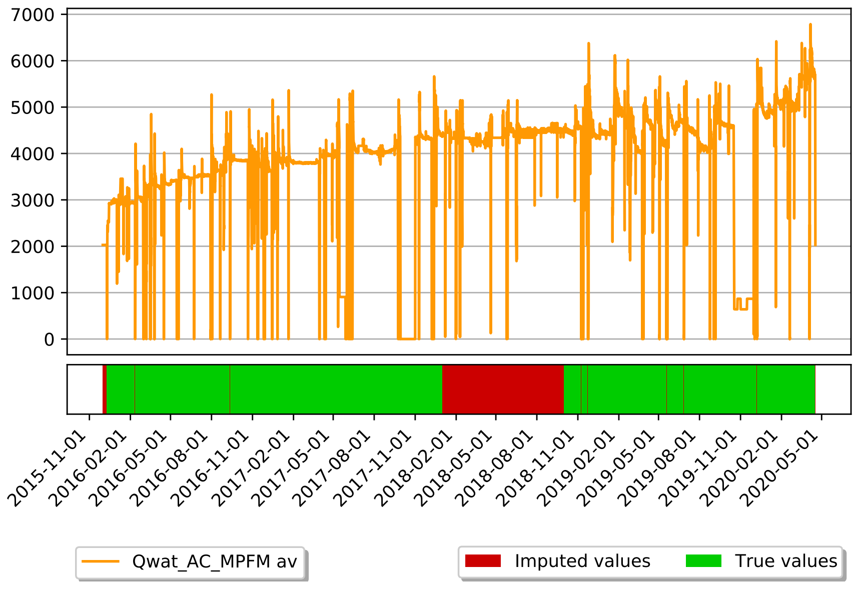


Figure II.6 – Water flow feature Qwat throughout ds3

Second forecasting dataset ds4

The dataset ds4 has close to the same number of features as ds3 but it is a bit longer (cf. Table II.2). In fact ds4 is a simplification of ds3 as it is close to being the same process but differs from it by focusing on one well (cf. Figure II.7). As only one well is taken into account, the forecasting task should be easier in ds4 than in ds3. As for ds3, a mixture of gas, oil and water are extracted by the well. It goes through a choke, a valve which controls the flow of the well, then the gas, oil and water flows are measured with a MPFM.

Name	Time interval	Start date	End date	Points	Features
ds3	1 h	2015/12/01 00:00	2020/04/17 00:00	38350	55
ds4	1 h	2014/06/28 12:00	2020/04/22 10:00	50999	53

Table II.2 – Dataset ds4 characteristics

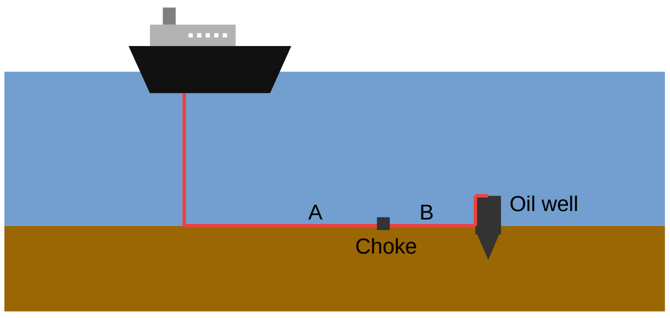


Figure II.7 – Simplified context of ds4: offshore oil well data focused on one well. The choke controls the flow of the well while a multiphase flow meter (MPFM) is applied in the area A to give the gas, oil and water flow.

As for ds3, ds4 only contains the average value for each signal computed by aggregating multiple timesteps into a constant bigger timestep. For all aggregation dates d distanced by a constant interval t, the values are computed by WideTech with:

$$average_d = \frac{\sum_{d \leq d_i < d+t} x_{d_i}}{\sum_{d \leq d_i < d+t} 1}, \tag{II.8}$$

where all the notation d_i refer to measurements dates and x_e is the measurement at date e.

The target features are still the gas, oil and water flows (Qgas, Qoil and Qwat). Those are forecasted in order to predict when the well will be forced to shutdown. This is reflected in the flows by their values going down to zero (cf. Figures II.8, II.9, II.10). The other features present in ds4 are temperature, pressure, flow, etc., measured in regions A and B (cf. Figure II.7), but also the choke percentage which controls the flow.

As for ds3, the dataset is preprocessed by removing its outliers and imputing its missing data (cf. Sections 2.2 and 2.1). Finally, the date is embedded as features as previously.

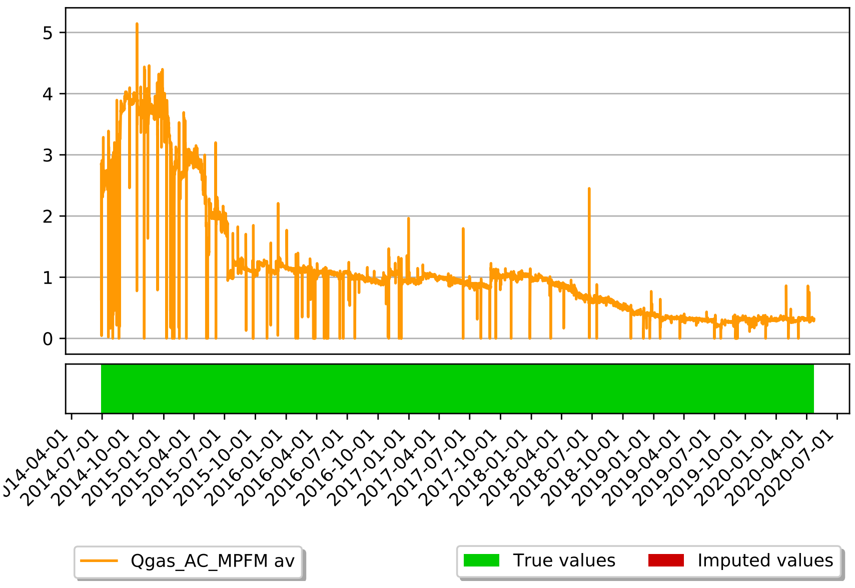


Figure II.8 – Gas flow feature Qgas throughout ds4

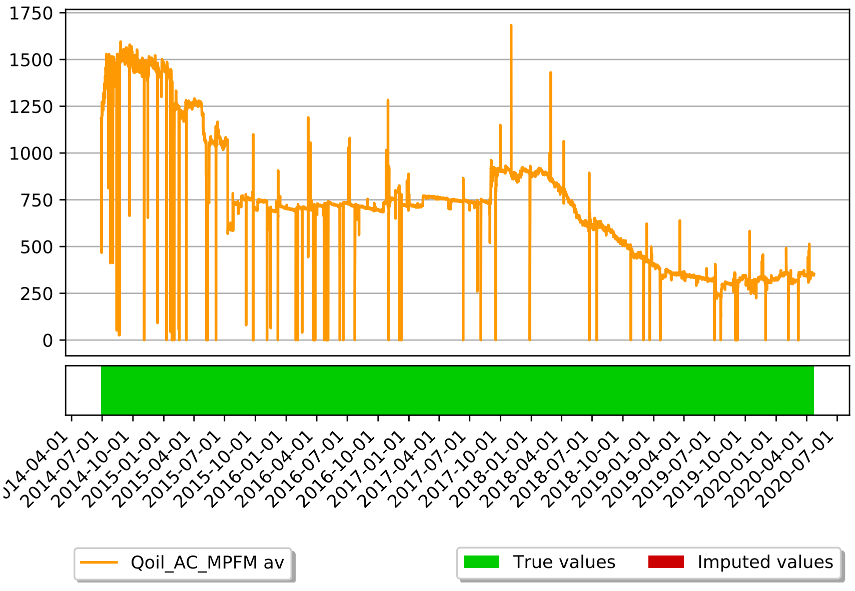


Figure II.9 – Oil flow feature Qoil throughout ds4

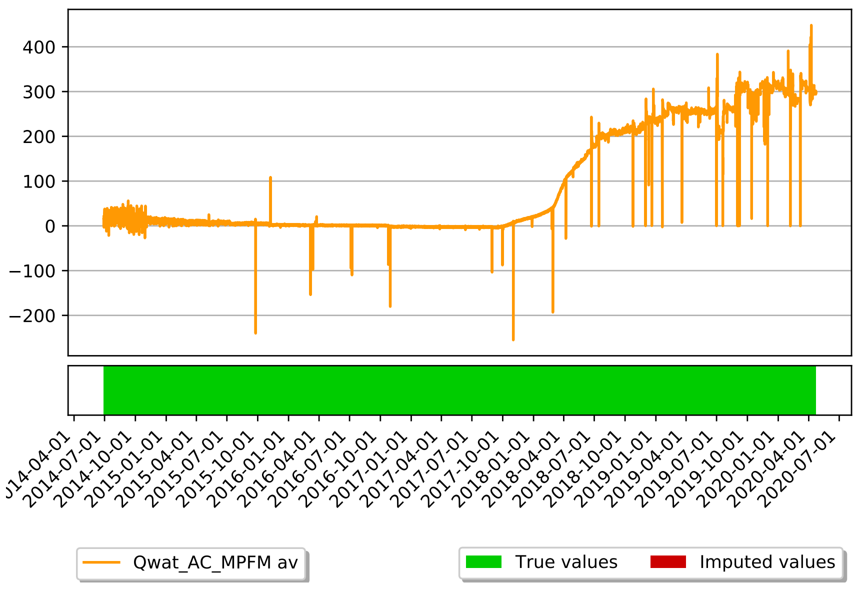


Figure II.10 – Water flow feature Qwat throughout ds4

Final forecasting dataset ds5

The last dataset ds5 differs from the two previous ones. The time difference between each timestep is two hours and it contains almost 5 times more features than the others (cf. Table II.3). This increase in feature is partly caused by the fact that each signal is here available in terms of gradient, standard deviation and average obtained by aggregating multiples timesteps into a larger constant timestep for each signal. Those are computed with: For all aggregation dates d distanced by a constant interval t, the values are computed by WideTech for each signal with:

$$average_d = \frac{\sum_{d \le d_i < d+t} x_{d_i}}{\sum_{d \le d_i < d+t} 1}, \tag{II.9}$$

$$standard\_deviation_d = \sqrt{\frac{1}{\sum_{d \le d_i < d+t} 1} \sum_{d \le d_i < d+t} (x_{d_i} - average_d)^2}, \tag{II.10}$$

$$gradient_d = x_{d_2} - x_{d1}, \quad with \begin{cases} d_1 = \min_{d \le d_i < d+t}(d_i) \\ d_2 = \max_{d \le d_i < d+t}(d_i) \end{cases}, \tag{II.11}$$

where all the notation d_i refer to measurements dates and x_e is the measurement at date e for a given signal.

Name	Time interval	Start date	End date	Points	Features
ds3	1 h	2015/12/01 00:00	2020/04/17 00:00	38350	55
ds4	1 h	2014/06/28 12:00	2020/04/22 10:00	50999	53
ds5	2 h	2011/12/31 23:00	2020/05/13 20:00	36308	252

Table II.3 – Dataset ds5 characteristics

This time, the production process is the granulation of solid fertilizer. During the granulation, cold or warm air is required at different steps of the process. With a fan system, the fresh air is routed toward the key spots before being retrieved after its usage by another fan system. While carrying the air around, some fertilizer particles might get stuck to fans which will cause it to vibrate after a certain amount has been collected. These vibrations can cause damage to the system and are to be avoided. If detected early enough, it is possible to clean the fans without stopping the system, however, if not, everything needs to be stopped as it cannot work without the air circuit.

The target feature is the average vibration of one of the four main fans of the process (vib01). This one is responsible for the extraction of all used air. As explained earlier, the goal is to predict when the vibration will reach a high level which would cause the system to shutdown. The average vibration feature of the fan is displayed in Figure II.11. The other features are water density, flows, pressures, etc., measured throughout the process.

This dataset is preprocessed by removing its outliers and imputing its missing data (cf. Sections 2.2 and 2.1). Moreover, the date is embedded the same way as for ds3 and ds4.

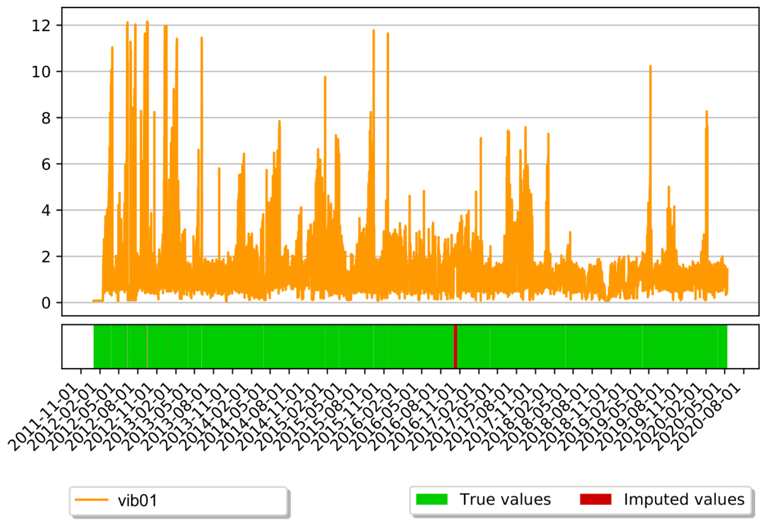


Figure II.11 – Average vibration of the fan feature (vib01) throughout ds5

3 Models considered

Two models are considered: Temporal Fusion Transformers (TFT) and DeepAR. The choice fell on those two as they have demonstrated high performance in multistep time series forecasting and are established candidates for the deep learning state-of-the-art in this task [Lim et al., 2019].

3.1 Temporal Fusion Transformers

This section describes the Temporal Fusion Transformers (TFT) model by summarizing [Lim et al., 2019].

Introduction

TFT is a multi-horizon forecaster, which contrary to auto-regressive models are capable of optimizing their predictions at multiple steps in the future. It is based on attention mechanisms commonly used in language processing tasks [Galassi et al., 2019] to retrieve the relevance of the given features. In TFT, the attention mechanisms are the foundation of a decoder capable of learning long-term dependencies. Moreover, TFT also makes use of gating mechanisms to minimize the contributions of irrelevant inputs.

Multi-horizon forecasting

Multi-horizon forecasting consists in predicting multiple future timesteps of the target time series based on static covariates, past targets, past observed inputs and past and future known inputs (cf. Figure II.12).

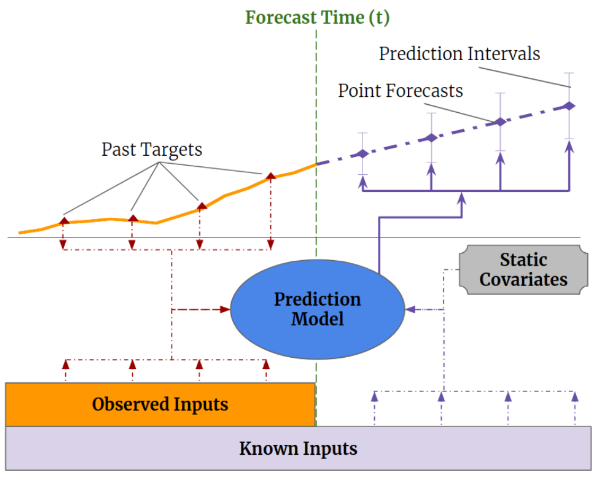


Figure II.12 – Illustration of Multi-horizon Forecasting with static covariates and various time-dependent inputs [Lim et al., 2019]

A time series dataset of I elements is considered. Each element i consists of three parts: a set of static covariates $s_i \in \mathbb{R}^{m_s}$, a set of inputs $\chi_{i,t} \in \mathbb{R}^{m_\chi}$ and a set of scalar targets $y_{i,t} \in \mathbb{R}$ at each timestep $t \in [0, T_i]$. Two categories form the time-dependent input features $\chi_{i,t} = [z_{i,t}^T, x_{i,t}^T]^T$. The observed inputs $z_{i,t} \in \mathbb{R}^{m_z}$ can be only measured at past timesteps and are unknown at forecast timesteps while $x_{i,t} \in \mathbb{R}^{m_x}$ are inputs known at all time (e.g. the day of the week).

Quantile regression is adopted to allow interval predictions instead of only point predictions. It is done by outputting percentiles for each forecast timestep. A quantile forecast is given by:

$$\hat{y}_i(q, t, \tau) = f(q, \tau, y_{i,t-k:t}, z_{i,t-k:t}, x_{i,t-k,t+\tau}, s_i), \tag{II.12}$$

where $\hat{y}_{i,t+\tau}(q, t, \tau)$ is the predicted q-th sample quantile of the τ-step-ahead forecast at time t, and $f_q(.)$ is a prediction model. As a multi-horizon model, all forecasts for $\tau \in [1, \tau_{max}]$ are predicted at once. For simplicity, the subscript i is omitted in the rest of the section.

Model architecture

The TFT model is based on 5 main components:

- **Gating mechanism :** It provides a way to skip over parts of the network, thus increasing the adaptability of the network. Gated residual networks are the main building blocks of TFT.
- **Variable selection networks :** They select the relevant features of each timestep.

- **Static Covariate Encoders :** They integrate static covariates into the network.
- **Temporal processing :** The temporal processing is divided in the short-term and long-term relationships. The short-term dependencies are processed with LSTM (Long Short-Term memory) while long-term ones are handled by a novel multi-head attention mechanism.
- **Multi-Horizon Forecast Intervals Prediction :** Quantiles are predicted at each horizon.

Each of those are detailed below whereas the global architecture is illustrated on Figure II.13.

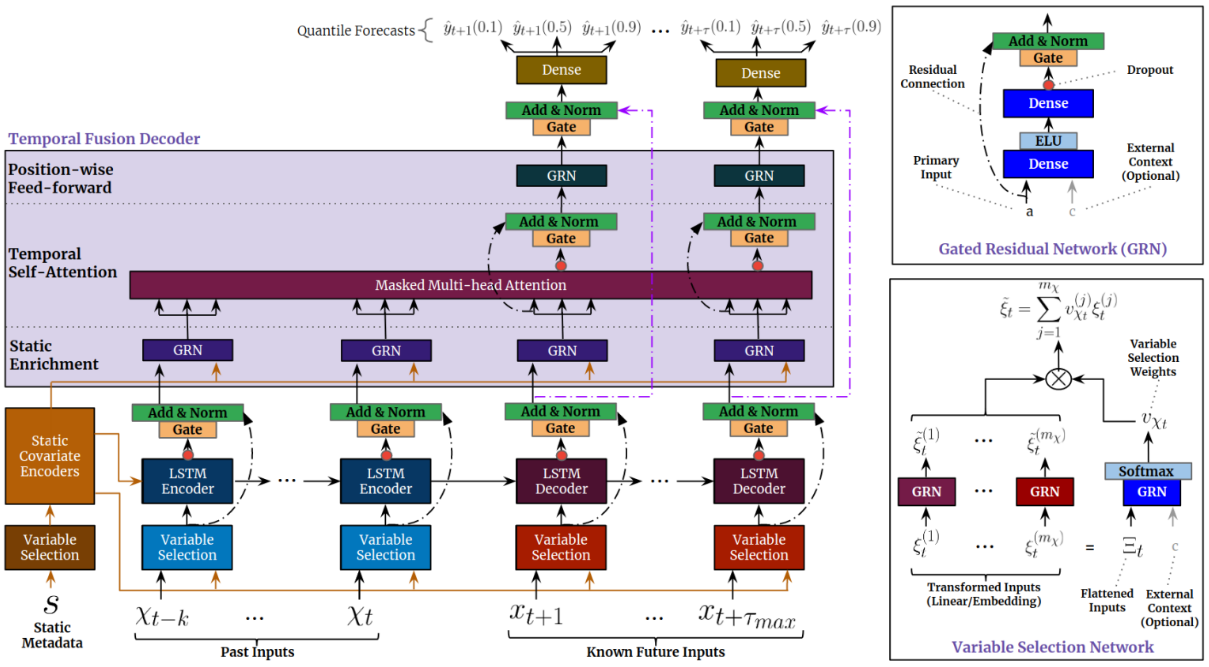


Figure II.13 – Temporal Fusion Transformer (TFT) architecture [Lim et al., 2019]

Gating mechanism: When building a neural network, one major difficulty is to fix the architecture. In some tasks, smaller and simpler networks can perform better than overly large and complex ones, however, the opposite is also true in other tasks. Thus this choice must be made according the level of non-linear processing required to go from inputs to outputs, which is task- and dataset-dependent.

To increase the adaptability to a wide range of tasks and datasets, GRNs (Gated Residual Networks) are implemented in TFT. Those apply non-linear processings only when required. Let $\boldsymbol{a}$ be a primary input and $\boldsymbol{c}$ an optional context, then GRN is defined by:

$$\mathrm{GRN}_\omega(\boldsymbol{a},\boldsymbol{c}) = \mathrm{LayerNorm}(\boldsymbol{a} + \mathrm{GLU}_\omega(\boldsymbol{\eta}_1)), \tag{II.13}$$

$$\boldsymbol{\eta}_1 = \boldsymbol{W}_{1,\omega}\boldsymbol{\eta}_2 + \boldsymbol{b}_{1,\omega}, \tag{II.14}$$

$$\boldsymbol{\eta}_2 = \mathrm{ELU}(\boldsymbol{W}_{2,\omega}\boldsymbol{a} + \boldsymbol{W}_{3,\omega}\boldsymbol{c} + \boldsymbol{b}_{2,\omega}), \tag{II.15}$$

where $\boldsymbol{\eta}_1$, $\boldsymbol{\eta}_2$ are intermediate layers, ELU is the exponential linear unit activation function, LayerNorm is the standard layer normalization of [Lei Ba et al., 2016] and ω is an index denoting how weights are shared. To skip unnecessary part of the networks, a Gated Linear Unit (GLU) [Dauphin et al., 2016] is adopted in equation (II.13). If $\sigma(.)$ is the activation function, $\boldsymbol{W}_{(.)}$ and $\boldsymbol{b}_{(.)}$ are the weights and biases, $\odot$ is the element-wise product between matrices, then GLU can be written:

$$\mathrm{GLU}_\omega(\boldsymbol{\gamma}) = \sigma(\boldsymbol{W}_{3,\omega}\boldsymbol{\gamma} + \boldsymbol{b}_{3,\omega}) \odot (\boldsymbol{W}_{4,\omega}\boldsymbol{\gamma} + \boldsymbol{b}_{4,\omega}). \tag{II.16}$$

The GLU tunes the contribution of GRN to the original input $\boldsymbol{a}$ (cf. equation II.13), which can be lowered down to zero if needed.

Variable selection networks: The precise relationship between the inputs and the outputs is often unknown. Moreover, noisy inputs (i.e. irrelevant or redundant) can negatively affect the model performance. In these common situations, variable selection mechanisms can show their usefulness by processing the input relevance.

Variable selection networks are applied in TFT to both static and time-dependent variables. However, separated networks are used for static, past and future inputs (shown in Figure II.13 by different colors). First the variables are transformed into a (d_{model})-dimensional vector, matching the required dimensions of subsequent layers: by linear transformation for continuous variables and entity embedding (i.e. turning positive integers into dense vectors of fixed size) for categorical ones. Then the resulting vector is used in the variable selection network. As explained above, there exists three networks respectively for the static, past and future inputs but only the historical one is explained below as the other two follow the same principle.

Let the transformed input by linear transformation/entity embedding of the j-th variable at time t be denoted by $\xi_t^{(j)} \in \mathbb{R}^{d_{model}}$, the flatten vector of all historical inputs at time t is $\Xi_t = \left[\xi_t^{(1)^T}, ..., \xi_t^{(m_\chi)^T}\right]^T$. Then, by feeding both Ξ_t and an external context vector $\boldsymbol{c}_s$ through a GRN followed by a Softmax layer, the variable selection weights are obtained:

$$\boldsymbol{v}_{\chi_t} = \mathrm{Softmax}\Big(\mathrm{GRN}_{v_\chi}(\Xi_t, \boldsymbol{c}_s)\Big), \tag{II.17}$$

where $\boldsymbol{v}_{\chi_t}$ is the vector of variable selection weights, and $\boldsymbol{c}_s$ is generated from a static covariate encoder (cf. Paragraph Static covariate encoders), except for the selection of static variable where $\boldsymbol{c}_s$ is ignored.

In parallel to the selection, each timestep of each feature $\xi_t^{(j)}$ is processed non-linearly by feeding it to its variable GRN, the GRN weights being shared across all timesteps t:

$$\tilde{\xi}_t^{(j)} = \mathrm{GRN}_{\tilde{\xi}^{(j)}}\left(\xi_t^{(j)}\right). \tag{II.18}$$

Finally, the processed variables $\tilde{\xi}_t^{(j)}$ are weighted by the selection weight vector $\boldsymbol{v}_{\chi_t}$:

$$\tilde{\xi}_t = \sum_{j=1}^{m_\chi} v_{\chi_t}^{(j)} \tilde{\xi}_t^{(j)}, \tag{II.19}$$

where $v_{\chi_t}^{(j)}$ is the j-th element of $\boldsymbol{v}_{\chi_t}$.

Static covariate encoders: Throughout the model, four different context vectors are used: $\boldsymbol{c}_s$ for the temporal variable selection, $\boldsymbol{c}_c, \boldsymbol{c}_h$ for the local processing of temporal variable and $\boldsymbol{c}_e$ for enriching temporal features with static information. Each vector is obtained by applying a different GRN encoder to the output of the static variable selection ξ:

$$\boldsymbol{c}_i = \mathrm{GRN}_{\boldsymbol{c}_i}(\xi), \quad \forall i \in \{s, c, h, e\}. \tag{II.20}$$

Long Short-Term Memory: The Long Short-Term Memory (LSTM) is a gated recurrent network used to process temporal inputs x_t, $\forall t \in [0, T]$ introduced by [Hochreiter and Schmidhuber, 1997]. It is applied in TFT for locally processing temporal variables (cf. Paragraph Locality enhancement with sequence-to-sequence layer).

For each input x_t, LSTM outputs its local processing y_t. As a recurrent neural network, the architecture is based on a stack of cells, linked by their recurrent state. Each cell takes as input a temporal input x_t, a recurrent cell state c_{t-1} and a recurrent output state h_{t-1} then it generates a recurrent cell state c_t and a recurrent output state h_t, which is also the local temporal processing output $y_t = h_t$ (cf. Figure II.14).

Let σ be an activation function, $\odot$ be the element-wise product between matrices, $\boldsymbol{W}_{(.)}$ and $\boldsymbol{b}_{(.)}$ be the weights and biases shared across all LSTM cells. Then inside the t-th LSTM cell, the following operations are done. First a forget gate $\boldsymbol{f}_t$ is computed, its role is to select the information to remove from the cell state c_{t-1}:

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_f[\boldsymbol{h}_{t-1}, \boldsymbol{x_t}] + \boldsymbol{b}_f). \tag{II.21}$$

Then an input gate $\boldsymbol{i}_t$ selects the information to update in the cell state:

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i[\boldsymbol{h}_{t-1}, \boldsymbol{x_t}] + \boldsymbol{b}_i), \tag{II.22}$$

$$\tilde{\boldsymbol{c}}_t = tanh(\boldsymbol{W}_c[\boldsymbol{h}_{t-1}, \boldsymbol{x_t}] + \boldsymbol{b}_c). \tag{II.23}$$

The cell state is computed:

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_t + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t. \tag{II.24}$$

Finally, an output gate $\boldsymbol{o}_t$ selects the information to output $\boldsymbol{h}_t$:

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o[\boldsymbol{h}_{t-1}, \boldsymbol{x_t}] + \boldsymbol{b}_o), \tag{II.25}$$

$$\boldsymbol{h}_t = \boldsymbol{y}_t = \boldsymbol{o}_t \odot tanh(\boldsymbol{c}_t) \tag{II.26}$$

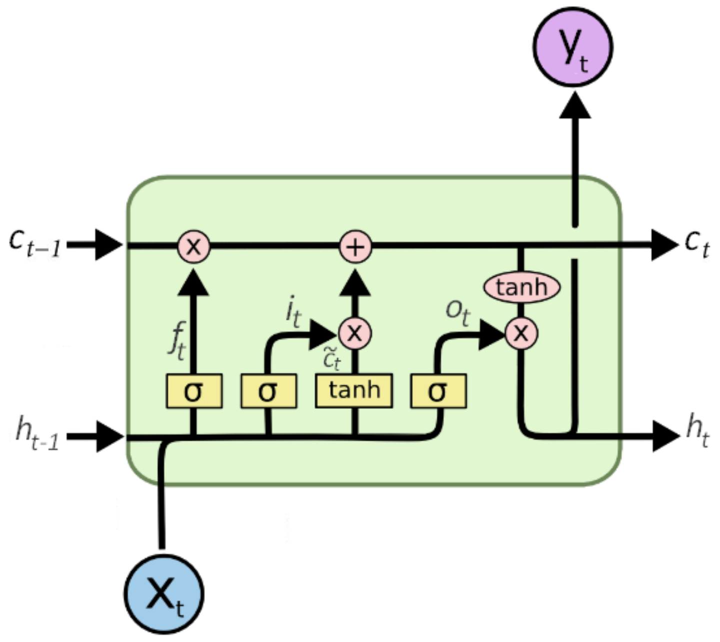


Figure II.14 – LSTM cell [Tixier, 2018]

Interpretable Multi-Head Attention: A self-attention mechanism is deployed in TFT to learn long-term relationships across different timesteps (cf. Paragraph Temporal fusion decoder). Let $\boldsymbol{Q} \in \mathbb{R}^{N \times d_{attn}}$ be the queries and $(\boldsymbol{K}, \boldsymbol{V})$ be the key-values pair where $\boldsymbol{K} \in \mathbb{R}^{N \times d_{attn}}$ and $\boldsymbol{V} \in \mathbb{R}^{N \times d_V}$. Broadly speaking, the attention mechanism is a scaling of the values based on queries and keys:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = A(\boldsymbol{Q}, \boldsymbol{K})\boldsymbol{V}. \tag{II.27}$$

$A()$ is a normalization function, commonly chosen as the dot-product attention [Vaswani et al., 2017]:

$$A(\boldsymbol{Q}, \boldsymbol{K}) = \text{Softmax}(\boldsymbol{Q}\boldsymbol{K}^T / \sqrt{d_{attn}}). \tag{II.28}$$

Multi-head attention, in Transformer [Vaswani et al., 2017], repeats this mechanism with different heads to obtain different representation subspaces:

$$\text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = [\boldsymbol{H}_1, ..., \boldsymbol{H}_{m_H}]\boldsymbol{W}_H, \tag{II.29}$$

$$\boldsymbol{H}_h = \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_Q^{(h)}, \boldsymbol{K}\boldsymbol{W}_K^{(h)}, \boldsymbol{V}\boldsymbol{W}_V^{(h)}), \tag{II.30}$$

where $\boldsymbol{W}_Q^{(h)}, \boldsymbol{W}_K^{(h)} \in \mathbb{R}^{d_{model} \times d_{attn}}$ and $\boldsymbol{W}_V^{(h)} \in \mathbb{R}^{d_{model} \times d_V}$ are the query, key and value weights for the h-th head, and $\boldsymbol{W}_H \in \mathbb{R}^{(m_H \cdot d_V) \times d_{model}}$ weights the outputs of each head.

As is, the multi-head attention is not very interpretable as analyzing attention weights alone would not be indicative of a particular feature's overall importance since different values are used in each head. To solve this issue, the mechanism is modified to share values in each head and to employ additive aggregation of all heads at the output:

$$\text{InterpretableMultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \tilde{\boldsymbol{H}}\tilde{\boldsymbol{W}}_H, \tag{II.31}$$

$$\begin{aligned} \tilde{\boldsymbol{H}} &= \tilde{A}(\boldsymbol{Q}, \boldsymbol{K})\boldsymbol{V}\boldsymbol{W}_V, & \text{(II.32)} \\ &= \Big\{\frac{1}{H}\sum_{h=1}^{m_H} A\big(\boldsymbol{Q}\boldsymbol{W}_Q^{(h)}, \boldsymbol{K}\boldsymbol{W}_K^{(h)}\big)\Big\}\boldsymbol{V}\boldsymbol{W}_V, & \text{(II.33)} \\ &= \frac{1}{H}\sum_{h=1}^{m_H} \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_Q^{(h)}, \boldsymbol{K}\boldsymbol{W}_K^{(h)}, \boldsymbol{V}\boldsymbol{W}_V), & \text{(II.34)} \end{aligned}$$

where $\boldsymbol{W}_V \in \mathbb{R}^{d_{model} \times d_V}$ is shared across all heads, and linear mapping is done with $\boldsymbol{W}_H \in \mathbb{R}^{d_{attn} \times d_{model}}$. With this modification, the multi-head attention can learn different temporal patterns for each head while increasing its interpretability.

Locality enhancement with sequence-to-sequence layer: Attention-based architectures can be enhanced by constructing features using pattern information on top of point-wise values [Li et al., 2019]. Indeed, a timestep significance in time series data is often based on neighbouring values, leading to the detection of anomalies, change-points, cyclical pattern, etc.. In [Li et al., 2019], local patterns are extracted with a single convolutional network, using the same filter across all timesteps. However, to be able to apply the locality enhancement when the number of past and future input changes, TFT proposes a sequence-to-sequence model.

The features about past time $\tilde{\xi}_{t-k:t}$ are feed into the encoder while the features about future time $\tilde{\xi}_{t+1:\tau_{max}}$ are feed into the decoder. This sequence-to-sequence layer then outputs the locality enhanced features $\boldsymbol{\phi}(t, n)$ $\forall n \in \{-k, ..., \tau_{max}\}$. It is implemented with LSTM in TFT, which is initialized in the first cell with the static context vectors $\boldsymbol{c}_c$

and $\boldsymbol{c}_h$, previously generated by the static covariate encoders, respectively as the recurrent cell state and recurrent cell output.

To keep the model adaptability, a gated skip connection is applied on top of the sequence-to-sequence layer:

$$\tilde{\boldsymbol{\phi}}(t,n) = \text{LayerNorm}\Big(\tilde{\xi}_{t+n} + \text{GLU}_{\tilde{\phi}}(\boldsymbol{\phi(t,n)})\Big), \quad \forall n \in \{-k, ..., \tau_{max}\}. \tag{II.35}$$

Temporal fusion decoder: The goal of the temporal fusion decoder is to decode the locality enhanced time features and capture the long-range dependencies between them. First, static covariates are used to enrich the features. It is done by applying a GRN, shared across all time, with the context vector $\boldsymbol{c}_e$ produced by the static encoders:

$$\boldsymbol{\theta}(t,n) = \text{GRN}_\theta\Big(\tilde{\boldsymbol{\phi}}(t,n), \boldsymbol{c}_e\Big), \quad \forall n \in \{-k, ..., \tau_{max}\}. \tag{II.36}$$

Those are grouped in the matrix $\boldsymbol{\Theta}(t) = [\boldsymbol{\theta}(t,-k), ...\boldsymbol{\theta}(t,\tau_{max})]^T$. Then, $\boldsymbol{\Theta}(t)$ is fed into an interpretable multi-head attention for each prediction time:

$$\boldsymbol{B}(t) = [\boldsymbol{\beta}(t,-k), ..., \boldsymbol{\beta}(t,\tau_{max}] = \text{InterpretableMultiHead}(\boldsymbol{\Theta}(t), \boldsymbol{\Theta}(t), \boldsymbol{\Theta}(t)). \tag{II.37}$$

The dimension d_V and d_{attn} are fixed based on the number of head m_H: $d_V = d_{attn} = d_{model}/m_H$. To prevent temporal dimension to attend to features preceding it, decoder masking [Li et al., 2019; Vaswani et al., 2017] is applied to the multi-head attention layer. The multi-head network is followed by a gating layer:

$$\boldsymbol{\delta}(t,n) = \text{LayerNorm}(\boldsymbol{\theta}(t,n) + \text{GLU}_\delta(\boldsymbol{\beta}(t,n))), \tag{II.38}$$

itself followed by a non-linear processing, shared across all timesteps:

$$\boldsymbol{\psi}(t,n) = \text{GRN}_\psi(\boldsymbol{\delta}(t,n)). \tag{II.39}$$

At last, the decoder is finalised by applying a gated skip connection over it:

$$\tilde{\boldsymbol{\psi}}(t,n) = \text{LayerNorm}\Big(\tilde{\boldsymbol{\phi}}(t,n) + \text{GLU}_{\tilde{\psi}}(\boldsymbol{\psi}(t,n))\Big). \tag{II.40}$$

Quantile outputs: Prediction intervals are predicted by TFT by simultaneously forecasting percentiles delimiting the intervals at each timestep. Those are generated by linear transformation of the temporal fusion decoder output:

$$\hat{y}(q,t,\tau) = \boldsymbol{W}_q \tilde{\boldsymbol{\psi}}(t,\tau) + b_q, \quad \forall \tau \in \{1, ..., \tau_{max}\}. \tag{II.41}$$

Training procedure

To optimize the model, in line with previous work [Wen et al., 2017], TFT is trained by minimizing the quantile loss summed accross all quantile outputs:

$$\mathcal{L}(\Omega, \boldsymbol{W}) = \sum_{y_t \in \Omega} \sum_{q \in \mathcal{Q}} \sum_{\tau=1}^{\tau_{max}} = \frac{QL(y_t, \hat{y}(q, t-\tau, \tau), q)}{M\tau_{max}} \tag{II.42}$$

$$QL(y, \hat{y}, q) = q \max(y - \hat{y}, 0) + (1-q) \max(\hat{y} - y, 0) \tag{II.43}$$

where the domain of the training data is Ω, the number of samples is M, the weight of TFT is $\boldsymbol{W}$ and the set of quantiles is $\mathcal{Q}$.

3.2 DeepAR

DeepAR is a generative, auto-regressive model. It is similar to the architectures described in Flunkert et al [Salinas et al., 2017]. DeepAR consists of a recurrent neural network (RNN) using Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) cells that takes the previous time points and covariates as input (cf. Paragraph LSTM and GRU cells).

This section is concerned with the forecasting model from [Salinas et al., 2017] and summarizes the paper.

Introduction

Contrary to most forecasting methods, DeepAR learns from of every time series jointly. It is based on an auto-regressive recurrent neural network model. At the paper release in 2017, it was outperforming the state-of-the-art forecasting methods on several problems.

Model

Let $z_{i,t}$ be the value of time series i at time t. Given past values $[z_{i,1}, \ldots, z_{i,t_0-2}, z_{i,t_0-1}] \triangleq \boldsymbol{z}_{i,1:t_0-1}$ the goal is to model the conditional distribution of the future values $[z_{i,t_0}, z_{i,t_0+1}, \ldots, z_{i,T}] \triangleq \boldsymbol{z}_{i,t_0:T}$,

$$P(\boldsymbol{z}_{i,t_0:T}|\boldsymbol{z}_{i,1:t_0-1}, \boldsymbol{x}_{i,1:T}),$$

where the covariates $\boldsymbol{x}_{i,1:T}$ are assumed to be known. The time ranges $[1 : t_0 - 1]$ and $[t_0 : T]$ are respectively the context range and the prediction range.

The model, summarized in Figure II.15, is based on an auto-regressive recurrent network. The model distribution $Q_\Theta(\boldsymbol{z}_{i,t_0:T}|\boldsymbol{z}_{i,1:t_0-1}, \boldsymbol{x}_{i,1:T})$ is assumed to be a product of likelihood factors:

$$Q_\Theta(\boldsymbol{z}_{i,t_0:T}|\boldsymbol{z}_{i,1:t_0-1}, \boldsymbol{x}_{i,1:T}) = \prod_{t=t_0}^{T} Q_\Theta(z_{i,t}|\boldsymbol{z}_{i,1:t-1}, \boldsymbol{x}_{i,1:T}) = \prod_{t=t_0}^{T} l(z_{i,t}|\theta(\boldsymbol{h}_{i,t}, \Theta))$$

with $\boldsymbol{h}_{i,t}$ the output of an autoregressive recurrent network $\boldsymbol{h}_{i,t} = h(\boldsymbol{h}_{i,t-1}, z_{i,t-1}, x_{i,t}, \Theta)$ - which will be fed as an input of the next timestep for $\boldsymbol{h}_{i,t+1}$ -, $h(\cdot)$ being a LSTM or GRU cell (cf. Paragraph LSTM and GRU cells) and the likelihood $l(z_{i,t}|\theta(\boldsymbol{h}_{i,t}, \Theta))$ a fixed distribution parametrized by a function $\theta(\boldsymbol{h}_{i,t}, \Theta)$. The initial state $\boldsymbol{h}_{i,t_0-1}$ contains the information of the context range $\boldsymbol{z}_{i,1:t_0-1}$ necessary for predicting values in the prediction range.

Given the model parameters Θ, one can directly obtain joint samples $\hat{\boldsymbol{z}}_{i,t_0:T} \sim Q_\Theta(\boldsymbol{z}_{i,t_0:T}|\boldsymbol{z}_{i,1:t_0-1}, \boldsymbol{x}_{i,1:T})$ through ancestral sampling: First, $\boldsymbol{h}_{i;t_0-1}$ is obtained as outputs of the recurrent network (cf. Figure II.16). Then for $t = t_0, ..., T-1, T$ we sample $\hat{z}_{i,t} \sim l(\cdot|\theta(\hat{\boldsymbol{h}}_{i,t}, \Theta))$ where $\hat{\boldsymbol{h}}_{i,t} = h(\boldsymbol{h}_{i,t-1}, \hat{z}_{i,t-1}, x_{i,t}, \Theta)$ is initialized with $\hat{\boldsymbol{h}}_{i,t_0-1} = \boldsymbol{h}_{i,t_0-1}$ and $\hat{z}_{i,t_0-1} = z_{i,t_0-1}$. Using those samples allows to compute quantities such as quantiles of the distribution of values at a specific time in the prediction range.

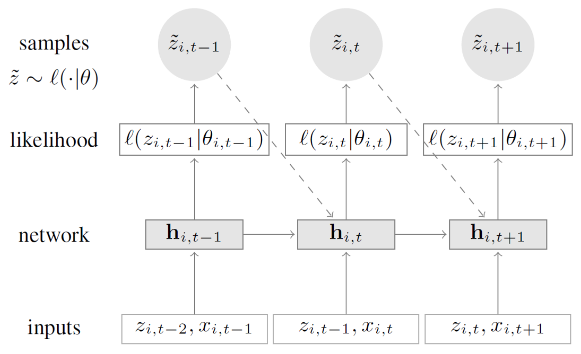


Figure II.15 – Summary of the model: At each timestep t, the inputs to the network are the covariates $x_{i,t}$, the target value at the previous timestep $z_{i,t-1}$, as well as the previous network output $\boldsymbol{h}_{i,t-1}$. The network output $\boldsymbol{h}_{i,t} = h(\boldsymbol{h}_{i,t-1}, z_{i,t-1}, x_{i,t}, \Theta)$ is then used to compute the parameters $\theta_{i,t} = \theta(\boldsymbol{h}_{i,t}, \Theta)$ of the likelihood $l(z|\theta)$, which is used for training the model parameters. When $z_{i,t}$ is not known (e.g. during prediction), a sample $\hat{z}_{i,t} \sim l(\cdot|\theta_{i,t})$ is fed back to the next step instead of the true value [Salinas et al., 2017]

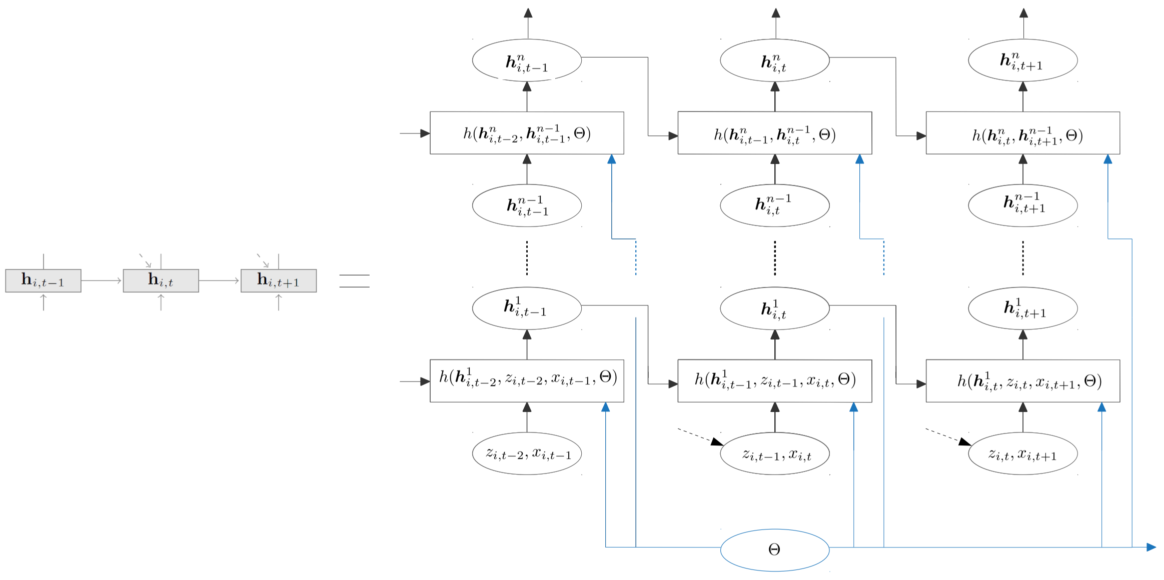

$\mathbf{h}_{i,t-1}$
$\mathbf{h}_{i,t}$
$\mathbf{h}_{i,t+1}$
$=$
$\boldsymbol{h}^n_{i,t-1}$
$\boldsymbol{h}^n_{i,t}$
$\boldsymbol{h}^n_{i,t+1}$
$h(\boldsymbol{h}^n_{i,t-2}, \boldsymbol{h}^{n-1}_{i,t-1}, \Theta)$
$h(\boldsymbol{h}^n_{i,t-1}, \boldsymbol{h}^{n-1}_{i,t}, \Theta)$
$h(\boldsymbol{h}^n_{i,t}, \boldsymbol{h}^{n-1}_{i,t+1}, \Theta)$
$\boldsymbol{h}^{n-1}_{i,t-1}$
$\boldsymbol{h}^{n-1}_{i,t}$
$\boldsymbol{h}^{n-1}_{i,t+1}$
$\boldsymbol{h}^1_{i,t-1}$
$\boldsymbol{h}^1_{i,t}$
$\boldsymbol{h}^1_{i,t+1}$
$h(\boldsymbol{h}^1_{i,t-2}, z_{i,t-2}, x_{i,t-1}, \Theta)$
$h(\boldsymbol{h}^1_{i,t-1}, z_{i,t-1}, x_{i,t}, \Theta)$
$h(\boldsymbol{h}^1_{i,t}, z_{i,t}, x_{i,t+1}, \Theta)$
$z_{i,t-2}, x_{i,t-1}$
$z_{i,t-1}, x_{i,t}$
$z_{i,t}, x_{i,t+1}$
Θ

Figure II.16 – Details of the RNN used in DeepAR with n the number of layers, $h(\cdot)$ a LSTM or GRU cell and $\boldsymbol{h}_{i,j}$ on the left-hand side are equal to $\boldsymbol{h}^n_{i,j}$ on the right-hand side for all j.

LSTM and GRU cells

The Long Short-Term Memory (LSTM) cell is a recurrent cell which has been explained previously in Section 3.1. Inspired by the LSTM cell, a Gated Recurrent Unit (GRU) is another type of recurrent cell. It was introduced by [Cho et al., 2014] and was thought to simplify the LSTM cell in language processing tasks. In this cell, only two gates are used: a reset gate $\boldsymbol{r}_t$ and an update gate $\boldsymbol{z}_t$ (cf. Figure II.17).

Let σ be an activation function, $\odot$ be the element-wise product between matrices, $\boldsymbol{W}_{(.)}$ and $\boldsymbol{b}_{(.)}$ be the weights and biases shared across all GRU cells. Considering the t-th GRU cell of a recurrent neural network, the following operations are done. First, parts of the previous hidden state are reset to the input values using $\boldsymbol{r}_t$, thus dropping irrelevant information:

$$\boldsymbol{r}_t = \sigma(\boldsymbol{W}_r^T[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_r), \tag{II.44}$$

$$\tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W}_h^T[\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_h). \tag{II.45}$$

Finally, how much the previous state will impact the new one is controlled by the update gate $\boldsymbol{z}_t$:

$$\boldsymbol{z}_t = \sigma(\boldsymbol{W}_z^T[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_z), \tag{II.46}$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t \tag{II.47}$$

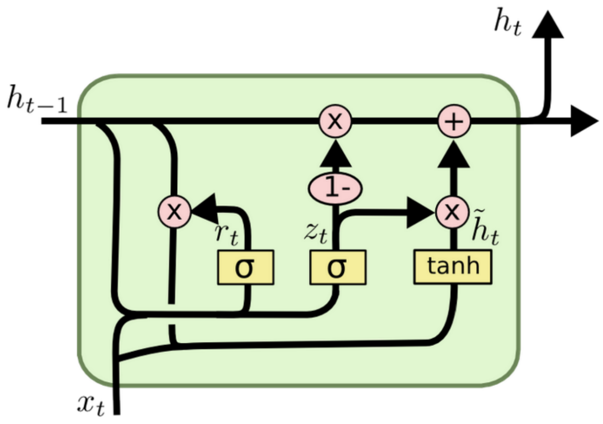


Figure II.17 – GRU cell [Tixier, 2018]

Likelihood model

The likelihood $l(z|\theta)$ should represent at best the statistical properties of the data. It can be chosen among any possible likelihood; e.g. Gaussian, negative-binomial, Bernoulli, etc..

For example, in the Gaussian likelihood case, the mean and the standard deviation are the parameters $\theta = (\mu, \sigma)$. Those are respectively given by the network output and a softplus activation to the network output to ensure $\sigma > 0$:

$$\begin{aligned} l_G(z|\mu, \sigma) &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(z-\mu)^2}{2\sigma^2}\right) \\ \mu(\boldsymbol{h}_{i,t}) &= \boldsymbol{w}_\mu^T \boldsymbol{h}_{i,t} + b_\mu \\ \sigma(\boldsymbol{h}_{i,t}) &= \log(1 + \exp(\boldsymbol{w}_\sigma^T \boldsymbol{h}_{i,t} + b_\sigma)) \end{aligned}$$

Loss function

The parameter Θ of the model, which consists of the parameters of the RNN $h(\cdot)$ and the parameters of $\theta(\cdot)$, can be learned by maximizing the log-likelihood

$$\mathcal{L} = \sum_{i=1}^{N} \sum_{t=t_0}^{T} \log l(z_{i,t}|\theta(\boldsymbol{h}_{i,t})), \tag{II.48}$$

with the dataset of time series $\{\boldsymbol{z}_{i,1:T}\}_{i:1,\ldots,N}$ and associated covariates $x_{i,1:T}$ known. In contrast to state space models with latent variables, no inference is required to compute $(II.48)$ as $\boldsymbol{h}_{i,t}$ is a deterministic function of the input. Thus it can be directly optimized with stochastic gradient descent with respect to Θ.

Data augmentation

Multiple training instances are created for each time series in the dataset by taking parts of it by sliding a window of the starting point and ending point on the series. The total window length $T = \text{prediction\_length} + \text{context\_length}$ stays constant. During the sliding, the prediction range is always covered by the ground truth data; however, it is not the case for the context range. For example, if a time series ranges from 2013-01-01 to 2017-01-01, it is possible to start the window on 2012-01-01. The unobserved data are set with zeros. Augmenting the data this way ensures that information about absolute time is only available to the model through covariates but not through the relative position of $z_{i,t}$.

Scale handling

To insure the method effectiveness in real-world forecasting, the time series are considered to have a wide range of values. However, the imbalance in the data causes a stochastic optimization procedure that picks training instances uniformly at random to visit the small number time series with a large scale very infrequently, which can result in underfitting on those time series unless the model is trained for a large number of epochs. To solve this problem, the sampling of the examples is non-uniform during training. Nevertheless, this is not necessary to obtain good performances.

Covariates

The covariates $x_{i,t}$ used by the model can be item-dependent, time-dependent or both. Those are used to provide additional information about the item or the time point to the model. For example, in single-item forecasting methods, time-dependent features are often used for modeling seasonality, by using time period indicator features (e.g. "day of the week"). They can also be used to include covariates that one expects to influence the outcome (e.g. price or promotion status in the demand forecasting setting), as long as the features' values are also available in the prediction range.

3.3 Models implementation

Temporal Fusion Transformers implementation

An open-source implementation of the Temporal Fusion Transformers (TFT) model which was released with the paper [Lim et al., 2019] can be found on GitHub[3]. This implementation is based on TensorFlow. The main hyperparameters which can be tuned are: the early stopping criteria (i.e. stopping after x iterations after which the loss on the validation set has not decrease), the hidden layer size of networks such as gated residual networks, embeddings, InterpretableMultiHeadAttention, etc., and the number of heads in the attention layer. In this implementation, even though the model architecture was thought to handle multivariate forecasting, weirdly it was not fully implemented. Thus, part of the work done in this part of the thesis was to modify the code found on GitHub to be able to apply multivariate forecasting during this task.

[3]GitHub URL: https://github.com/google-research/google-research/tree/master/tft

DeepAR implementation

DeepAR is a model introduced by [Salinas et al., 2017], however, no code was released with the paper. Fortunately, it has been implemented in Gluon Time Series (GluonTS) based on this paper. GluonTS is a toolkit developed by Amazon scientists based on the Gluon framework [Alexandrov et al., 2019]. Its goal is to regroup all tools necessary to build deep learning models for time series forecasting and anomaly detection. Many state-of-the-art models are preimplemented within it, nevertheless, as a recent framework, some bugs are still to be corrected.

In its implementation, DeepAR's architecture can be tuned based on three main hyperparameters. Firstly, the cell type can be chosen between a LSTM or a GRU cell. Secondly, the output size of the LSTM/GRU cell (i.e. its hidden state size) can be fixed. Lastly, the number of layers of cells stacked on top of each other can be modified. The Figure II.16 shows the details of the RNN architecture with the output size being the size of $\boldsymbol{h}_i^j\ \forall i, j$ and n the number of layers. However, this implementation is not exempt from bugs. Due to this, the multivariate forecasting is not available.

Moreover, even if it is possible to do a validation test at each training epoch[4], and if the parameters with the lowest validation error are taken at the end, no early stopping is implemented. It makes overfitting more difficult to handle and the training potentially longer than necessary.

In Section 2.3, time features were described. However, the DeepAR model uses its own time features of the future timesteps instead instead of those previously defined. In this study, the time features are the following:

- *hour of the day :* hour of the day encoded as value between [-0.5, 0.5],
- *day of the week :* day of the week encoded as value between [-0.5, 0.5]
- *day of the month :* day of the month encoded as values between [-0.5, 0.5],
- *day of the year :* day of the year encoded as value between [-0.5, 0.5].

4 Forecasting evaluation metrics

As explained previously the goal of this task is to forecast multiple future timesteps of target time series. In addition to predicting the exact values (i.e. point forecasting), confidence intervals are also provided. Those are based on percentiles, either directly provided as output of the model (e.g. TFT model) or computed based on a fixed number of samples out of a probability distribution (e.g. DeepAR model). To evaluate the point and interval forecasts, the q-Risk metric has been used in [Salinas et al., 2017; Lim et al., 2019]. However, due to shortcomings explained in Section 4.1, a second metric is applied. The latter is the Mean Scaled Interval Score (MSIS). It is used in the M4 competition, an annual open competition whose purpose is to increase the forecasting accuracy and advance the theory and practice of forecasting [Makridakis et al., 2020].

4.1 q-Risk

The q-Risk is based on the quantile loss (QL):

$$QL(y, \hat{y}, q) = q \max(0, y - \hat{y}) + (1 - q) \max(0, \hat{y} - y), \tag{II.49}$$

[4]In terms of artificial neural networks, an epoch refers to one cycle through the full training dataset [DeepAI, 2020].

with it, the q-risk for a quantile q is defined by:

$$q - \text{Risk} = \frac{2\sum_{i=1}^{N}\sum_{t=t_0}^{\tau} QL(y_t^{(i)}, \hat{y}_t^{(i)}, q)}{\sum_{i=1}^{N}\sum_{t=t_0}^{\tau} y_t^{(i)}}, \tag{II.50}$$

where N is the number of testing samples, t_0 and τ are respectively the first and last forecasting times, $y_t^{(i)}$ is the values of time t of the i-th testing samples and $\hat{y}_t^{(i)}$ is the q-th forecast of time t of the i-th testing samples.

The q-Risk ranges from 0 to $+\infty$, with 0 being the best score and $+\infty$ the worst. Its value represents an idea of the mean distance between the forecast and the ground truth, normalized by the mean value of all testing samples. However, what this metric fails to do is to take into account how much the ground truth values changes over time. For example, let the mean be large compared to the change in ground truth values: a mean of 10000 and values alternating randomly between 9999 and 10001. Then let the 0.9-forecast consistently make a prediction greater than the ground truth by 5. The 0.9-Risk is $0.9 - \text{Risk} = \frac{2(0+0.5)}{10000} = 0.0001$. Now, let the mean be small compared to the change in ground truth value: mean of 0.0001 while values alternate randomly between -10000 and 10000. Again, let the 0.9-forecast consistently be greater than the ground truth by 5. The 0.9-Risk becomes: $0.9 - \text{Risk} = \frac{2(0+0.5)}{0.0001} = 10000$. Now comparing the two, on the one hand, the forecast are bad as distance between the ground truth minimum and maximum is 2 but the prediction is always greater by 5. This situation gives $0.9 - \text{Risk} = 0.0001$. On the other hand, a good prediction where ground truth alternates between points distanced by 20000 but the forecasts are only greater by 5: $0.9 - \text{Risk} = 10000$. The second case has a much larger q-Risk while its prediction is more accurate, which is counter-intuitive. These two simple examples demonstrate that an evaluation of quantiles prediction based on the q-Risk alone is prone to interpretation error. Due to this, another metric is also used: Mean Scaled Interval Score.

4.2 Mean Scaled Interval Score

Confidence intervals can be evaluated with the Mean Scaled Interval Score (MSIS). It is based on the interval width, its distance to the ground truth and the mean absolute error of the one-step naive forecast method (the denominator of (II.51)). This naive forecast method uses the true value from the prior season as the forecast. For example, if the time series displays a monthly seasonality, each prediction of the naive method is equal to the values at the same time during the previous month. Thus, the error of this forecast method reflects how much the time series changes from one season to another, which can be an indicator of the time series predictability. It is computed with the following equation:

$$MSIS = \frac{1}{N}\sum_{i=1}^{N} \frac{\sum_{t=t_0}^{\tau}\left((U_t^{(i)} - L_t^{(i)}) + \frac{2}{a}(L_t^{(i)} - y_t^{(i)})\mathbf{1}_{(y_t^{(i)} < L_t^{(i)})} + \frac{2}{a}(y_t^{(i)} - U_t^{(i)})\mathbf{1}_{(y_t^{(i)} > U_t^{(i)})}\right)}{\sum_{t=t_0}^{\tau} |y_t^{(i)} - y_{t-m}^{(i)}|} \tag{II.51}$$

where N is the number of testing samples, $x_t^{(i)}$ is the x value of time t of the i-th testing samples, t_0 and τ are respectively the first and last forecasting time, U is upper bound of the confidence interval, L is the lower bound of the confidence interval, y is the ground truth, m is the seasonality (e.g. if there is a seasonality of 1 day and the timestep is one minute, $m = 24 * 60$), and a which should depend on the percentage of the interval (e.g.

for a 95% interval, $a = 0.05$) is arbitrarily fixed to 0.05 to be able to compare different percentage intervals with the same metric.

The MSIS can be decomposed in three parts based on the three terms of the nominator of equation (II.51). The first one depends on the width of the interval while the second and third ones penalize the distance between the interval and the ground truth when the latter is outside of the bounds. Thus, the MSIS has a good score (close to zero) when the interval is tightly surrounding the ground truth. Moreover, the higher the error of the naive method, indicating a higher difficulty to predict, the higher the denominator, thus reducing the value of the MSIS in consequence. Altogether, this metric gives a good evaluation of confidence intervals by taking into account the complexity of the time series, the width of the interval and the distance to the ground truth when the interval does not contain it.

5 Evaluation of the solution

The two considered models, DeepAR and TFT, are evaluated in this section based on their predictions of three datasets (cf. Section 2.3). The experimental setup which is shared for each dataset evaluation is first described before the dataset experiments.

5.1 Shared experimental setup

Models configurations

The models are trained to take as input all features of a certain number of consecutive timesteps and the time features of all the timesteps to forecast. The time features are the only features of the future timesteps which are given as input as they are the only ones known. Then, for each forecast horizon, the models output the point forecast of each target feature and the 5-th and 95-th percentile in order to construct a 90% confidence interval.

For each evaluation, four configurations are considered, two per model. The second one for each model differs from the first one by its training dataset which has been weighted to accentuate the importance of timesteps of interest. As the forecasting goal for the experiments data is to be able to detect when an abrupt change of value will occur, timesteps are considered ‘interesting’ if the time series to predict goes through a large change of value. However, weighting timesteps directly is not possible in the current implementation of TFT. Instead, for both models, it was done by reducing the learning and validation datasets to one of their subsets. The subsets are created by taking the maximal number of interest points inside the dataset. Then if I interest points are found, I non-interest points and I points taken uniformly at random without replacement are added to the subsets.

Those configurations go through a round of hyperparameter optimization after which the final model is obtained by retraining with the best hyperparameter set, if necessary. The optimization is done by training a model with early stopping[5] (if available) with a given hyperparameter set before evaluating it. Once each set has been tested, the one with the lowest error is chosen. However, a perfect optimization search takes time. It is thus approximated by testing only N sets of the search space, taken randomly without

[5]Early stopping is the action of stopping the training after x epochs where the loss on the validation test has not decrease. However it is not implemented in DeepAR.

replacement. N has been arbitrarily fixed to 15. During the optimization, models are trained on a learning set and use a validation set for the early stopping, which is only available for TFT. When the data quickly evolves with time, it is important once the final hyperparameters are obtained for the model to be retrained to a larger learning set, closer to the testing set, to insure that the data used is not 'expired'. This new learning set is obtained using the previous learning set and a large part of the previous validation set. It is only applied when the data requires it.

The TFT configurations are trained with early stopping during its hyperparameter optimization and without for the final training (cf. Tables II.4 and II.5). The parameters to optimize are those for which it is difficult to fix a value without testing by trial and error which is the most appropriate. For TFT, those are the following.

- *The context length :* It is the number of previous timesteps required to do the most accurate forecasts. The context lengths tested depend on the data and is given by the user.
- *The neural network hidden layer size :* It is used as the dimension for the GLUs, the categorical variable embedding networks, the variable selection networks, the LSTM cells, the gated residual networks, the multi-head attention layer, and all other 'neural network blocks' of the model.
- *The dropout rate :* It is the dropout rate used in all 'network network blocks' beside the LSTM cell.
- *The learning rate :* It is the initial rate of the ADAM optimizer used [Kingma and Ba, 2014].
- *The number of attention heads :* It is the number of heads which allows the model to jointly attend to information from different representation subspaces at different positions [Vaswani et al., 2017].

Hyperparameter	Value
Early stopping (number)	15
Epochs, with early stopping (number)	200
Epochs, without early stopping (number)	100
Batch size	32
Batches/epoch	number of obs./batch size

Table II.4 – TFT fixed training hyperparameters

Hyperparameter	Values
Context length	*dataset dependent*
Neural network hidden layer size	[5, 10, 20, 40, 100, 300]
Dropout rate	[0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.9]
Learning rate	[1e-4, 1e-3, 1e-2]
Attention heads (number)	[1, 4]

Table II.5 – TFT hyperparameters to optimize

On the other hand, even if the validation loss denotes the best training epoch, the DeepAR configurations are trained without early stopping during its optimization since it is not implemented (cf. Tables II.6 and II.7). Then the final models are trained on the best parameters without early stopping or validation loss on a new training set if necessary. The optimization is done on the following parameters.

- *The context length and the learning rate :* It is the same as explained above for TFT.
- *The number of stacking layers :* The number of layers in the recurrent neural network.
- *The cell type :* The type of cell in the recurrent neural network.
- *The cell hidden state size :* The hidden state size of the cell in the recurrent neural network.
- *The number of gaussians :* The number of gaussians in the mixture of gaussians considered as the probability distribution of each timestep.
- *The dropout rate :* The output of each LSTM cell is feed to a Zoneout cell which uses this dropout rate [Krueger et al., 2016]. Zoneout is an application of dropout where the values are reset to their previous state ($h_t = h_{t-1}$) instead of being dropped out ($h_t = 0$).

Hyperparameter	Value
Epochs, with early stopping (number)	200
Epochs, without early stopping (number)	100
Batch size	32
Batches/epoch	100

Table II.6 – DeepAR fixed training hyperparameters

Hyperparameter	Values
Context length	*dataset dependent*
Stacking layers (number)	[1, 2, 4]
Cell type	[GRU, LSTM]
Cell hidden state size	[2, 5, 15, 45, 70]
Gaussians (number)	[1, 3, 10]
Dropout rate	[0.1, 0.3, 0.5]
Learning rate	[1e-4, 1e-3, 1e-2]

Table II.7 – DeepAR hyperparameters to optimize

Dataset rescaling and outliers removal

The outliers of all features (except time features) are removed by applying the interquartile range method (cf. Section 2.2).

Then, the datasets are rescaled during pre- and post-processing with a standardization for TFT configurations and with a normalization followed by standardization for DeepAR configurations (cf. Section 2.2). The normalization was added for DeepAR as during the model validation, it was observed that it struggled to keep the output within the range of values seen during training. As the probability distribution of outputs takes the form of a mixture of gaussians, it is not impossible for points to be sampled very far from the main range. This explains why normalization was necessary for DeepAR but not for TFT.

Evaluation method

To evaluate them, each final model from the previous configurations is asked to forecast timesteps of a testing set. The percentiles requested are the [5-th, 50-th, 95-th], which give the mean forecast and the 90-confidence intervals. Then the models are evaluated based on three things:

- *the q-Risk :* measuring the q-loss for each quantile q,
- *the MSIS :* measuring the MSIS of each confidence interval,
- *the coverage :* measuring the percentage of time that a confidence interval contains the true value.

As explained before, the q-Risk allows to easily compare quantile forecasts of different models on the same data (cf. Section 4). However, it is more difficult to evaluate their accuracy as it normalizes the metric based on the mean value which might be much larger than the change occurring in it. Instead, the MSIS normalizes based on the naive forecasting error, which makes it easier to evaluate confidence intervals. Moreover, the coverage gives an additional information about confidence intervals which helps understanding the performance of the models.

A good confidence interval is one which is tightly surrounding the true value. Thus, one of the difficulties of the model is to understand when changes are happening and how big those are. One thing to avoid is to have a large constant interval containing all possible values of the time series. However, when the testing set contains mostly constant values with a few large peaks, corresponding to rare events to predict, averaging the metrics does not evaluate well those events and the above cannot be checked correctly. Thus the testing set is divided in three based on the change in target value. Let d be the difference between the minimum and maximum target value of the time series. The three subsets contain the time series windows where a difference in target values between the last context timestep and any prediction timestep are greater than $0.3d$ for the first subset, greater than $0.1d$ and not in subset 1 for the second one and all those not in subsets 1 or 2 for the third one. This way, it becomes easy to see if the intervals are too big when the target value stays more or less constant or to see if the model fails to predict when substantial changes are happening for example.

5.2 Evaluation based on dataset ds3

This section evaluates the DeepAR and TFT models for each configuration (cf. Section 5.1) on the dataset ds3 (cf. Section 2.3). On top of the shared experimental setup explained previously, below is the specific one applied here followed by the experiment results.

Specific experiment setup for dataset ds3

The dataset ds3 contains time series which evolves as time goes by. Thus, even if close to 5 years of hourly measurements are available, only two years are used in the training set to train the model only on the most recent history. The dataset ds3 is divided as follows, with e being the end date of ds3:

- *learning set :* contains timesteps whose date is within $[e - 4\text{years}, e - 2\text{years}[$,
- *validation set :* contains timesteps whose date is within $[e - 2\text{years}, e - 1\text{year}[$,
- *testing set :* contains timesteps whose date is within $[e - 1\text{year}, e]$,

Then, out of the 56 features, 3 are designed as target values: the oil, gas and water average flow. As DeepAR implementation didn't allow multivariate forecasting, three different models have been trained for each target feature.

Finally, the prediction length was set to 2 for a simple multistep forecast and the context length to optimize was chosen within the arbitrary set [2, 6, 12, 24].

Experiment results

The Q-Risk is displayed in Table II.8. Firstly, the TFT models are consistently having a larger error when the change of value within the prediction window is bigger for the target value. This could be explained by them being incapable of predicting when those changes are going to happen. The worst q-Risks in the [30, inf[percentage interval change are around 0.8. It corresponds to errors of about 80% of the mean target values. In Figures II.18, II.19 and II.20, it is observed that the intervals are, indeed, not containing the true values when they go down to 0. Indeed, even if the interval seems to be close to the true value, it is probably just shifted by 1 and not much better than the naive forecast prediction. Comparing the two configurations, the first one, which uses all timesteps during training, always has an error close to or lower than the second one. This shows that weighting samples did not improve the model accuracy.

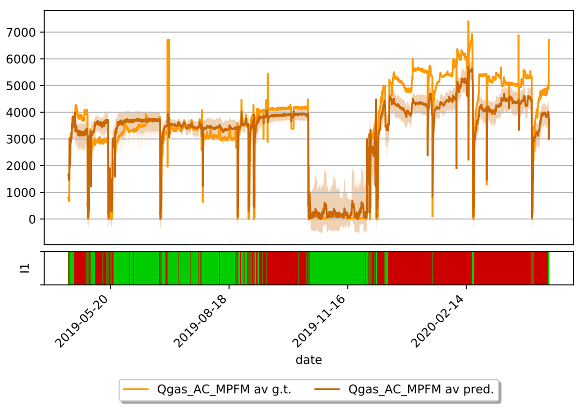
Secondly, the DeepAR models show different patterns in the score compared to TFT. For example, its lowest errors are not only occurring when the change in values are low but also occurring when they are high. Although, this statement is only valid for the 5-th and 95-th percentiles. It is explained by the higher coverage gained by the larger intervals compensating for the low accuracy of the 50-th percentile (cf. Figures II.18, II.19 and II.20). This last one does not necessarily have high value for the q-Risk of the [30, inf[interval since instead of predicting the value, most of the time, it gives a constant value minimizing the error.

Finally, comparing the TFT and DeepAR configurations on their q-Risk, while no clear winner stands out, both have high errors in respect to the mean target values, which tends to indicate poor performance. By analysing Table II.9, a clearer picture of the situation can be made. Focusing on the interval [0, 10[, the obtained MSIS are huge. As changes in value between two timesteps are little, the naive forecast method makes tiny errors. Thus the width of the interval and distance to the true value are highly penalized. In the one hand, TFT configurations are mostly penalized by the distance to the ground truth as its coverage is low, while on the other, it is their interval widths which increases the DeepAR configurations' error. Then, the MSIS is much lower for the percentage change intervals [10,30[and [30, inf[as the naive forecast error increases. In particular, in [30, inf[of the water flow (Qwat_AC_MPFM av) forecasts of the second DeepAR configurations, the low MSIS indicates good confidence intervals. Indeed, from Figure II.20, intervals of width close to 3000 can be seen containing the points with the highest change in value between timesteps. Although, 3000 seems large compared to a value change of about 1000 (causing an equal naive forecasting error), the width is not too bad.

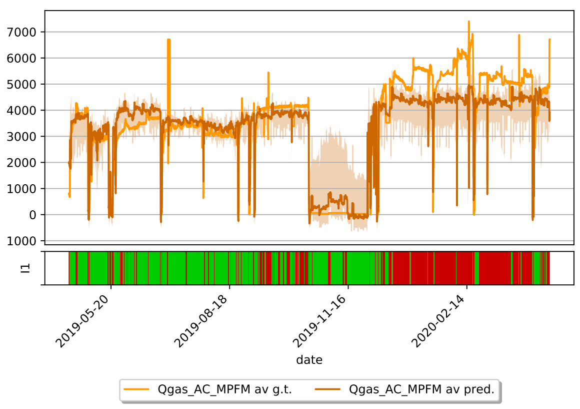
From the results, it can be seen that DeepAR configurations, with their loose confidence intervals, are better when large changes of values occur. However, TFT configurations, with their tight intervals, are better in the other situations. Unfortunately, neither the tight intervals incapable of forecasting peaks of TFT configurations or the loose intervals encompassing most values of the DeepAR configurations are acceptable solutions.

			TFT		DeepAR	
Target	Percentile	Change (%)	Conf. 1	Conf. 2	Conf. 1	Conf. 2
Qgas_AC_MPFM av	5	[0, 10[	**0.034**	0.040	0.047	0.085
		[10, 30[	0.227	0.207	0.282	**0.082**
		[30, inf[	0.606	0.430	0.311	**0.085**
	50	[0, 10[	0.152	**0.151**	0.275	0.230
		[10, 30[	**0.375**	0.437	0.762	0.478
		[30, inf[	0.596	0.603	0.511	**0.372**
	95	[0, 10[	0.125	**0.114**	0.115	0.406
		[10, 30[	0.205	0.213	**0.169**	0.293
		[30, inf[	0.369	0.356	**0.236**	0.505
Qoil_AC_MPFM av	5	[0, 10[	**0.045**	0.147	0.057	0.093
		[10, 30[	0.313	0.434	0.432	**0.092**
		[30, inf[	0.380	0.361	0.123	**0.094**
	50	[0, 10[	**0.151**	0.333	0.167	0.342
		[10, 30[	**0.456**	0.519	0.669	0.912
		[30, inf[	0.647	0.703	**0.388**	0.468
	95	[0, 10[	**0.046**	0.062	0.106	0.104
		[10, 30[	0.123	**0.098**	0.167	0.160
		[30, inf[	0.522	0.400	**0.248**	0.281
Qwat_AC_MPFM av	5	[0, 10[	0.031	0.043	**0.026**	0.058
		[10, 30[	0.214	0.173	0.101	**0.078**
		[30, inf[	0.951	0.828	0.340	**0.071**
	50	[0, 10[	**0.105**	0.159	0.133	0.210
		[10, 30[	0.320	0.382	0.462	**0.311**
		[30, inf[	0.761	0.753	0.493	**0.437**
	95	[0, 10[	**0.066**	0.144	0.082	0.071
		[10, 30[	0.128	0.209	0.116	**0.090**
		[30, inf[	0.227	0.214	0.138	**0.106**
Mean	5	[0, 10[	**0.037**	0.077	0.043	0.079
		[10, 30[	0.251	0.271	0.272	**0.084**
		[30, inf[	0.646	0.540	0.258	**0.083**
	50	[0, 10[	**0.136**	0.214	0.192	0.261
		[10, 30[	**0.384**	0.446	0.631	0.567
		[30, inf[	0.668	0.686	0.464	**0.426**
	95	[0, 10[	**0.079**	0.107	0.101	0.194
		[10, 30[	0.152	0.173	**0.151**	0.181
		[30, inf[	0.373	0.323	**0.207**	0.297

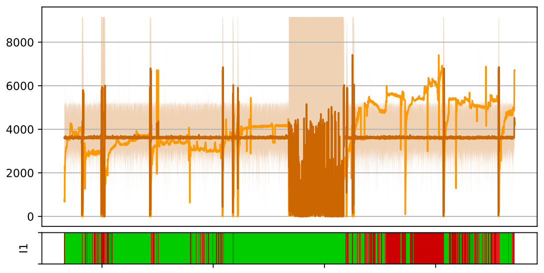
Table II.8 – Q-Risk obtained on the ds3 testing set by the final models of each configuration where the ones labelled 1 are the basic configurations while the ones labelled 2 only use the interest points during training (cf. Section 5.1). The results are arranged by the percentage of change in the target values of the prediction window (cf. Section 5.1).



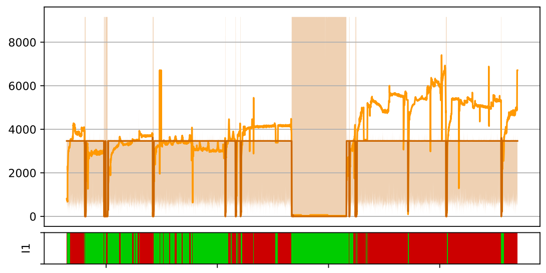
`(a)` TFT configuration 1: coverage= 0.445



`(b)` TFT configuration 2: coverage= 0.568

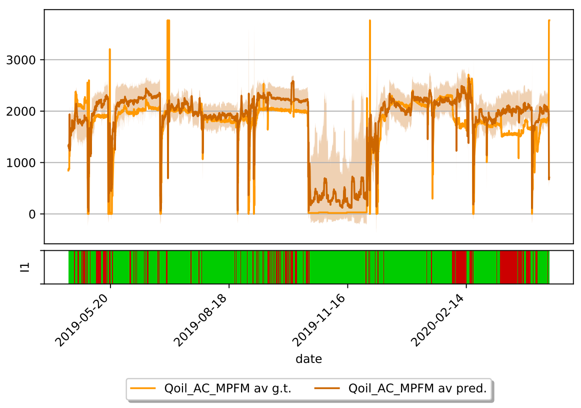


`(c)` DeepAR configuration 1: coverage= **0.739**

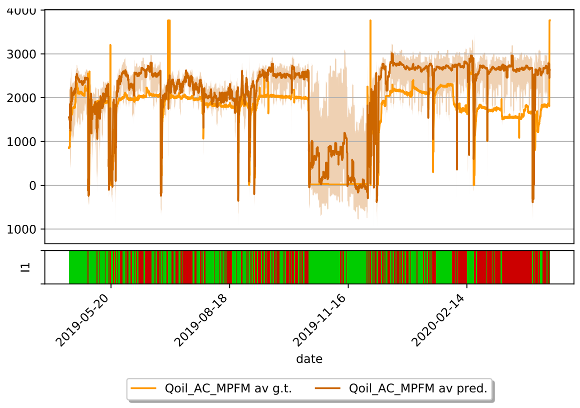


`(d)` DeepAR configuration 2: coverage= 0.425

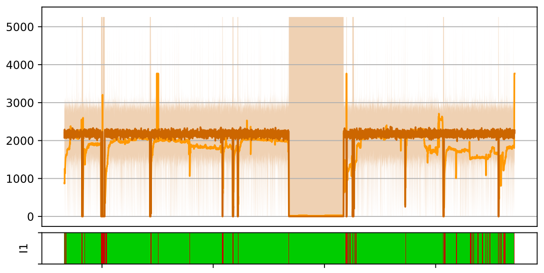
`Figure II.18` – One step ahead forecasts of the gas flow feature (Qgas_AC_MPFM_av), stacked together, on the testing set. Green indicates when the ground truth is contained in the interval and red when it is not. The notations g.t. and pred. respectively stand for ground truth (the true values of the time series) and prediction (the forecast of the time serie). The interval encompassing the forecast is the 90% confidence interval.



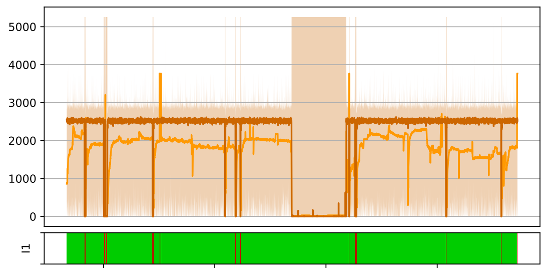
`(a)` TFT configuration 1: coverage= 0.778



`(b)` TFT configuration 2: coverage= 0.542

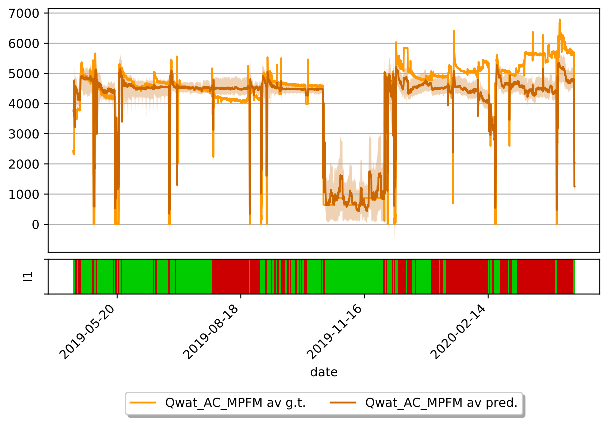


`(c)` DeepAR configuration 1: coverage= 0.938

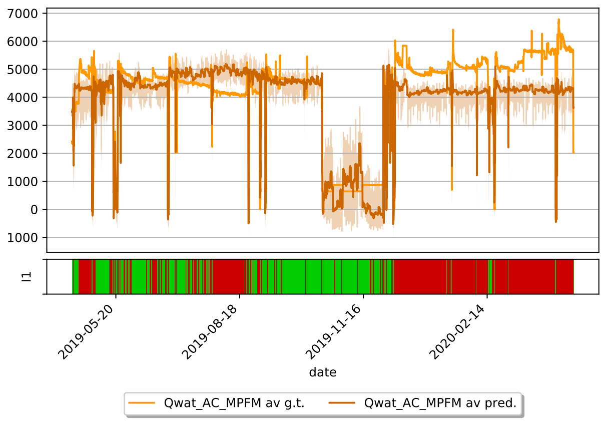


`(d)` DeepAR configuration 2: coverage= **0.984**

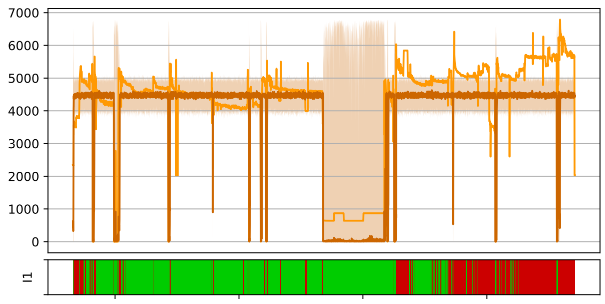
`Figure II.19` – One step ahead forecasts of the oil flow feature (Qoil_AC_MPFM_av), stacked together, on the testing set. Green indicates when the ground truth is contained in the interval and red when it is not. The notations g.t. and pred. respectively stand for ground truth (the true values of the time series) and prediction (the forecast of the time serie). The interval encompassing the forecast is the 90% confidence interval.



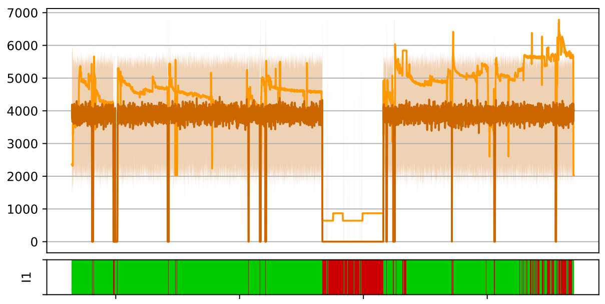
(a) TFT configuration 1: coverage= 0.549



(b) TFT configuration 2: coverage= 0.442



(c) DeepAR configuration 1: coverage= 0.680



(d) DeepAR configuration 2: coverage= **0.773**

Figure II.20 – One step ahead forecasts of the water flow feature (Qwat_AC_MPFM_av), stacked together, on the testing set. Green indicates when the ground truth is contained in the interval and red when it is not. The notations g.t. and pred. respectively stand for ground truth (the true values of the time series) and prediction (the forecast of the time serie). The interval encompassing the forecast is the 90% confidence interval.

		TFT		DeepAR	
Target	Change (%)	Conf. 1	Conf. 2	Conf. 1	Conf. 2
Qgas_AC_MPFM av	[0, 10[	**478639.532**	665640.783	2270827.935	2264904.507
	[10, 30[	20.805	**19.51**	24.013	20.096
	[30, inf[	31.525	24.798	**15.802**	16.903
Qoil_AC_MPFM av	[0, 10[	**16326415.24**	25399123.092	42804850.136	42716145.629
	[10, 30[	20.64	25.238	29.911	**7.723**
	[30, inf[	29.376	23.753	10.041	**8.903**
Qwat_AC_MPFM av	[0, 10[	**9651344.705**	14082181.604	32186998.104	151794534.084
	[10, 30[	18.992	21.751	13.193	**9.79**
	[30, inf[	32.424	28.571	12.849	**3.186**
Mean	[0, 10[	**8818799.826**	13382315.160	25754225.392	65591861.407
	[10, 30[	20.146	22.166	22.372	**12.536**
	[30, inf[	31.108	25.707	12.897	**9.664**

Table II.9 – MSIS obtained on the ds3 testing set by the final models of each configuration where the ones labelled 1 are the basic configurations while the ones labelled 2 only use the interest points during training (cf. Section 5.1). The results are arranged by the percentage of change in the target values of the prediction window (cf. Section 5.1).

5.3 Evaluation based on dataset ds4

This section evaluates the DeepAR and TFT models for each configuration (cf. Section 5.1) on the dataset ds4 (cf. Section 2.3).

Specific experiment setup for dataset ds4

The dataset ds4 contains time series which evolve quickly and only the recent data are useful. Thus, not only the training set is restricted to recent timesteps only but after the hyperparameter optimization, the model is retrained with a training set closer to the testing one. The dataset ds4 is divided as follows, with e being the end date of ds4:

- *learning set :* contains timesteps whose date is within $[e - 4\text{years}, e - 2\text{years}[$,
- *validation set :* contains timesteps whose date is within $[e - 2\text{years}, e - 1\text{year}[$,
- *learning set for final model :* contains timesteps whose date is within $[e-2.5\text{years}, e-1.5\text{year}[$,
- *validation set for final model :* contains timesteps whose date is within $[e - 1.5\text{years}, e - 1\text{year}[$,
- *testing set :* contains timesteps whose date is within $[e - 1\text{year}, e]$,

The target values are the same as for ds3: the oil, gas and water average flow. One model per target value is still trained for DeepAR as its implementation does not allow multivariate forecasting.

Finally, the prediction length is set to 2 for a simple multistep forecast and the context length to optimize is chosen within the arbitrary set [2, 6, 24].

Experiment results

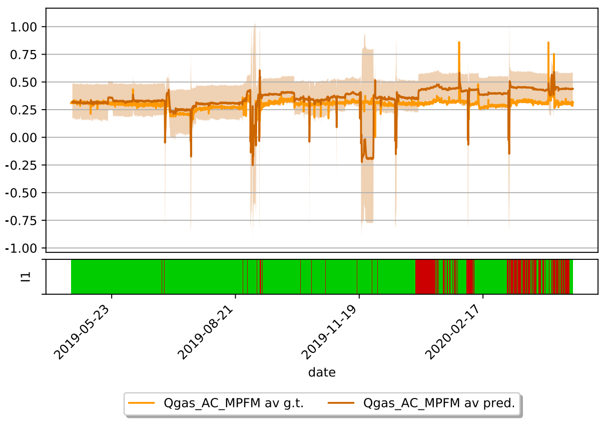
The Figures II.21, II.22 and II.23 display the one step ahead forecasts for the target features, while the quantitative results are contained in Tables II.10 and II.11.

For the first target feature, Qgas, Figure II.21 shows 3 qualitatively bad performances, configuration 2 of TFT and both ones of DeepAR, and 1 promising one, configuration 1 of TFT. The two DeepAR configurations forecast large confidence intervals enclosing most values. This results in a high coverage, but also demonstrates their inability to accurately predict future valuex. The MSIS confirms this with high score for the change percentage interval [0, 10[while the q-Risk highlights the error on the confidence interval upper bound and on the 50-th percentile for the first configuration. Looking at the TFT configurations, the second one forecasts almost the same interval for each timestep which, while large, is tighter than the confidence intervals of the DeepAR configurations. This allows it to get a better MSIS. The configuration 1 of TFT, the most promising of the four, with its small confidence intervals has the best MSIS for [0, 10[, however, it is still very high. Then for the [10, 30[and [30, inf[, the MSIS is actually worse than the second configuration. This indicates that while its confidence intervals are smaller, they are not accurate and have a worse performance than configuration 2 because of the distance between the intervals and the true values.

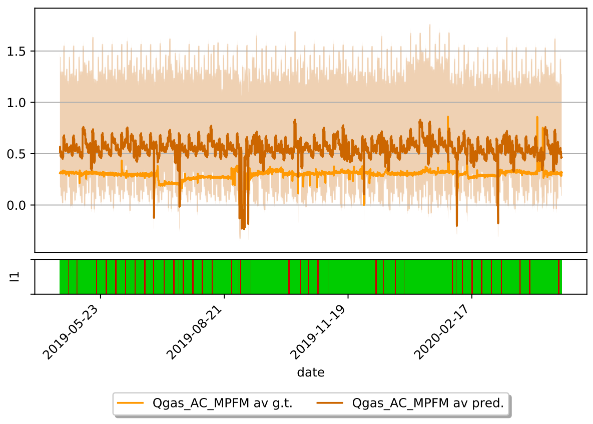
For the second target value, Qoil, shown in Figure II.22, a similar pattern than on the first target value is observed for the TFT configurations. However, both are less accurate than for Qgas, which is confirmed by their higher MSIS and q-Risk. Then the best of the two DeepAR configurations, based on MSIS and q-Risk, is the second one. However, it constantly forecasts the same interval and 50-th percentile, highlighting that it is incapable of predicting the future timestep values.

The forecasts for the last target feature, Qwat, are not any better than the previous ones. This time, all have the same problem as they constantly predict the value to be smaller than it actually is. This is because the mean value of Qwat increases with time, but the models apparently didn't learn that properly. On top of this, the same previous tendency to large intervals occurs here.

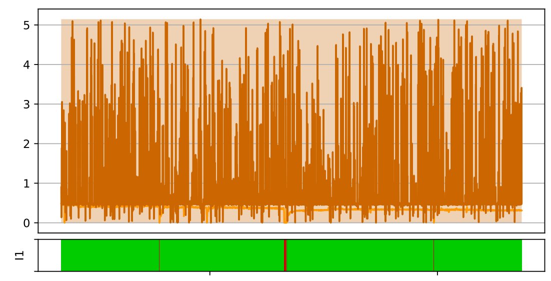
In summary, while the TFT configuration 1 seemed more promising with its smaller intervals, its accuracy was off and better score were obtained by the DeepAR models. Nevertheless, results stayed bad as all configurations struggled to predict the future timesteps.



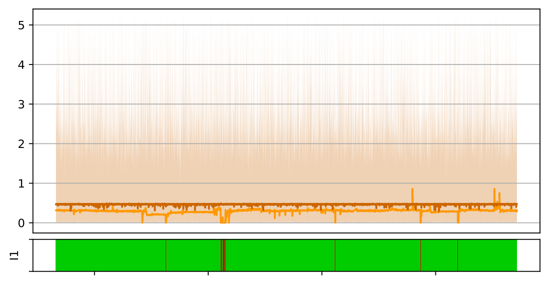
(a) TFT configuration 1: coverage= 0.826



(b) TFT configuration 2: coverage= 0.915

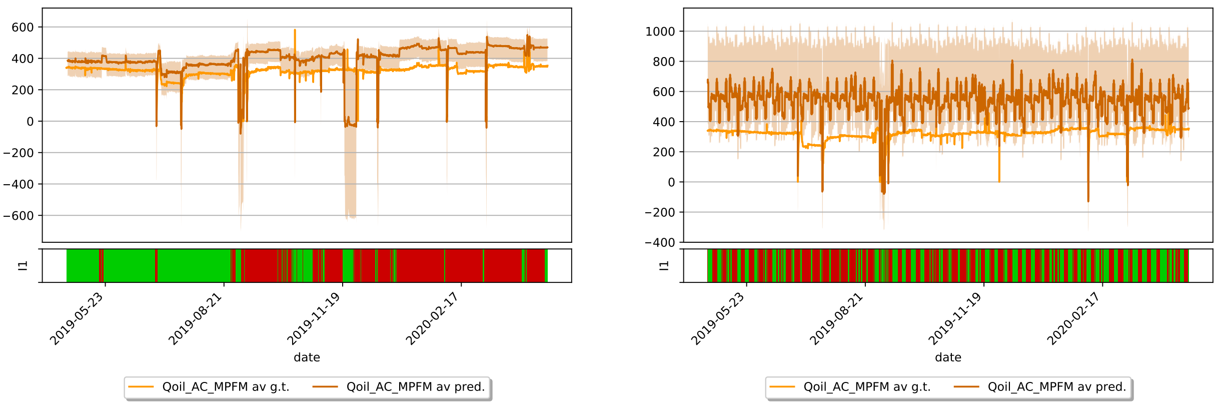


(c) DeepAR configuration 1: coverage= **0.993**

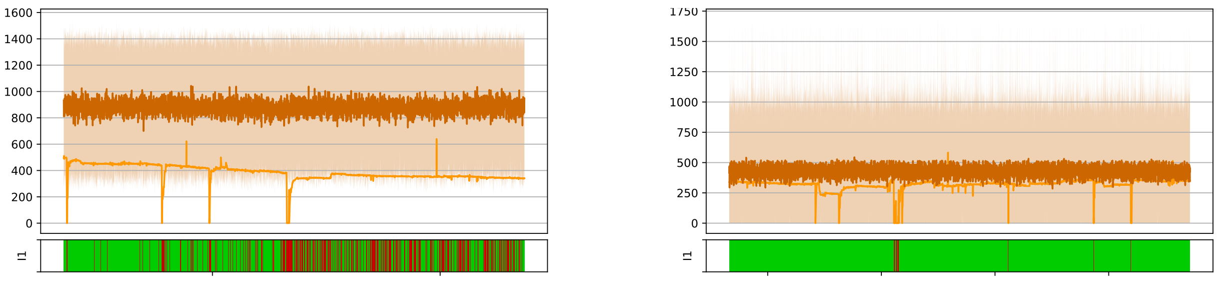


(d) DeepAR configuration 2: coverage= 0.989

Figure II.21 – One step ahead forecasts of the gas flow feature (Qgas_AC_MPFM_av), stacked together, on the testing set. Green indicates when the ground truth is contained in the interval and red when it is not. The notations g.t. and pred. respectively stand for ground truth (the true values of the time series) and prediction (the forecast of the time serie). The interval encompassing the forecast is the 90% confidence interval.

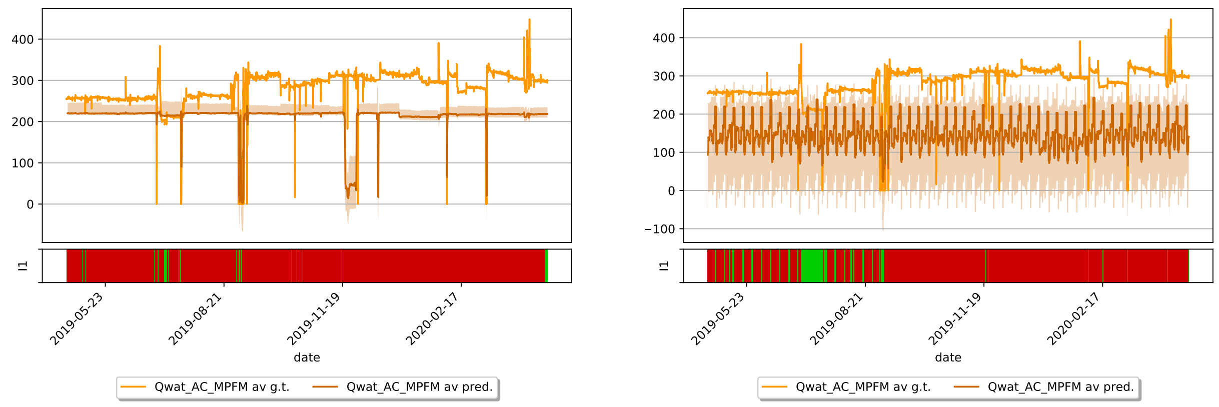


(a) TFT configuration 1: coverage= 0.449 (b) TFT configuration 2: coverage= 0.436

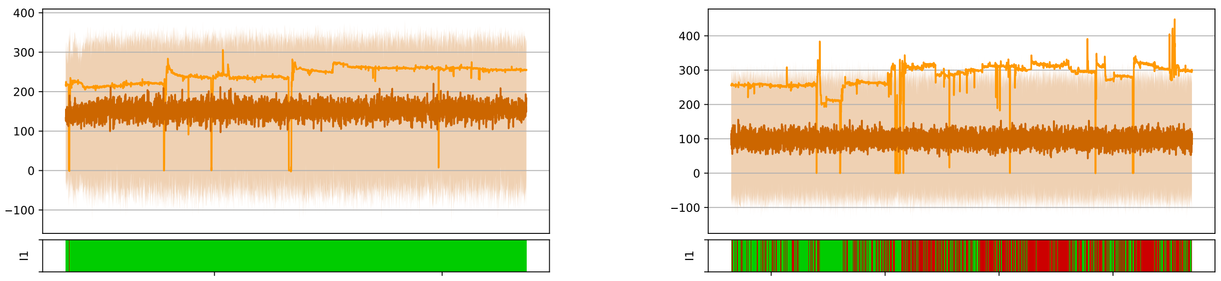


(c) DeepAR configuration 1: coverage= 0.730 (d) DeepAR configuration 2: coverage= **0.990**

Figure II.22 – One step ahead forecasts of the oil flow feature (Qoil_AC_MPFM_av), stacked together, on the testing set. Green indicates when the ground truth is contained in the interval and red when it is not. The notations g.t. and pred. respectively stand for ground truth (the true values of the time series) and prediction (the forecast of the time serie). The interval encompassing the forecast is the 90% confidence interval.



(a) TFT configuration 1: coverage= 0.031 (b) TFT configuration 2: coverage= 0.0115



(c) DeepAR configuration 1: coverage= **0.999** (d) DeepAR configuration 2: coverage= 0.433

Figure II.23 – One step ahead forecasts of the water flow feature (Qwat_AC_MPFM_av), stacked together, on the testing set. Green indicates when the ground truth is contained in the interval and red when it is not. The notations g.t. and pred. respectively stand for ground truth (the true values of the time series) and prediction (the forecast of the time serie). The interval encompassing the forecast is the 90% confidence interval.

Target	Change (%)	TFT Conf. 1	TFT Conf. 2	DeepAR Conf. 1	DeepAR Conf. 2
Qgas_AC_MPFM av	[0, 10[	**210.898**	596.754	1685.583	1115.11
	[10, 30[	23.415	**22.003**	59.987	33.029
	[30, inf[	24.615	**9.327**	25.141	15.471
Qoil_AC_MPFM av	[0, 10[	3719.407	7358.845	**1832.856**	3003.175
	[10, 30[	45.379	40.519	76.179	**17.419**
	[30, inf[	40.214	39.832	33.956	**8.071**
Qwat_AC_MPFM av	[0, 10[	3669.932	3705.47	**741.588**	1708.529
	[10, 30[	63.08	46.532	**11.609**	26.5
	[30, inf[	31.72	9.686	**3.877**	6.352
Mean	[0, 10[	2533.412	3887.023	**1420.009**	1942.271
	[10, 30[	43.958	36.351	49.258	**25.649**
	[30, inf[	32.183	19.615	20.991	**9.965**

Table II.10 – MSIS obtained on the ds4 testing set by the final models of each configuration where the ones labelled 1 are the basic configurations while the ones labelled 2 only use the interest points during training (cf. Section 5.1). The results are arranged by the percentage of change in the target values of the prediction window (cf. Section 5.1).

			TFT		DeepAR	
Target	Percentile	Change (%)	Conf. 1	Conf. 2	Conf. 1	Conf. 2
Qgas_AC_MPFM av	5	[0, 10[	**0.064**	0.087	0.1	0.097
		[10, 30[	0.375	0.176	**0.1**	**0.1**
		[30, inf[	0.33	0.142	**0.1**	0.115
	50	[0, 10[	**0.272**	0.862	1.346	0.588
		[10, 30[	**0.668**	1.102	2.158	1.026
		[30, inf[	**0.552**	0.688	1.504	0.65
	95	[0, 10[	**0.082**	0.362	1.334	0.794
		[10, 30[	**0.229**	0.416	1.657	0.937
		[30, inf[	0.284	**0.24**	1.765	0.58
Qoil_AC_MPFM av	5	[0, 10[	0.123	0.236	**0.077**	0.1
		[10, 30[	0.547	0.512	1.384	**0.1**
		[30, inf[	1.233	1.113	0.758	**0.1**
	50	[0, 10[	**0.312**	0.72	1.283	0.339
		[10, 30[	**0.729**	0.783	2.988	0.811
		[30, inf[	**1.088**	1.529	2.08	1.118
	95	[0, 10[	**0.05**	0.196	0.264	0.23
		[10, 30[	0.264	**0.253**	0.54	0.32
		[30, inf[	**0.185**	0.344	0.369	0.437
Qwat_AC_MPFM av	5	[0, 10[	**0.034**	0.102	0.123	0.128
		[10, 30[	0.272	**0.113**	0.135	0.139
		[30, inf[	0.951	**0.115**	0.134	0.152
	50	[0, 10[	**0.256**	0.487	0.377	0.661
		[10, 30[	0.543	0.61	**0.533**	0.677
		[30, inf[	0.81	**0.725**	0.801	0.754
	95	[0, 10[	0.339	0.323	**0.038**	0.115
		[10, 30[	0.55	0.451	**0.117**	0.228
		[30, inf[	0.387	0.373	**0.174**	0.237
Mean	5	[0, 10[	**0.074**	0.142	0.100	0.108
		[10, 30[	0.398	0.267	0.540	**0.113**
		[30, inf[	0.838	0.457	0.331	**0.122**
	50	[0, 10[	**0.280**	0.690	1.002	0.529
		[10, 30[	**0.647**	0.832	1.893	0.838
		[30, inf[	**0.817**	0.981	1.462	0.841
	95	[0, 10[	**0.157**	0.294	0.545	0.380
		[10, 30[	**0.348**	0.373	0.771	0.495
		[30, inf[	**0.285**	0.319	0.769	0.418

Table II.11 – Q-Risk obtained on the ds4 testing set by the final models of each configuration where the ones labelled 1 are the basic configurations while the ones labelled 2 only use the interest points during training (cf. Section 5.1). The results are arranged by the percentage of change in the target values of the prediction window (cf. Section 5.1).

5.4 Evaluation based on dataset ds5

The last dataset, ds5 (cf. Section 2.3), is evaluated in this section on each DeepAR and TFT models explained in Section 5.1.

Specific experiment setup for dataset ds5

Contrary to the past two datasets, the training dataset is not restricted to the time series of the past 2 years. In fact, the events to forecast, the high peaks, happened similarly throughout the years, which is why old data can still be useful for the training. For the same reason, models don't need retraining after the optimization, like for ds4. Thus the dataset is divided the following way, considering l being the number of timesteps in ds5 and $date(i)$ the date of the i-th timestep:

- *learning set :* contains timesteps whose date is within $[date(1), date(\lfloor l/2 \rfloor)[$,
- *validation set :* contains timesteps whose date is within $[date(\lfloor l/2 \rfloor), date(\lfloor 3l/4 \rfloor)[$,
- *testing set :* contains timesteps whose date is within $[date(\lfloor 3l/4 \rfloor), date(l)[$,

The target value is the average vibration of a particular fan (vib01).

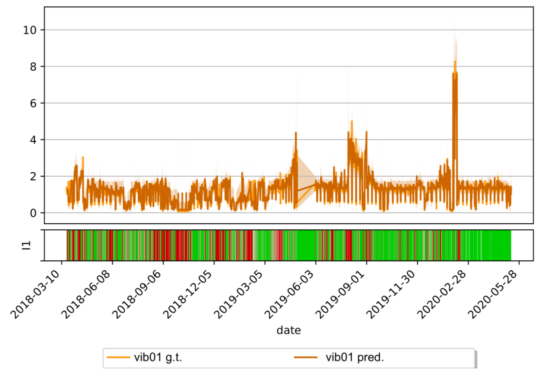
Finally, as previously the prediction length is set to 2 and the context length possible values are 2, 6 and 24.

Experiment results

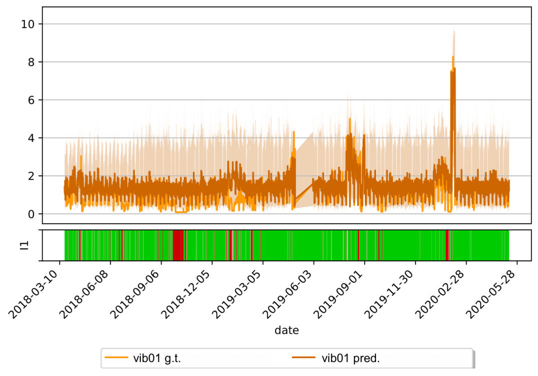
Qualitative results are shown on Figure II.24 with the one step ahead forecast on the full testing set, and Figure II.25 with a zoom on the forecast done with biggest peak of the testing set within the prediction windows. On the other hand, Tables II.12 and II.13 respectively containing the MSIS and q-Risk give quantitative scores.

The best 50-th percentile forecast is done by the first configuration of TFT since it has the lowest q-Risk on all percentage change intervals. However, even if it seems to fit the true values well on Figure II.24, the high q-Risk for [30, inf[indicates that it is not capable of predicting well changes in values. Thus it is probably close to a naive forecast method. Now, in terms of confidence interval, this same configuration has the best MSIS for [0, 10[. Nevertheless, it is still very high, due to its low coverage. For the [10, 30[and [30, inf[change intervals, it is respectively the second configuration of TFT and DeepAR which have the lowest MSIS. While the second DeepAR configuration obtains this with a naive constant large confidence interval, the TFT configuration is more interesting. As seen in Figure II.25, its confidence interval encompasses the largest peak at a value of 7 however, it never did a confidence interval as big as this one on the testing set as seen in Figure II.24. Those were mostly between 0 and 4. This means that while it struggles to detect change within 0 and 4, it is capable of predicting when the large peaks are happening.

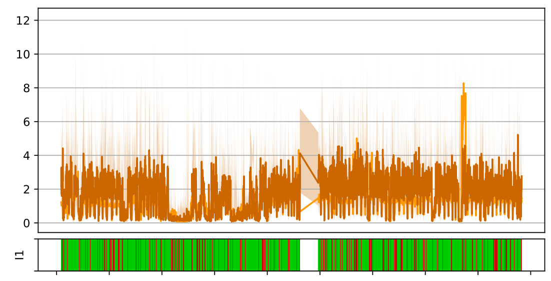
In the end, the TFT models have shown more potential than the DeepAR ones. The first DeepAR configuration had much tighter confidence interval but had problems predicting the large peak. On the contrary, the second TFT configuration had large confidence interval but was capable of predicting the large peaks. Still, they only showed potential but not accurate usable results.



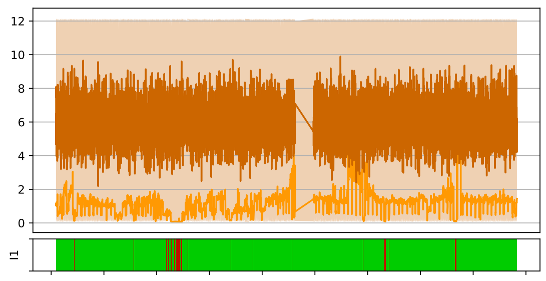
(a) TFT configuration 1: coverage= 0.640



(b) TFT configuration 2: coverage= 0.879

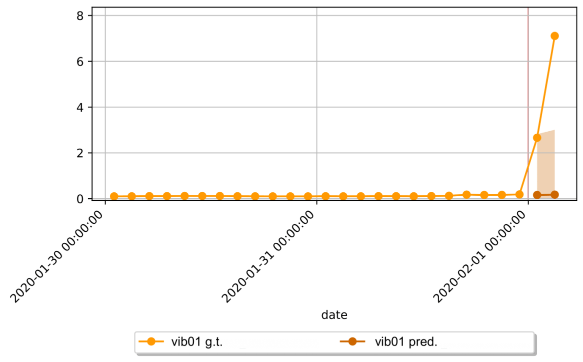


(c) DeepAR configuration 1: coverage= 0.856

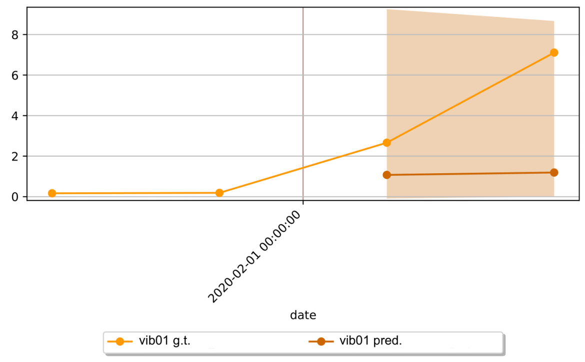


(d) DeepAR configuration 2: coverage= **0.985**

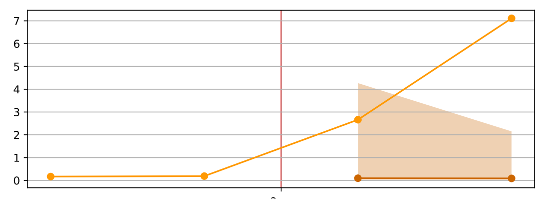
Figure II.24 – One step ahead forecasts of the ds5 target feature (vib01), stacked together, on the testing set. Green indicates when the ground truth is contained in the interval and red when it is not. The notations g.t. and pred. respectively stand for ground truth (the true values of the time series) and prediction (the forecast of the time serie). The interval encompassing the forecast is the 90% confidence interval.



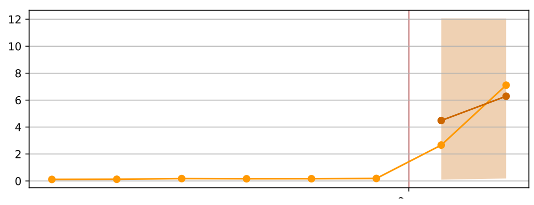
(a) TFT configuration 1



(b) TFT configuration 2



(c) DeepAR configuration 1



(d) DeepAR configuration 2

Figure II.25 – Two steps ahead forecasts of the ds5 target feature (vib01) on the biggest peak of the testing set. The points show the values of each timestep, the vertical red line indicates the end of the context windows, g.t. stands for ground truth while pred. stands for prediction and the interval surrounding the point forecasts is the 90% confidence interval.

Target	Change (%)	TFT Conf. 1	TFT Conf. 2	DeepAR Conf. 1	DeepAR Conf. 2
vib01	[0, 10[	**149.016**	2273.578	365.264	2627.063
	[10, 30[	9.855	**8.161**	17.296	17.604
	[30, inf[	18.68	8.916	12.145	**6.441**

Table II.12 – MSIS obtained on the ds5 testing set by the final models of each configuration where the ones labelled 1 are the basic configurations while the ones labelled 2 only use the interest points during training (cf. Section 5.1). The results are arranged by the percentage of change in the target values of the prediction window (cf. Section 5.1).

Target	Percentile	Change (%)	TFT Conf. 1	TFT Conf. 2	DeepAR Conf. 1	DeepAR Conf. 2
vib01	5	[0, 10[	**0.033**	0.108	0.082	0.093
		[10, 30[	0.147	0.114	**0.09**	0.094
		[30, inf[	0.173	0.179	**0.092**	0.096
	50	[0, 10[	**0.104**	0.386	0.572	3.913
		[10, 30[	**0.409**	0.455	0.608	2.31
		[30, inf[	**0.668**	0.763	0.698	1.306
	95	[0, 10[	**0.057**	0.236	0.228	0.883
		[10, 30[	**0.145**	0.162	0.318	0.542
		[30, inf[	0.62	**0.303**	0.495	0.343

Table II.13 – Q-Risk obtained on the ds5 testing set by the final models of each configuration where the ones labelled 1 are the basic configurations while the ones labelled 2 only use the interest points during training (cf. Section 5.1). The results are arranged by the percentage of change in the target values of the prediction window (cf. Section 5.1).

6 Conclusion

In this chapter, multistep multivariate forecasting by machine learning was applied to time series of production processes. The end goal was to be able to predict particular events before they occurred in order to prevent them or to be able to prepare for them. Those events are identifiable in the time series by particular patterns such as sudden high variations in values. As for the first task in chapter 1, it is expected to encounter a wide range of datasets, each with different time series and events of interest. The model should be able to be adapted to each dataset and give accurate forecasts.

To solve this task, deep learning models were considered; more precisely, DeepAR [Salinas et al., 2017] and Temporal Fusion Transformers (TFT) [Lim et al., 2019]. Those two were chosen as they demonstrate deep learning state-of-the-art performance. TFT is a model which simultaneously learns multiple quantiles which can then be used to form confidence intervals. It is based on a multi-head attention layer to process long-term relationships, Long Short-Term Memory (LSTM) for local processing, variable selection network to remove redundant and irrelevant features and gating mechanisms to skip unnecessary parts of the model. Its capability to adapt itself with the gating mechanisms is particularly interesting given the fact that it could be applied to many different datasets in this task. On the other hand, DeepAR is an auto-regressive model based on recurrent neural networks with LSTM or Gated Recurrent Unit (GRU) cells. Contrary to TFT, it learns the parameters from a given probability distribution for each forecast horizon. Then, one can sample from those probability distributions to forecast each horizon, or can compute confidence intervals by sampling multiple times. It is faster and less complex than TFT.

The implementation of DeepAR used is the one from the framework GluonTS [Alexandrov et al., 2019], while the code for TFT was retrieved in their GitHub account[6] and slightly modified to handle multivariate.

The model evaluation was proceeded on 3 different datasets. Each has had their outliers removed and missing data replaced. They were also normalized and/or standardized as a pre- and post-processing to increase the models accuracy. Their performances were evaluated based on two metrics: the q-Risk and the Mean Scaled Interval Score (MSIS). The prediction length was initially fixed to 2, and planned to be increased if the results were good. The models first went through a hyperparameter optimization based on an evaluation on a validation set before being tested on a testing set.

The DeepAR and TFT models tested on the first and second datasets showed a poor performance. While TFT had smaller confidence intervals, it was incapable of predicting the change in values while DeepAR which had the same problem made large confidence intervals encompassing all possible values. For the third dataset, DeepAR followed the same pattern with its large intervals, however, TFT demonstrated potential in forecasting sudden high peaks. Nevertheless the TFT forecasts were still not accurate enough to be usable.

The DeepAR models never managed to reduce the width of their confidence intervals during the evalution. It might be caused by the normalization which made use of non linear functions. Those functions have caused a non linear rescaling which can negatively affects the loss function. Indeed, the optimization of the parameters to minimize the loss function in the new data scale probably does not minimize the loss function in the original scale.

In conclusion, the low accuracy of the models indicates that it might be difficult to implement one model capable of handling any kind of dataset for this task. Instead, it might be better to take one dataset at a time and try to find a model corresponding to its needs. Moreover, by taking more time to adapt the model to the dataset it will now be possible to do feature processing. It was not experimented with in this chapter as it could not be easily done in a way applicable to all datasets. However, it is likely that an individual feature tuning would increase the performance of the models.

[6]GitHub URL: `https://github.com/google-research/google-research/tree/master/tft`

Chapter III

Study conclusion

In this work, two tasks were tackled: time series segmentation and multi-step multivariate time series forcasting of data from production processes. Together, they make a powerful tool capable of analysing the past and predicting the future. More than to obtain a perfect solution for the tasks, the goal was to do a proof of concept for both.

The first task was solved with tree-based models. The test result was uneven with two classes well classified and three badly classified. However, those last three were also difficult to differentiate by the engineer making the annotations. Moreover, the evaluation was performed on a single dataset. Altogether, while it is difficult to make a strong affirmation about the performance in this situation, it is easy to claim that this solution showed high potential. On top of this, the high interpretability of tree-based method opens the door for a hybrid method at middle distance between the previous WideTech segmentation method and the tree-based models.

The second task didn't go as well as the first one. It was done by using deep learning methods, however, the test done on three datasets showed two results lacking accuracy and one performance with a slight potential. It demonstrates that having a single model doing automated forecasts with little to no human input during training is difficult for these datasets. Instead, putting more manual time in each dataset to do some personalized feature processing might improve the accuracy of the models.

In conclusion, applied to data from production processes, a proof of concept was made for using tree-based models in the task of time series segmentation and tests demonstrated the difficulty of using a fully automated solution for time series forecasting of a wide range of time series.

Bibliography

Alexandrov, A., Benidis, K., Bohlke-Schneider, M., Flunkert, V., Gasthaus, J., Januschowski, T., Maddix, D. C., Rangapuram, S., Salinas, D., Schulz, J., Stella, L., Caner Türkmen, A., and Wang, Y. (2019). GluonTS: Probabilistic Time Series Models in Python. *arXiv e-prints*, page arXiv:1906.05264.

Azur, M., Stuart, E., Frangakis, C., and Leaf, P. (2011). Multiple imputation by chained equations: What is it and how does it work? *International Journal of Methods in Psychiatric Research*, 20(1):40–49.

Cai, D., Zhang, C., and He, X. (2010). Unsupervised feature selection for multi-cluster data. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, page 333–342, New York, NY, USA. Association for Computing Machinery.

Chen, T. (2017). Xgboost: A scalable tree boosting system.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv e-prints*, page arXiv:1406.1078.

Dauphin, Y. N., Fan, A., Auli, M., and Grangier, D. (2016). Language Modeling with Gated Convolutional Networks. *arXiv e-prints*, page arXiv:1612.08083.

DeepAI (2020). Epoch. https://deepai.org/machine-learning-glossary-and-terms/epoch. Accessed: 2020-06-03.

Eurostat (2014). Glossary:forecast horizon. https://ec.europa.eu/eurostat/statistics-explained/index.php/Glossary:Forecast_horizon. Accessed: 2020-06-03.

Friedman, J., Hastie, T., and Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *Ann. Statist.*, 28(2):337–407.

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Ann. Statist.*, 29(5):1189–1232.

Galassi, A., Lippi, M., and Torroni, P. (2019). Attention, please! A Critical Review of Neural Attention Models in Natural Language Processing. *arXiv e-prints*, page arXiv:1902.02181.

Hastie, T. J. (2016). *The elements of statistical learning : data mining, inference and prediction.* Springer series in statistics. Springer-Verlag, New York, 2nd ed. edition.

He, X., Cai, D., and Niyogi, P. (2005). Laplacian score for feature selection. In *Proceedings of the 18th International Conference on Neural Information Processing Systems*, NIPS'05, page 507–514, Cambridge, MA, USA. MIT Press.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.

Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980.

Krueger, D., Maharaj, T., Kramár, J., Pezeshki, M., Ballas, N., Ke, N. R., Goyal, A., Bengio, Y., Courville, A., and Pal, C. (2016). Zoneout: Regularizing RNNs by Randomly Preserving Hidden Activations. *arXiv e-prints*, page arXiv:1606.01305.

LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (2012). *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg.

Lei Ba, J., Kiros, J. R., and Hinton, G. E. (2016). Layer Normalization. *arXiv e-prints*, page arXiv:1607.06450.

Li, S., Jin, X., Xuan, Y., Zhou, X., Chen, W., Wang, Y.-X., and Yan, X. (2019). Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting. *arXiv e-prints*, page arXiv:1907.00235.

Li, Z. and Tang, J. (2015). Unsupervised feature selection via nonnegative spectral analysis and redundancy control. *IEEE Transactions on Image Processing*, 24(12):5343–5355.

Lim, B., Arik, S. O., Loeff, N., and Pfister, T. (2019). Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting. *arXiv e-prints*, page arXiv:1912.09363.

Louppe, G. (2020). Deep learning, lecture 5: Training neural networks. University of Liege, University lecture.

Mack, C., Su, Z., and Westreich, D. (2018). Managing missing data in patient registries: Addendum to registries for evaluating patient outcomes: A user's guide, third edition. Chapter : Types of Missing Data.

Makridakis, S., Spiliotis, E., and Assimakopoulos, V. (2020). The m4 competition: 100,000 time series and 61 forecasting methods. *International Journal of Forecasting*, 36(1):54 – 74. M4 Competition.

Salinas, D., Flunkert, V., and Gasthaus, J. (2017). DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks. *arXiv e-prints*, page arXiv:1704.04110.

Tixier, A. (2018). Notes on deep learning for nlp.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention Is All You Need. *arXiv e-prints*, page arXiv:1706.03762.

Wen, R., Torkkola, K., Narayanaswamy, B., and Madeka, D. (2017). A Multi-Horizon Quantile Recurrent Forecaster. *arXiv e-prints*, page arXiv:1711.11053.

Zwillinger, D. and Kokoska, S. (2000). Crc standard probability and statistics tables and formulae.