Master's Degree in Mechatronic Engineering

Politecnico di Torino

Master's Degree Thesis

UNSUPERVISED MACHINE LEARNING ALGORITHMS FOR EDGE NOVELTY DETECTION

Supervisors

Prof. Marcello CHIABERGE

Dott. Umberto ALBERTIN

Dott. Gianluca DARA

Candidate

Ariel PRIARONE

03 2024

Acknowledgements

ACKNOWLEDGMENTS

"HI"
Goofy, Google by Google

Abstract

Nowadays, the fourth industrial revolution is taking place, and it is characterized by the integration of Artificial Intelligence and the Internet of Things paradigm into factories. At the same time, in many industrial applications, the maintenance approach remained unchanged for decades. The main reason that holds back the evolution of the maintenance approach is the high costs of implementation of Condition-Based or Preventive maintenance strategies and a lack of knowledge about the modelling or behaviour of a failing system.

A framework that can be used for detecting a novelty behaviour (ND), a known fault (FD), and estimating the Remaining Useful Life (RUL) of the maintained system is proposed. To overcome the lack of models an Unsupervised Machine Learning (UML) approach is used and, on the other hand, to overcome the difficulties of implementation on different systems a modular and general-purpose framework has been developed.

A quite new frontier in the field of Predictive Maintenance is the implementation of the Artificial Intelligence application directly in the maintained system, this approach is called Edge Computing.

In this work, the framework has been developed to be run and tested on a PC using various Unsupervised Machine Learning algorithms, Using the Python language. The algorithms that appeared to maximize the performance-resources ratio have been then implemented in a microcontroller, using the C language. The edge implementation has been tested with laboratory experiments.

The proposed solution relies on the application of Machine Learning to features extracted by the time-domain signal by the framework itself. With this structure, the framework can be easily set up on a machine, read signals from sensors, extract features, and then train the models. The framework then switches to evaluation mode to perform ND, FD, and RUL predictions. After a Nd or FD event, the model can be refined performing a retraining using only the data that generated the event.

The considered features are settable for each sensor. The implementation of the feature extraction manages time-domain features (Mean, RMS, Standard Deviation, P2P, Skewness, Kurtosis) and frequency-domain features (The power of coefficients

of a Wavelet Packet Decomposition of any depth, or the FFT of the signal). The framework is designed to be easily extended to additional features and sensors.

The PC implementation is based on Software Agents that act autonomously on a common database. The following algorithms have been implemented to perform ND, FD, and RUL predictions: K-means, DBSCAN, Gaussian Mixture Models, One-Class Support Vector Machines, Isolation Forest and Local Outlier Factor. The K-means model was then chosen for the Edge implementation.

All the cited algorithms' performances have been compared on a dataset of bearings vibration published online by the Center for Intelligent Maintenance Systems (IMS) of the University of Cincinnati. The Edge implementation of the framework has been tested in laboratory experiments, using an accelerometer to measure the vibrations generated by an active shaker and, on a second set of tests, by a linear actuator.

The last refinement of the framework has been the implementation of additional feature scaling after the standardization, using a Random Forest model to refine the UML model. This mitigates the effect that noisy features have on the performance of the algorithms. This approach has been tested on the laboratory data.

Table of Contents

List of Tables

List of Figures

Chapter 1

Proposed Framework

In the **??** the features extraction process has been described. Before approaching the problem of performing ND in edge computing, let's build a framework in `python` that runs on a PC that is *configurable*, *modular*, *expandable* and *general purpose*. This framework will be used to test the features extraction process and to test the ML algorithms before selecting one of them to be implemented in edge computing framework, which will be harder to configure.

A real case application would probably have several signals of several physical quantities, so a general approach that can manage different types of features, and extract from each of them the most relevant information, is needed.

The proposed framework is thought to be set up on any type and combination of sensors. The framework is thought to manage data that are correlated to a specific fault. For example, think about a CNC machine like the one in **figure 1.1**. It has five axes, so a solution would be to instance the framework five times, one for each axis, linked to vibration sensors, temperature sensors etc. of the considered axis. This would allow us to pinpoint the fault to a specific axis. Another concern is, what if the single axis is seeing a normal condition, but the machine as a whole is not? This may happen if the tool has a problem: the vibration registered in the spindle would be normal in general, but would not be normal *related* to the feed rate that another axis is imposing. To address this scenario, other instances of the framework can be set up, that also receive the speeds from the machine controller and the feed rate from the CNC program as well as data from the sensors used in the other instances. This would allow us to detect a more complex fault, that is not characteristic of any part of the machine, but of the machine as a whole. The former kinds of instances would allow specific faults to be detected, giving also an idea of what the fault is, the latter would allow complex faults to be detected, but would not give a precise idea of what the fault is. The framework is thought to be able to manage both of these scenarios, and to be able to manage them together.

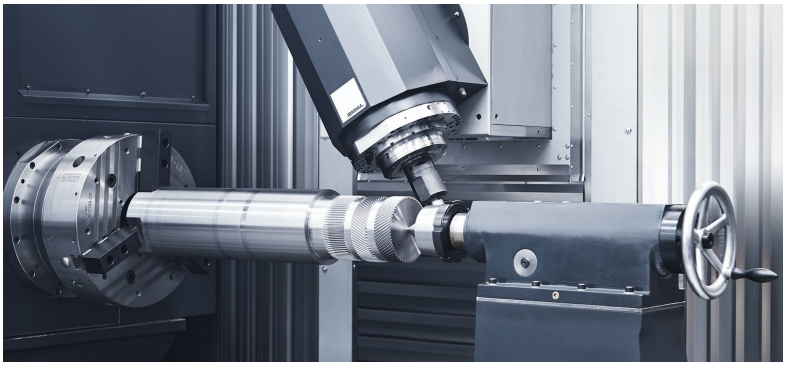

Figure 1.1: *A 5-axis CNC milling machine. [1]*

1.1 Commissioning

To adapt the framework to a specific machine, the commissioning of the ML system would have to be done in steps. Starting from the data acquisition and ending with the predictions of RUL and model updates, the steps are described in this section.

1.1.1 Data structure

The first phase of adaptation of the framework to a machine is to define what data to sample and how to sample them. This includes the decision of which sensors to use, the sampling frequencies, the data acquisition system and which features are needed to be extracted from each sensor data. At this point, if more than one instance of the framework is needed, the sets of sensors and features to be used in each instance are defined. For example, in a shaft with two bearings, each with two accelerometers, the first instance of the framework would be linked to the first bearing, and would use the data from the two accelerometers to extract the features that are needed to detect the fault in the first bearing. The second instance of the framework would be linked to the second bearing, and would use the data from the other two accelerometers to extract the features that are needed to detect the fault in the second bearing. Optionally, a third instance of the framework would use the data from all four accelerometers to detect a generic fault in the shaft. Those decisions influence the structure of the database, which will be described in **section 1.2**.

1.1.2 Data acquisition

Once the structure of the data is defined, the first phase of the commissioning procedure is to set up the data acquisition. This has to be done when the machine is new or, at least, someone guarantees that the machine is in a healthy condition.

During the previous phase, the number of instances of the framework is defined. Each instance would have its own database. This phase is just a matter of storing the data that will be used to train the models the first time. A software agent, which we call Field Agent (FiA), is responsible for this task. This phase lasts until the database is filled with enough data to train the models.

1.1.3 Training

The second phase of the commissioning procedure is to train the models. Once the healthy data are enough to characterize all the normal conditions of the maintained system, all the recorded data are elaborated by another software agent that we call Feature Agent (FA). This agent extracts all the features from the time-series and stores them in a structured way.

Once all the features are available in the database, another agent called Machine Learning Agent (MLA) is responsible for training the models. All the models considered are UML models. The models are trained on a standardized version of the feature matrix. The standardization is done w.r.t. the time evolution, i.e. all the features used for training have a time evolution with zero mean and unit variance. This is done because most ML algorithms are sensitive to the scale of the features.

All that has been said is valid for a single instance of the framework. If more than one instance is needed, the training phase has to be done for each instance.

1.1.4 Evaluation

At this point, a model that represents the normal condition of the system is available (actually a model for each instance of the framework). The next step is to evaluate the model.

In this phase, the machine continues to perform its normal operations. The Field Agent provides the sampled data, the Feature Agent extracts the features and the Machine Learning Agent evaluates the health of the system. The proposed novelty/fault metric and procedure are specific to the model used, as described in the dedicated chapter about UML models (**??** for the k-means, **??** for DBSCAN, **??** for GMM, **??** for ν-SVM, **??** and **??** for LOF).

Now the ND is up and running. The novelty metric is used to decide if the system is healthy or not. The metric is plotted for the user to see. Note that in a

classic ML approach, the dataset is split into a training set and a test set. In this case, the test set is the data that are sampled during the evaluation phase. It's equivalent to saying that the model is trained on the past data and evaluated on the future data, or that the framework works in testing phase for an undetermined amount of time, until the user decides to update the models. This is equivalent to a test phase because if during this phase the framework outputs too many false positives, the user will decide to update the models. Otherwise, it means that the models are working properly and this phase can last indefinitely.

1.1.5 Model update

Once the metric overshoots the threshold, the MLA warns the user about the novelty detected, and starts to perform PdM predicting the future evolution of the metric and the RUL of the system. Again, this condition can last indefinitely, once new data are sampled, the MLA evaluates the health of the system and updates the predictions.

It's up to the user to determine if the warning is a false positive, a novel but healthy behavior, or a real fault. In the first two cases, the user can decide to command the MLA to update the models. The snapshots that generated the warning are incorporated into the training set, and the models are retrained, returning to the evaluation phase. In the latter case, the user can simply perform the repairs/maintenance needed and restore the system to a healthy condition or can use the snapshots that generated the warning to train a new model that represents the fault condition.

If the user decides to train also the second model with the snapshot declared faulty, the system returns into the evaluation phase, but now the MLA outputs two metrics: one estimates the health of the system and the other how similar the behavior of the system is compared to any known fault condition.

1.1.6 Predictions

The metric generated by the MLA is useful to detect novelties, with a CBM approach. To actually perform PdM, the MLA has also to predict the future evolution of the metric. Since, as anticipated in the introduction, this framework aims to output *degradation based* predictions, a suitable fitting curve has to be used. Degradation-based failures are led by an initial defect that worsens over time. Often, the presence of an early defect further increases the worsening rate of the system. For this reason, and by observing the features evolution on publically available datasets, it seems reasonable to model the degradation with an exponential curve. This approach has been used in [2] and [3].

The candidate function to fit would then be:

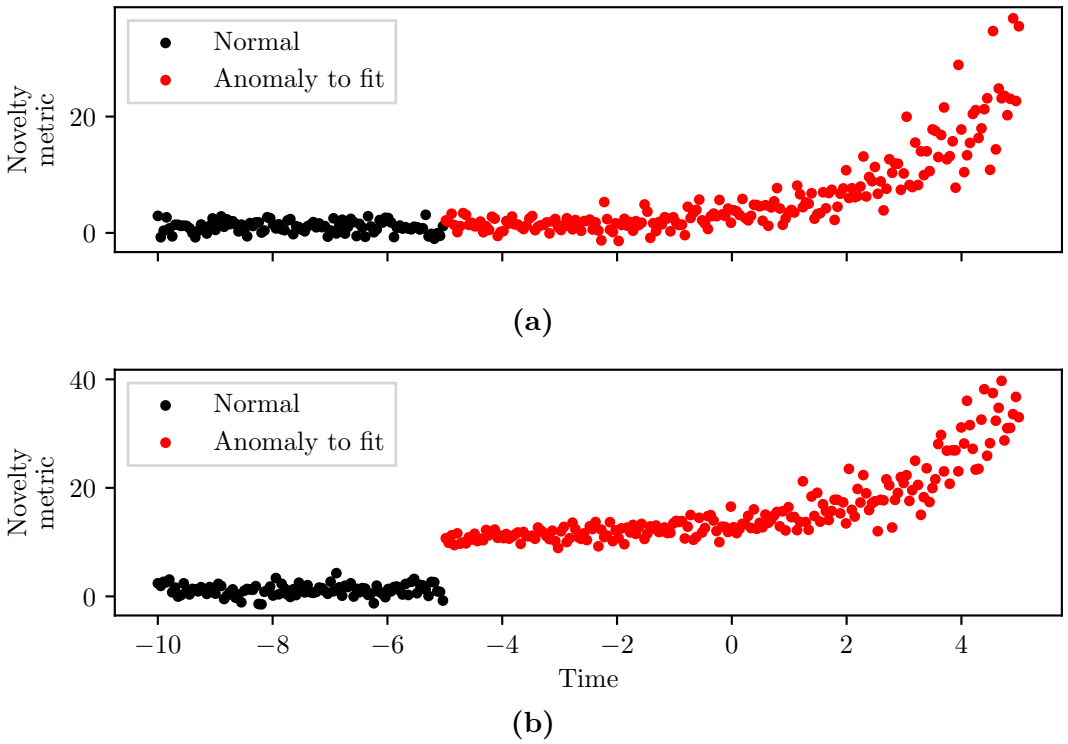

Figure 1.2: *Novelty metric data to fit with an exponential curve.*

$$f(t) = a \cdot e^{b \cdot t} \tag{1.1}$$

where a and b are the parameters to be estimated. This may work in the case depicted in **figure 1.2a**, in which the novelty metric starts from zero and then increases exponentially. This is a problem for a general case in which the metric can have a plateau, or start as a step that signals the early defect, and then start an exponentially decay, as the example in **figure 1.2b**. In our case, in **??**, the set of defined metrics to be used are usually negative, to depict a normal behavior, and not zero.

The **equation 1.1** is nice, because by extracting the logarithm from both sides (and calling $y = \log(f(t))$) we obtain a linear equation that can be fitted easily with the least squares method described in **??**, but for the reasons explained above, it's not suitable for our case. A better candidate is:

$$f(t) = a \cdot e^{b \cdot t} + c \tag{1.2}$$

This is a better candidate because it can depict the case in which the metric starts from a value different from zero, and then increases exponentially.

`Scipy` fit

Unfortunately, **equation 1.2** cannot be aranged in linear form, so the least squares solution has to be found with some other procedure. The `python` library `scipy.optimize` has a function called `curve_fit` that can be used to fit a generic function to a dataset. The problem with this recursive solution is that is very sensitive to outliers and can't really estimate the parameter c correctly, as is shown in **figure 1.3**.

Closed form fit with least squares

Fortunately, in [4], the author provides a *nonrecursive* solution to the problem that minimizes the LS error w.r.t. the parameters a, b and c. The solution has been converted into the following **algorithm 1** and implemented in the framework. The fitting with this closed-form solution is shown in **figure 1.3**. This optimal solution is used as default in the framework to predict the future evolution of the metric, the `scipy` implementation can still be used by changing a parameter in the configuration file because it has the advantage of fitting any function, so it may be useful in particular cases.

Algorithm 1 Exponential regression of the novelty metric.

function ExpRegression(x, y)
  ▷ $x = \{x_1, \dots, x_n\}$ is the array of x-coordinates of the data points.
  ▷ $y = \{y_1, \dots, y_n\}$ is the array of y-coordinates of the data points.
  ▷ Returns the parameters a, b and c of the exponential function $y(x) = a \cdot e^{b \cdot x} + c$ that minimize the least squares error.
  $S_1 \leftarrow 0$
  $S_k \leftarrow S_{k-1} + \frac{1}{2}(y_k + y_{k-1})(x_k - x_{k-1}) \quad \text{for} \quad k \in [2, n] \cap \mathbb{N}$
  $$\begin{bmatrix} A_1 \\ B_1 \end{bmatrix} \leftarrow \begin{bmatrix} \Sigma(x_k - x_1)^2 & \Sigma(x_k - x_1) \cdot S_k \\ \Sigma(x_k - x_1) \cdot S_k & \Sigma S_k^2 \end{bmatrix}^{-1} \begin{bmatrix} \Sigma(y_k - y_1)(x_k - x_1) \\ \Sigma(y_k - y_1) \cdot S_k \end{bmatrix}$$
  $a_1 \leftarrow -\frac{A_1}{B_1}$
  $c_1 \leftarrow B_1$
  $c_2 \leftarrow c_1$
  $\theta_k \leftarrow e^{c_2 \cdot x_k} \quad \forall k \in [1, n] \cap \mathbb{N}$
  $$\begin{bmatrix} a_2 \\ b_2 \end{bmatrix} \leftarrow \begin{bmatrix} n & \Sigma\theta_k \\ \Sigma\theta_k & \Sigma\theta_k^2 \end{bmatrix}^{-1} \begin{bmatrix} \Sigma y_k \\ \Sigma y_k \cdot \theta_k \end{bmatrix}$$
  $a \leftarrow b_2$
  $b \leftarrow c_2$
  $c \leftarrow a_2$
  return a, b, c
end function

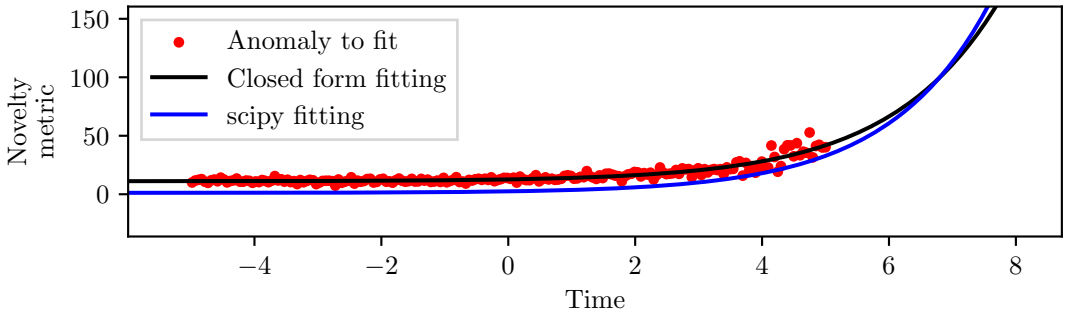

Figure 1.3: *Fitted curve for RUL prediction. The* `scipy` *library fit fails to estimate the parameter c. The closed-form solution actually minimizes the error.*

1.1.7 Instance structure

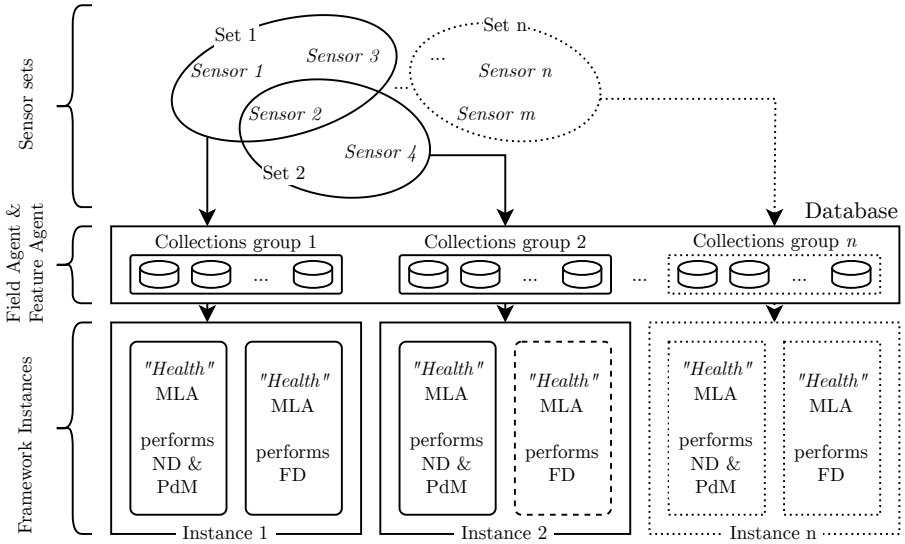

Figure 1.4: *The structure of the instances of the framework.*

As anticipated, in order to address different kinds of malfunctions, the framework is thought to be instanced several times on the same system. To visualize the structure of the instances, consider the general case described in **figure 1.4**. Any instance can rely on data gathered from any sensor subset and can perform ND and FD. In the figure, the first instance reads the first three sensors and has both the ND and FD algorithms active (this means that a fault dataset has been gathered in the past). The second instance reads a shared sensor with the first one, plus two new sensors. The dashed line surrounding the FD algorithm means that is not

active (no fault dataset has been gathered in the past, or it has not been trained). This instance performs only ND. The last instance is just a general prosecution of the structure, that can be scaled indefinitely.

1.2 Database

In the previous **section 1.1**, the setup behavior phases of the framework have been described, referring to a generic "database", without specifying the structure of the database. Let's now address the problem of storing the data efficiently and effectively. Instead of relying on `python` data structures, it is better to use a dedicated database manager.

The proposed framework uses MongoDB for the following reasons. It is a widely-used, open-source NoSQL database that is designed to handle unstructured or semi-structured data. It utilizes a document-oriented data model, storing data in flexible, JSON-like BSON format. MongoDB is suitable for implementation in a ND framework due to its scalability, flexibility, and real-time data processing capabilities. In novelty detection, the system often deals with diverse and dynamic data sources, making MongoDB's "unstructureness" advantageous for handling varying data formats and evolving data requirements. It has the ability to handle large volumes of data and support scaling allowing for efficient storage and retrieval of information in real-time, crucial for real-time applications. Moreover, MongoDB has a rich query language and secondary indexes that allow for fast and efficient querying of data and a library for `python` that makes it easy to use. The JSON format is also human-readable, which makes it easy to understand the data stored in the database, and "mongoDB Compass" is a graphical user interface that allows one to easily explore the database.

1.2.1 Collections

Table 1.1: Collections contained in the MongoDB database

Collection	**Content**
raw	time-series and information about them
unconsumed	snapshots to be evaluated
quarantined	snapshots detected as novelty waiting to be declared healthy, faulty or be discarded
healthy	snapshots declared as normal behavior

Continued on next page

Table 1.1: Collections contained in the MongoDB database (Continued)

healthy train	training dataset (scaled, processed, packet) for the ND UML model
faulty	snapshots declared as faulty behavior
faulty train	training dataset (scaled, processed, packet) for the ND UML model
models	models trained on healthy and faulty data the metrics and predictions to be shown
backup	time-series, features, models, etc.

MongoDB structure is based on collections, that are groups of (JSON) documents. A document is a set of key-value pairs that can be nested in several layers. Documents have a dynamic schema, which means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data. To store the data needed by the framework the collections reported in **table 1.1** are used. In the following paragraphs, the structure and purposes of each collection are described.

Raw Thinking about the data flow, the first interface between the hardware and the software would be the sensor readings. Every sensor should have a name and be sampled at a constant frequency (or, at least, the sensors that provide data for frequency-domain feature extraction should have a constant sampling frequency). This data is stored in the raw collection, with the JSON structure summarized in **??**.

Table 1.2: *Structure of the "raw" collection JSON configuration file.*

Field	**Sub-Field**	**Type**
_id		string
timestamp		ISO date
sensor 1	sampling frequency	float
	time-serie	list[float]
sensor 2	sampling frequency	float
	time-serie	list[float]
...	...	...
sensor n	sampling fequency	float
	time-serie	list[float]

where `_id` is the unique identifier of the document, `timestamp` is the time at which the data was acquired, in ISO format, and `Sensor 1` to `Sensor n` are the

names of the sensors. Each sensor has a `sampFreq` field that contains the sampling frequency of that particular sensor, and a `timeSerie` field that contains the data acquired by the sensor, as a list. The `timeSerie` field is a list of floating point numbers, that can be of any length. Note that the sampling frequencies of different sensors can be different, for example, if a timestamp contains 1s period of data, a vibration sensor would be linked to an array with several thousands of samples, while a temperature sensor would be linked to only one sample.

Unconsumed Once defined the structure that the time-series will have in the database, let's define the structure of the snapshots. The features extracted from the time-series are stored in the unconsumed collection, with the JSON structure described in **??**.

Table 1.3: *Structure of the "unconsumed" collection JSON configuration file.*

Field	**Sub-Field**	**Type**
_id	-	string
timestamp	-	ISO date
sensor 1	mean	float
	root mean square	float
	peak to peak	float
	standard deviation	float
	skewness	float
	kurtosis	float
	wavelet coefficient 1	float
	wavelet coefficient 2	float
	⋮	⋮
	wavelet coefficient $2^{\text{three dept}}$	float
sensor 2	mean	float
	root mean square	float
	peak to peak	float
	standard deviation	float
	skewness	float
	kurtosis	float
	wavelet coefficient 1	float
	wavelet coefficient 2	float
	⋮	⋮
	wavelet coefficient $2^{\text{three dept}}$	float
⋮	⋮	⋮
sensor n	mean	float

	root mean square		float
	⋮		⋮
	wavelet coefficient	$2^{\text{three dept}}$	float
novelty evaluated flag	-		boolean

Notice that different sensors can have different features. The "novelty evaluated" field is a boolean that is set to `false` when the snapshot is created, and is set to `true` when the ND algorithm evaluates the snapshot. This field is used to avoid evaluating the same snapshot multiple times while leaving it in the collection until also the FD algorithm is performed. At this point, the snapshot will be moved either to the backup collection, discarded or to the quarantine collection if either the ND or the FD flag it.

Quarantined The "quarantined" collection is used to store the snapshots that were flagged as "novelty" by the ND algorithm or as "faulty" by the FD algorithm (or were flagged by both of them). The structure is the same as the "unconsumed" collection, but the "novelty evaluated" field is not present since, at this point, the snapshots are guaranteed to have been evaluated. The snapshots in this collection are waiting to be declared as "healthy" or "faulty" by the user or to be discarded.

Healthy The idea behind the "healthy" collection is to store the snapshots that are acquired during the first work phase of the framework, before training, or the snapshots that were in the "quarantine" collection and were declared as healthy by the user. The documents in this collection have the same structure as the documents in the "quarantined" collection.

Healthy train In this collection the healthy snapshots are packed together in different documents, each of them useful in a different phase of the training process.

The first document hase the `id training_set`, that contains all the `N` training snapshots, each of them with `n` sensors signals, characterized by `F` features. For ease of accessibility, every bottom-nested field is a list of `N` elements. The structure is resumed in **??**.

Table 1.4: *Structure of the "healthy train" collection JSON configuration file.*

Field	Sub-Field	Type
_id	-	string
timestamp	-	list[ISO date]
sensor 1	feature 1	list[float]
	feature 2	list[float]

	⋮	⋮
	feature F	list[float]
sensor 2	feature 1	list[float]
	feature 2	list[float]
	⋮	⋮
	feature F	llist[loat]
⋮	⋮	⋮
sensor n	feature 1	list[float]
	feature 2	llist[loat]
	⋮	⋮
	feature F	list[float]

This collection contains other three documents:

- `training set scaled`, that contains the scaled training set, having the same structure as the `training set` document;
- `training set MIN MAX`, that contains the minimum and maximum values of the features of the training set, useful to plot the features with a reference of the bounds of the training set. It has the same structure of the `training set` document, but the bottom-nested fields are lists of two elements (the minimum and the maximum value);
- `StandardScaler_pickled`. It contains the `StandardScaler` object that was used to scale the training set. This object is encoded in Pickle format, and it is used during the evaluation phase to scale the snapshots before evaluating them.

Faulty This collection serves the same exact purpose as the "healthy" collection, but for the faulty snapshots. Faulty snapshots are not discarded because they can be used to train the FD UML algorithm.

Faulty train This collection serves the same exact purpose as the "healthy train" collection, but for the faulty snapshots.

Models This collection contains the models trained on the healthy and faulty data and a buffer of the predictions and metrics to be displayed to the user.

The structure of the models documents is just an identifier and the `python` object of the model, encoded in Pickle format. The structure of the predictions and metrics documents is the **??**:

Table 1.5: *Structure of the "models" collection JSON configuration file.*

Field	Sub-Field	Type
_id	-	string
timestamp	-	list[ISO date]
values	-	list[float]
assigned cluster	-	list[int]
anomaly flag	-	list[bool]
prediction curve parameters	-	pickle format

Backup The backup collection is a general-purpose container for any document that needs to be stored for backup purposes. It can contain time-series, features, models, etc. The structure of the documents in this collection is the same as the structure of the documents in the other collections.

1.3 Software Agents

In the previous sections, the software agents were mentioned as the main actors in the framework. This section will provide a more detailed description of the software agents, their role and their interaction with the environment and the data, following the flow from the hardware through the time-series, the feature domain to the ML algorithms. The reference layout is the one in **figure 1.5**.

Software agents are autonomous programs that perform a specific task in a cycle. In the proposed `python` implementation, the agents are classes that are instanced and run in a loop.

1.3.1 Field Agent

The Field Agent is the interface between the hardware and the software. It is responsible for the acquisition of the data from the sensors and the communication with the database. Since some features are related to the spectrum of the data, a precise and fixed sampling frequency is needed. Hence, the FiA must incorporate a synchronization with the ADC. It stores data in the *raw* collection and the *backup* collection. In **figure 1.6**, the flow of operations is shown as a flowchart, emphasizing the importance of the synchronization with the ADC. This software

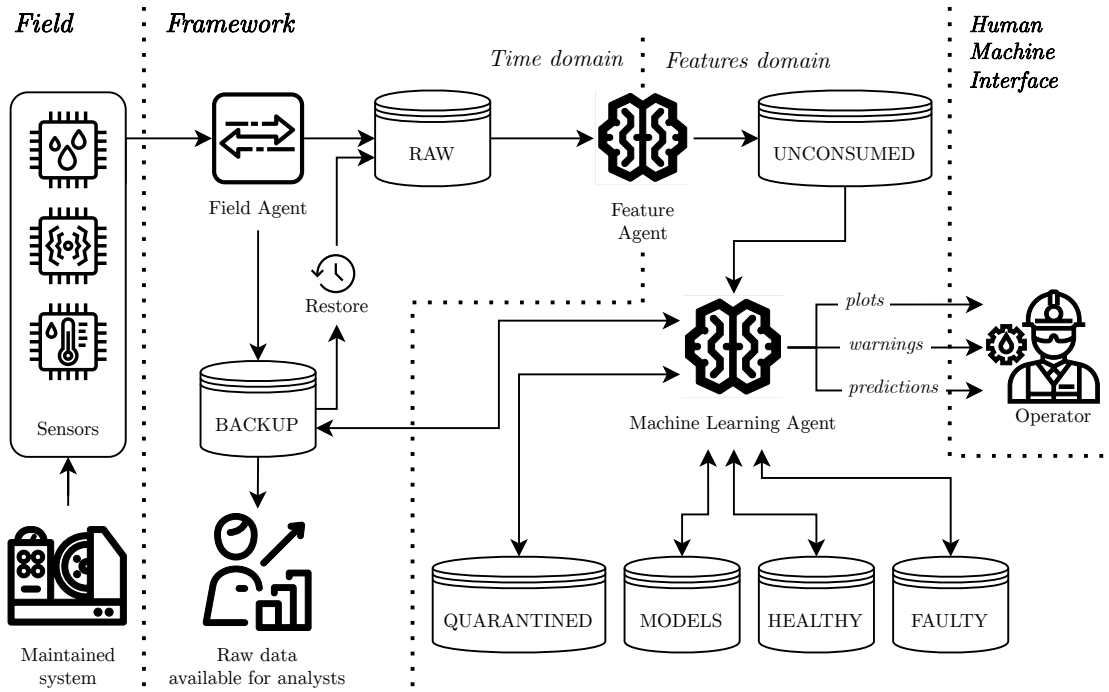

Figure 1.5: *Framework logical structure*

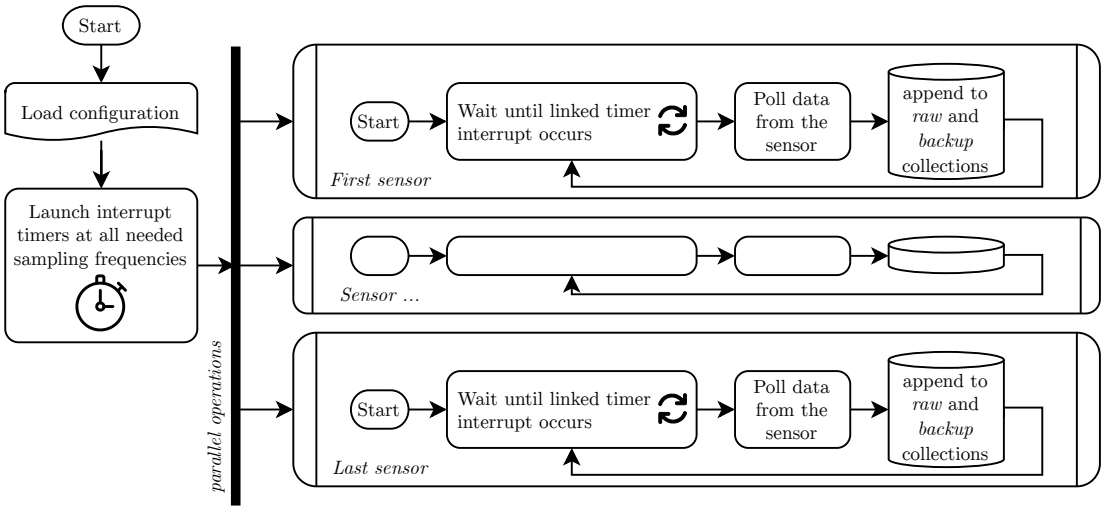

Figure 1.6: *Field Agent flowchart*

agent has not been implemented in `python`, because the experimental validation of this work, as it will be described in **chapter 2**, has been performed directly on the edge computing platform. During the tests on the publicly available datasets, an

abstract version of the FiA has been used, that reads the data from the CSV files.

1.3.2 Feature Agent (FA)

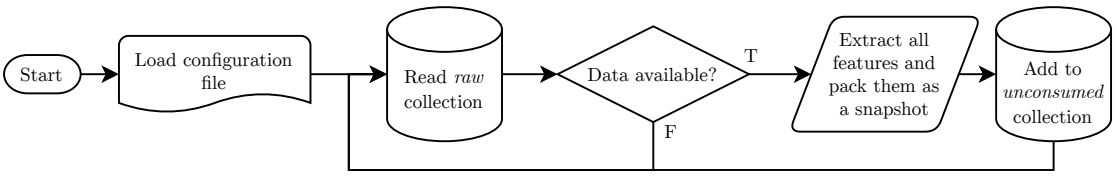

Figure 1.7: *Feature Agent flowchart*

The FA is responsible for the feature extraction from the raw data. It reads the data from the *raw* collection, extracts the features and stores them in the *unconsumed* collection. The flow of operation is shown in **figure 1.7**. The FA is implemented in `python` and it is a class that has been designed to be easily expandable and configurable. The methods implemented in the FA class are shown in **table 1.6**.

Table 1.6: *FA class implemented methods*

Method	**Description**
readFromRaw	reads a snapshot from the raw collection and stores it in the instance self
extractFeatures	extract all the features from the current snapshots, for all the sensors
extractTimeFeautures	extract mean, rms, P2P, std, skewness and kurtosis, based on the config file for the specified sensor
extractFreqFeautures	extract the wavelet coefficients for the specified sensor, up to the configured depth
deleteFromraw	delete current snap record from the *raw* collection
writeToUnconsumed	write the extracted features to the *unconsumed* collection
initialize_barPlotFeatures	initializes the bar plot of the features that is shown to the user
barPlotFeatures	updates the bar plot with new features

run	perform the agent operations in a loop. idle until new data are available in *raw* collection

1.3.3 Machine Learning Agent (MLA)

The MLA is responsible for the training and the evaluation of the ML models. It reads the data from the *unconsumed* collection, evaluates the snapshot and stores the result in the *models* collection, it also constantly updates the information about the novelty or fault metric to the user. The flow of operation is shown in **figure 1.8**. The methods implemented in the MLA class are shown in **table 1.7**.

This agent is designed to be configured as a ND or FD agent with just one hyperparameter. If it is instanced for ND, it uses the healthy collection as a training dataset, if it is instanced for FD it uses the faulty collection. The metric used to evaluate the snapshots is the novelty metric for ND and the fault metric for FD. According to the procedure defined in **??**.

Table 1.7: *MLA class implemented methods*

Method	**Description**
run	run the MLA according to its current state
evaluate	evaluate the current snapshot based on the novelty or the fault metric, according to the type of instance
predict	fits the novelty metric with a degradation curve to predict the future evolution
mark_snap_evaluated	set the evaluated flag to true for the current snapshot
delete_evaluated_snap	remove the evaluated snapshots from the *unconsumed* collection
scale_features	scales the features of the current snapshot according to the standard scaler used during the training procedure
evaluate_error	compute the novelty or fault metric for the current snapshot, according to the type of instance
calculate_train_cluster_dist	compute the radiuses of the clusters during the training procedure
prepare_train_data	performs the preprocessing of the data before training the model
pack_train_data	pack the training snapshot in a matrix

___move_to_train	move an entire collection of snapshots to the training collection
standardize_features	make all the features in the training matrix have zero mean and unitary variance
save_features_limits	save the unscaled bounds of the training features
save_StdScaler	store the standard scaler instance in Pickle format into the database
retrieve_StdScaler	restore the standard scaler instance in Pickle format from the database
save_KMeans	store the model instance in Pickle format
retrieve_KMeans	restore the model instance in Pickle format
_append_features	append the current features in a document
train	performs the training of the clustering models
evaluate_silhouette	compute the silhouette score of the training set snapshots
___plot_silhouette	plots the silhouette score against the number of clusters
evaluate_inertia	compute the inertia score of the training set snapshots
___plot_inertia	plots the inertia score against the number of clusters
packFeaturesMatrix	format all the training features as a matrix
retrain	perform a new training of the models

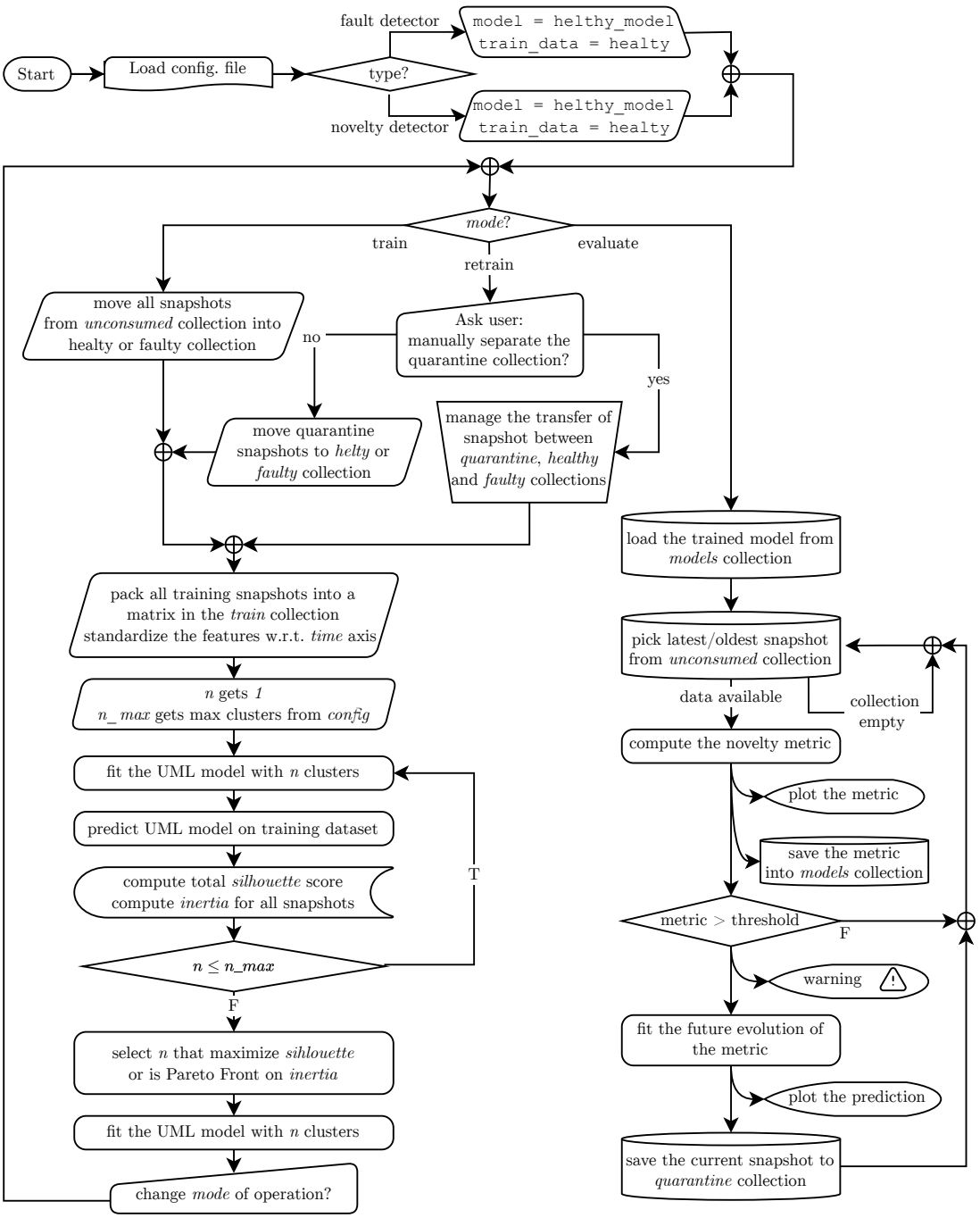

Figure 1.8: *Machine Learning Agent flowchart. When it is instanced for ND, the MLA uses the healthy collection as a training dataset, when it is instanced for FD it uses the faulty collection.*

1.3.4 Configuration of the framework

All the configurations described in this chapter are stored in the `config.yaml` file. This file is read by the agents at the beginning of their execution. The configuration file is divided into sections by topic: database, models etc. An example of a valid configuration file for the IMS dataset is shown in below.

```yaml
Database:           # Database configuration
URI:                mongodb://localhost:27017   # MongoDB URI
db:                 Shaft                       # Database name
collection:       # Collection configuration
  back:             BACKUP         # Backup collection name
  raw:              RAW            # Raw data collection name
  unconsumed:       UNCONSUMED     # Unconsumed data collection name
  healthy:          HEALTHY        # Healthy data collection name
  healthy_train:    HEALTHY_TRAIN # Healthy data collection name for
   ↪ training (some healthy data are not used if not novelty)
  quarantined:      QUARANTINED    # Quarantined data collection name
  faulty:           FAULTY         # Faulty data collection name
  faulty_train:     FAULTY_TRAIN   # Faulty data collection name for
   ↪ training (some faulty data are not used if not fault novelty)
  models:           MODELS         # Models collection name
sensors:          # Expected sensors in the database
  Bearing 1 x:    # Sensor 1 name
    features:     # Features configuration
      wavPowers:    on             # Wavelet powers
      mean:         on             # Mean value
      rms:          on             # Root mean square
      peak:         on             # Peak to peak value
      std:          on             # Standard deviation
      skew:         on             # Skewness
      kurt:         on             # Kurtosis
  Bearing 1 y:    # Sensor 2 name
    features:     # Features configuration
      wavPowers:    on             # Wavelet powers
      mean:         on             # Mean value
      rms:          on             # Root mean square
      peak:         on             # Peak to peak value
      std:          on             # Standard deviation
      skew:         on             # Skewness
      kurt:         on             # Kurtosis
  ...:
  Bearing 4 y:    # Sensor n name
    features:     # Features configuration
      wavPowers:    on             # Wavelet powers
```

```
37      mean:          on            # Mean value
38      rms:           on            # Root mean square
39      peak:          on            # Peak to peak value
40      std:           on            # Standard deviation
41      skew:          on            # Skewness
42      kurt:          on            # Kurtosis
43  wavelet:           # Wavelet packet transform configuration
44    type:              db10        # Type of wavelet to use
45    mode:              symmetric   # Mode of the packet transform
46    maxlevel:          6           # Max level -> num of features =
      ↪ 2^level
47  model:             # machine learning model configuration
48    max_clusters:      100         # Max number of clusters
49    max_iterations:    1000        # Max number of iterations
50    error_queue_size:  1           # number of predictions parameters
      ↪ to keep in queue for plotting/predicting
51    error_plot_size:   2000        # Size of the plotting queue
52  novelty:           # Novelty/fault configuration
53    threshold:         0.10        # Threshold for novelty / fault
      ↪ detection: relative percentage of the distance to the cluster
      ↪ w.r.t the max distance to the assigned cluster in the training
      ↪ set
54    forecast_threshold: 0.8        # Threshold for novelty / fault
      ↪ prediction
55    n_fit:             250         # Number of samples used to fit the
      ↪ prediction curve
56    outlier_filter:    1           # Number of consecutive outliers to
      ↪ filter (1 means two outliers will be considered as novelty/fault)
57    regressor:         exp         # Regressor to use for prediction:
      ↪ exp, scipy, poly. if exp uses a custom function, if scipy uses
      ↪ scipy.optimize.curve_fit, if poly fit polynomial of degree
58  miscellanea:
59    logpath:           C:\Users\JohnSmith\Documents\framework  # Path to
      ↪ the logs
```

1.3.5 Command Line Interface (CLI)

To ease the interaction with the user, a CLI has been implemented. It relies on the `click` and `typer` libraries for `python`. The CLI allows the user to instance the agents, to configure the framework, to monitor the agents and to interact with the database. All the commands are provided with a help message that can be accessed by typing `-help` after the command, as shown in **figure 1.9** for the command `run-machine-learning-agent`. The commands implemented in the

CLI are shown in **table 1.8**.

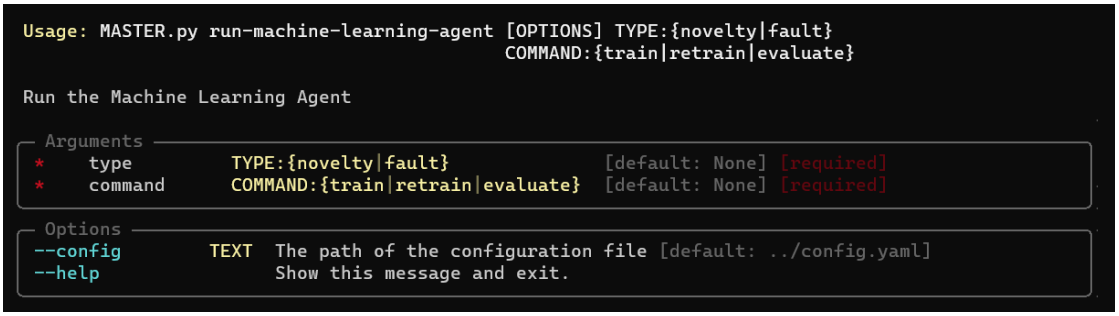

Figure 1.9: *Command Line Interface help message*

Table 1.8: *CLI implemented commands*

Command	Description
copy-collection	Move all the documents from one collection to another
create-empty-db	Create an empty database in MongoDB. The database should not exist already. It is configured according with "config.yaml" file.
ims-converter	Transfer the data from the glsims textual files into the MongoDB database in a suitable way.
fault-indicator	This function plots the fault metric.
novelty-indicator	This function plots the novelty metric.
move-collection	Move all the documents from one collection to another
plot-features	Plot the features of the last snapshot in the UNCONSUMED collection
run-feature-agent	Run the Feature Agent - takes the last snapshot from RAW collection, extracts features and writes them to UNCONSUMED collection
run-machine-learning-agent	Run the Machine Learning Agent

Chapter 2

Validation

This chapter is dedicated to the validation of the framework on real-world data. In **??**, the reference dataset [5] has been introduced. Firstly, the `python` implementation of the framework is validated on the IMS dataset several times with different configurations to show the flexibility of the framework, and try to find the best configuration for the dataset.

The first test in the IMS dataset is carried out with all the implemented machine learning models, then only the K-mean model is used in the following tests. In all tests an outlier filter has been implemented, so that the MLA will warn about the novelty behavior only if two consecutive snapshots are labeled as outliers. The number of consecutive snapshots is a parameter that can be adjusted in the framework settings.

Then, the edge computing implementation of the framework based on the K-means clustering is validated experimentally on a machine and with a laboratory shaker.

2.1 IMS dataset No.1 - Bearing 3x sensor

To start the validation, let's subdivide the test No.1 of the IMS dataset into *training* and *testing* datasets. The first 500 samples are used for training, and the remaining samples are used for testing.

For all the algorithms, the assumption about the system is that, even if the degradation is continuous, the system is surely healthy until 2003-11-07. The threshold for performing the ND is set conformingly to this assumption, for every model considered. Otherwise, the performance of any model could be artificially made as good as desired, by simply setting the threshold to a lower value.

The configuration file is set to use the data from the "bearing 3x" sensor, extracting all the time-domain and frequency-domain features described in **??**. The

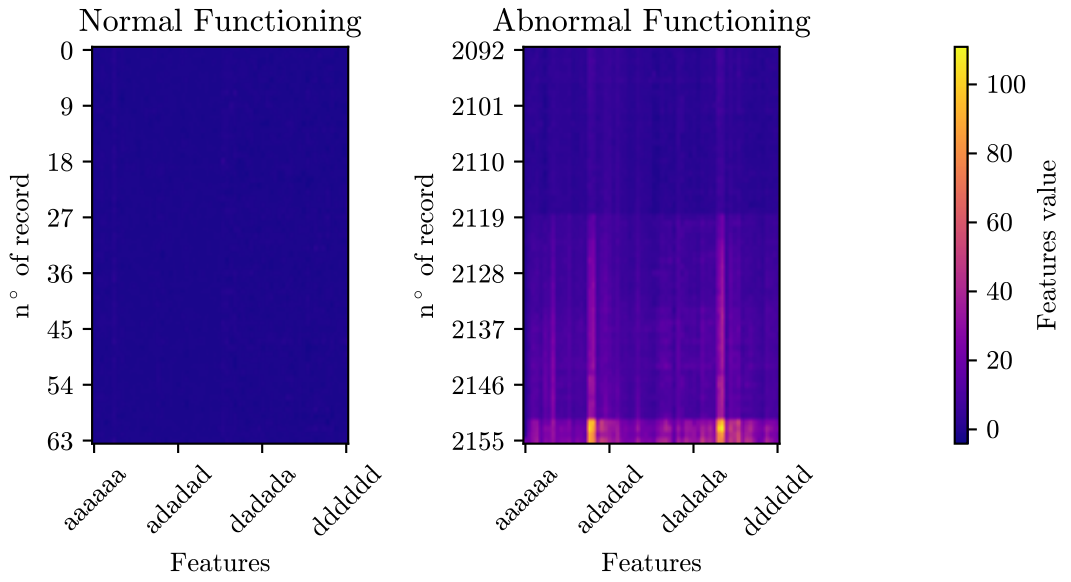

Figure 2.1: *Heatmap of the standardized features value for the test n°1 of IMS dataset*

training dataset is used to train the MLA to recognize the normal behavior of the bearing, and the testing dataset is used to validate the trained model. The **??** shows the parameters of test No.1 of the IMS dataset. For display purposes, the features are standardized, and the heatmap of the standardized features is shown in **figure 2.1** in normal and abnormal conditions.

The abstract version of the FiA has been used to extract the features from the dataset, creating all the snapshots in the set $\mathbf{S} = \{\boldsymbol{\mathcal{S}}_1, \boldsymbol{\mathcal{S}}_2, \ldots, \boldsymbol{\mathcal{S}}_{500}\}$. These snapshots are stored in the *unconsumed* collection of the database.

2.1.1 Training - K-means

Using the commands of the CLI, the training procedure has been launched:

```
1  C:/Users/JohnSmith/Code/framework> python ./MASTER.py
   ↪ run-feature-agent
2  C:/Users/JohnSmith/Code/framework> python ./MASTER.py
   ↪ run-machine-learning-agent novelty train
```

where the first command runs the FiA and the second one runs an "healthy" instance of the MLA in training mode. At this point, the MLA asks the user to move the snapshots from the *unconsumed* to the *healthy* collection, since the *healthy* collection is empty. After the confirmation, the MLA starts the training with a

different number of clusters, and outputs the scoring in the form of silhouette and inertia scores. The results are shown in **figure 2.2** and **figure 2.3**. The user can confirm that the best number of clusters is 2, as the silhouette score is the highest and the inertia score is at the POF point, or insert another number of clusters, remembering that it is best to overestimate the number of clusters to increase the system sensitivity, as discussed in **??**.

In this case, the number of clusters has been set to 2, so that the MLA saves the model trained with $n = 2$ into the database. Even if the feature space has high dimensionality, the agent plot to the user also a scatter plot of a subset of features of the training dataset, to have a visual feedback of the clustering, as shown in **figure 2.4**, where the points are the snapshots, the crosses are the centroids and the colors represent the assigned cluster. We can observe that selecting 2 as the number of clusters is adequate and that the projections of the clusters' shapes on some planes are not perfectly spherical but, at least, they are not too elongated. This is a good sign for the K-means algorithm, as discussed in **??**.

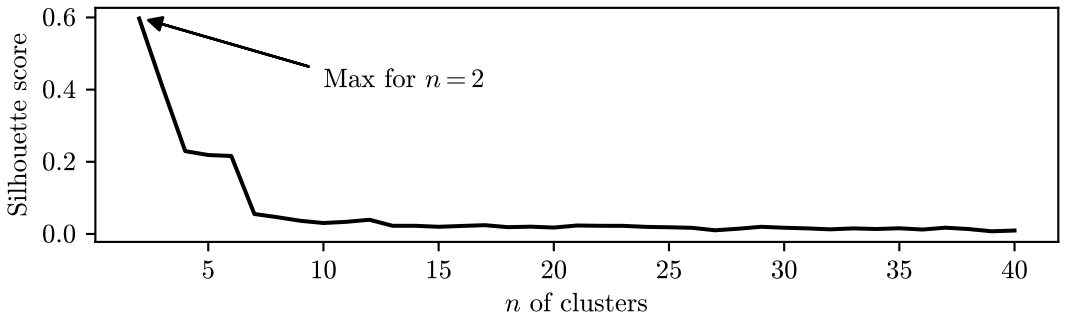

Figure 2.2: *Silhouette score for clustering the test n°1 of IMS dataset (K-means)*

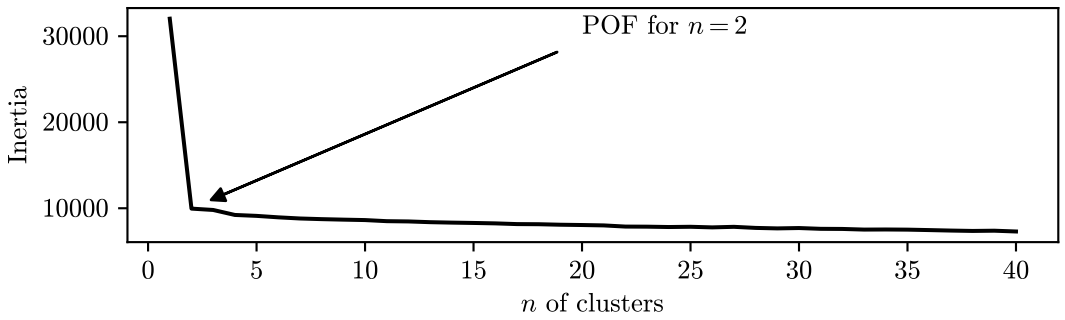

Figure 2.3: *Inertia score for clustering the test n°1 of IMS dataset (K-means)*

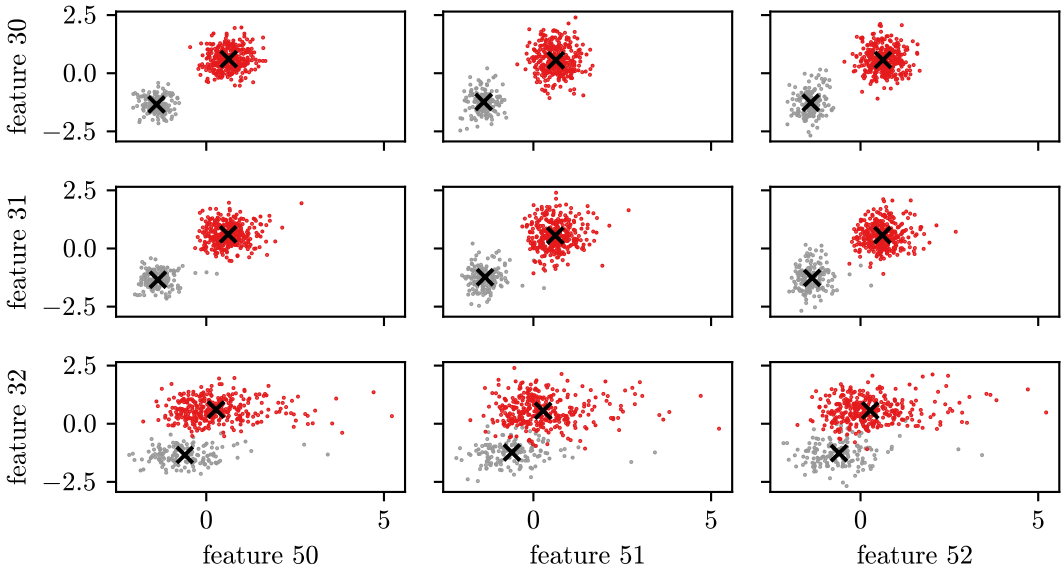

Figure 2.4: *Scatterplot of training snapshot for the test n°1 of IMS dataset*

2.1.2 ND Validation - K-means

Using the validation partition of the dataset, it is possible to set the MLA in *evaluate* mode. The FiA uses the validation partition and fills the *raw* collection with the time-series. The FA extract the features and continuously fill the *unconsumed* collection with the snapshots. The MLA evaluates the snapshots according to **??** and plots the result, as well as generating a warning if the novelty metric is greater than a certain threshold (in this case 50%, but it is configurable in the usual `.yaml` file). The results are shown in **figure 2.5**, where we can see that the framework detects the novelty quite early, at 2003-11-16 07:46, while the dataset authors, declared the test finished because of bearing defects (not catastrophic failures) at 2003-11-25 23:40. The comparison of the margin of early detection for different algorithms will be resumed later.

In **figure 2.6**, a detailed view of the ND metric becoming consistently greater than the threshold is shown.

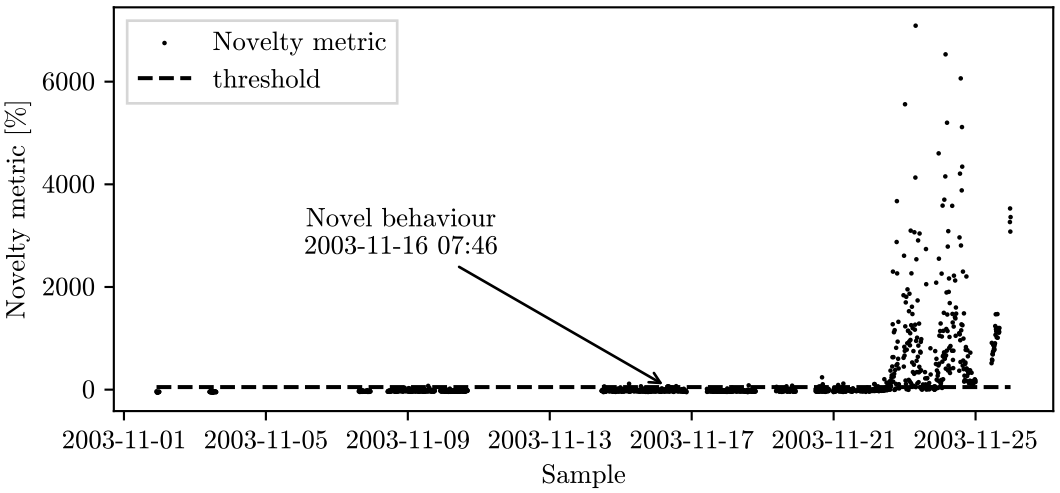

Figure 2.5: *Results of ND for the test n°1 of IMS dataset (K-means)*

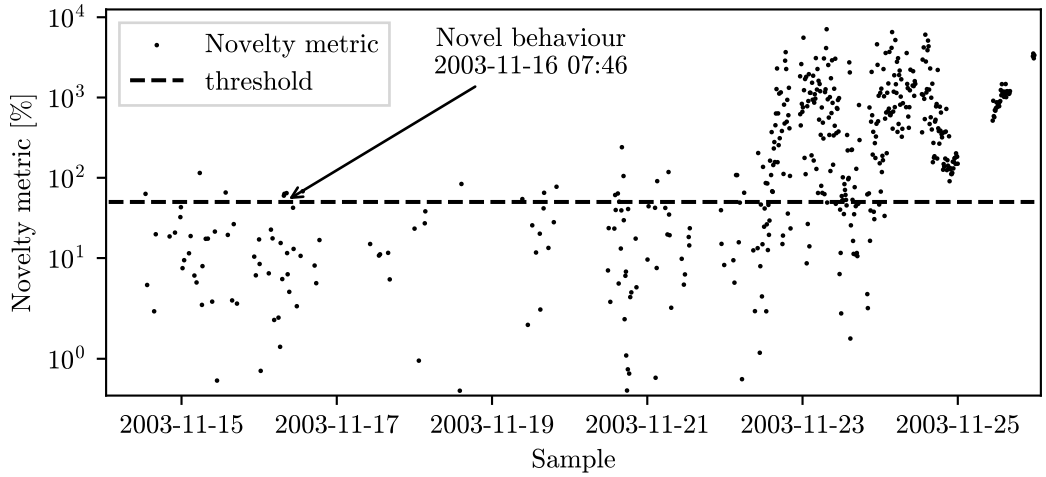

Figure 2.6: *Results of ND for the test n°1 of IMS dataset (K-means) - detailed view*

2.1.3 Training - DBSCAN

Using the same partition of the dataset as for the K-means training, we can train a DBSCAN model. In this case, the silhouette score has to be used to select a suitable value of the radius ε. As shown in **figure 2.7**, the optimal value is 8, which corresponds correctly to the generation of two clusters.

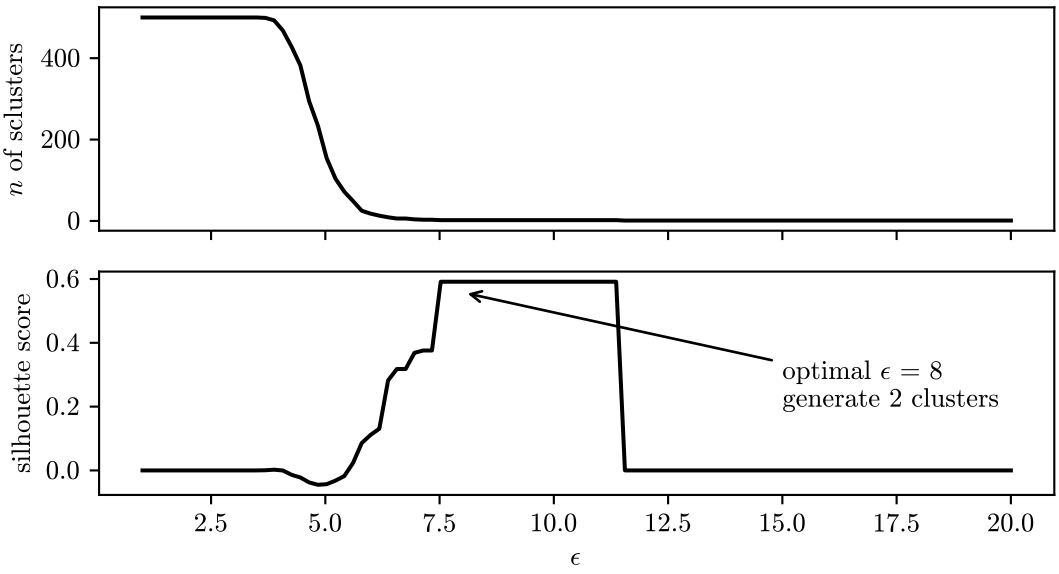

Figure 2.7: *Silhouette score for clustering the test n°1 of IMS dataset (DBSCAN)*

2.1.4 ND Validation - DBSCAN

As it has been done for the K-means, the validation partition of the dataset is now used for performing ND with the DBSCAN model, as described in **??**. The result is shown in **figure 2.8**, where we can see that the DBSCAN model detects the novelty at 2003-11-22 15:06, that is quite early, but not as early as the K-means model. This is because the metric generated by the DBSCAN model has a greater variance so, instead of increasing consistently, it overshoots the threshold quite before this time but fails to consistently stay above the threshold.

2.1.5 Training - GMM

Let's now try with the GMM model. The metric for selecting the number of clusters is now the BIC and the AIC, as shown in **figure 2.9**. The two metrics diverge but, as discussed in **??**, the AIC tends to perform better. In this case, minimizing the AIC leads to select 25 as the number of clusters, which is much more than what was selected with the K-means, but still a reasonable choice, also considering that the GMM is a soft clustering algorithm and that we are using the density as a metric to perform ND.

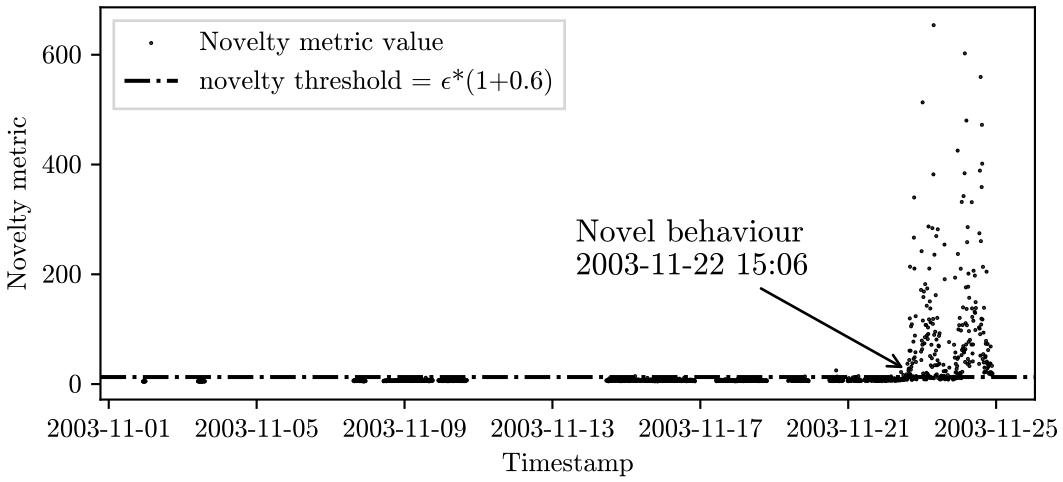

Figure 2.8: *Results of ND for the test n°1 of IMS dataset (DBSCAN)*

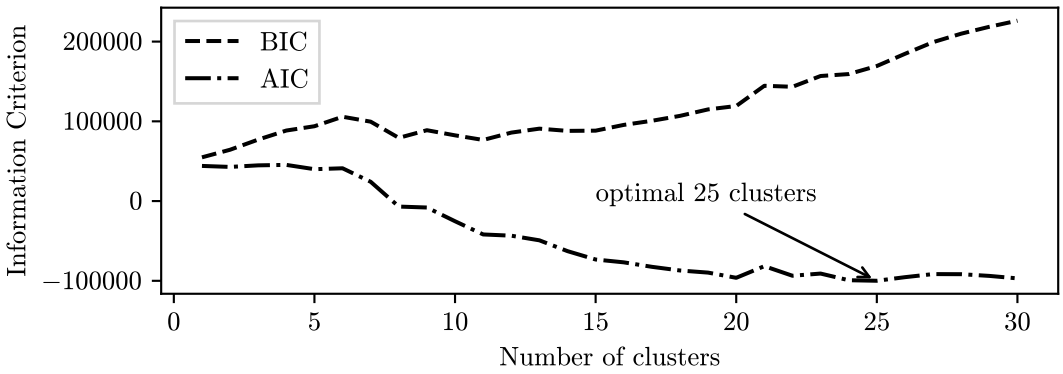

Figure 2.9: *BIC and AIC for clustering the test n°1 of IMS dataset (GMM)*

2.1.6 ND Validation - GMM

The validation partition of the dataset is now used for performing ND with the GMM model. The result is shown in **figure 2.10**, where we can see that the GMM model detects the novelty at 2003-11-22 03:47. The considerations about this result are the same as for the DBSCAN model, and in fact, the timestamp of the detection event is really close to the one obtained with DBSCAN. In **figure 2.10**, the metric (density value) appears in colored dots, as each color represents the cluster to which the snapshot has been assigned.

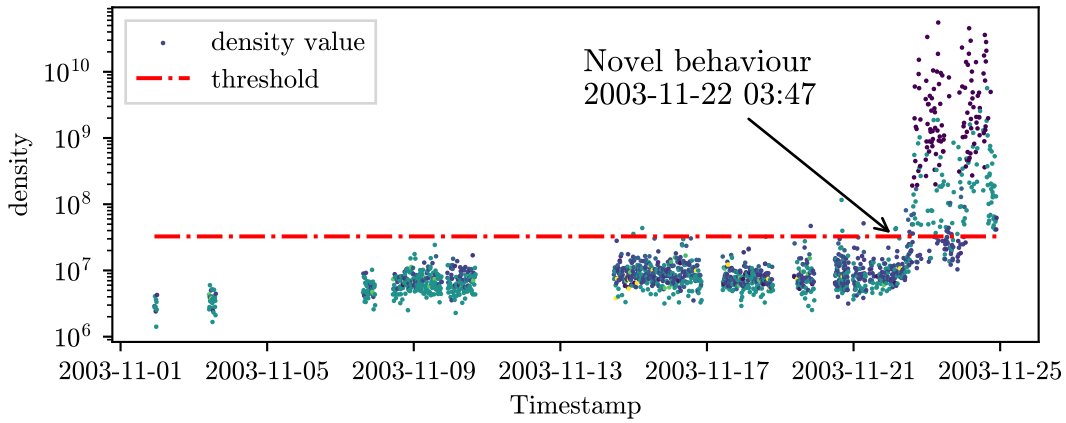

Figure 2.10: *Results of ND for the test n°1 of IMS dataset (GMM)*

2.1.7 ND Validation - Bayesian GMM

The other Gaussian model is the BGMM, since this approach is totally unsupervised, only the validation results are reported here. The result is shown in **figure 2.11**, where we can see that the BGMM model detects the novelty around the same time as the GMM model, at 2003-11-22 03:45.

In both GMM and BGMM the metric (density value) spans a lot of decades, so the plots are done on a logarithmic scale.

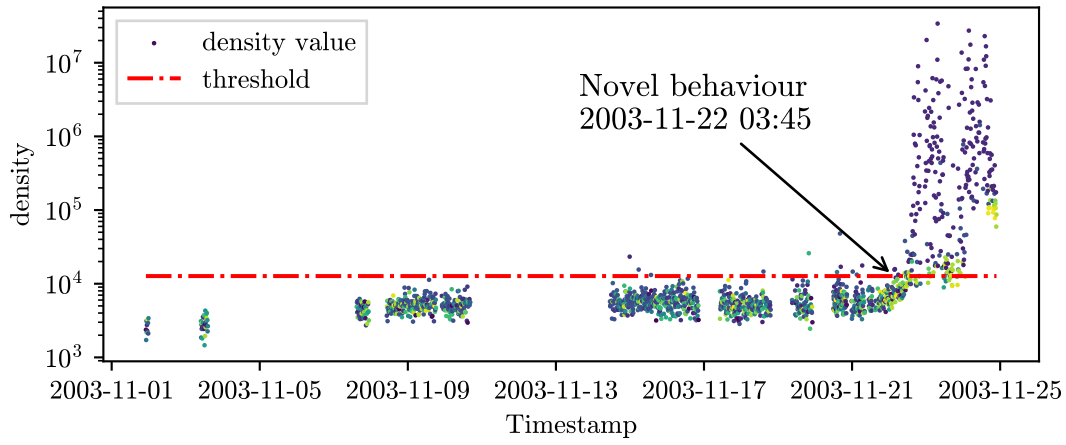

Figure 2.11: *Results of ND for the test n°1 of IMS dataset (BGMM)*

2.1.8 ND Validation - ν-SVM

The next algorithm to test is the ν-SVM. Again, this is totally unsupervised, so only the validation results are reported here. The novelty metric evolution over time is shown in **figure 2.12**, where we can see that the ν-SVM model detects the novelty at 2003-11-22 14:56, which is comparable with the DBSCAN and GMM models.

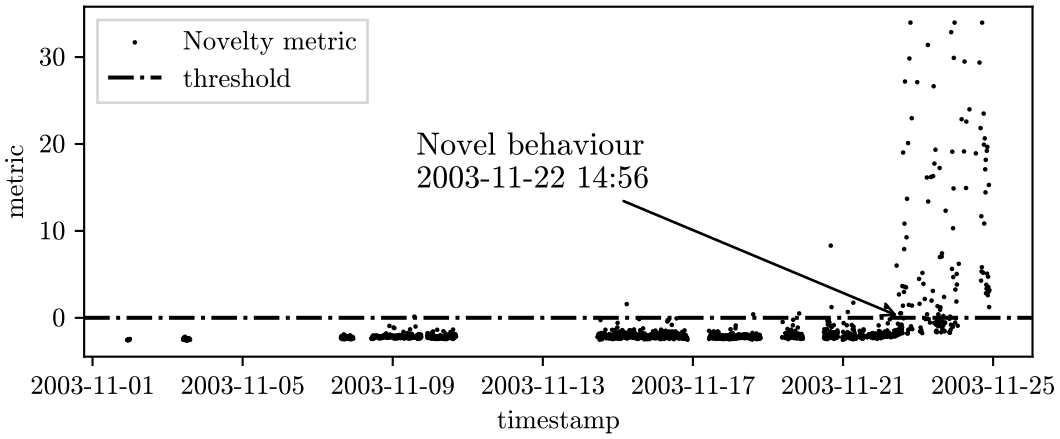

Figure 2.12: *Results of ND for the test n°1 of IMS dataset (ν-SVM)*

2.1.9 ND Validation - iForest

The second last technique to test is the one based on the iForest model. The result is shown in **figure 2.13**, where we can see that the iForest model detects the novelty at 2003-11-16 10:08:46, that is a good result comparable with the K-means model. The problem with the metric of the iForest is that it increases a lot the variance around the ND event, but the mean does not increase consistently, so a lot of snapshots are discarded as outliers, before the ND event.

This is, in my opinion, a promising approach. With these settings a lot of snapshots are discarded as outliers, but with a different outlier filter, based on the percentage of novelty samples in a window, the iForest model could perform even better.

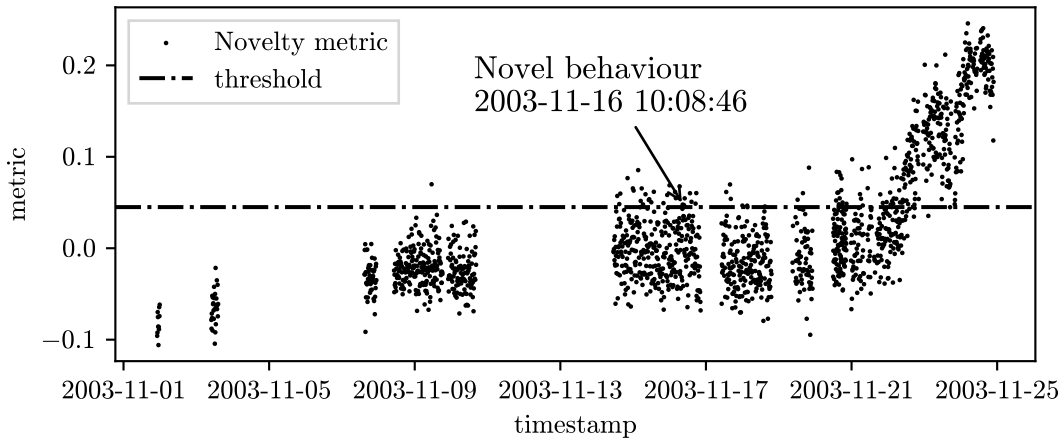

Figure 2.13: *Results of ND for the test n°1 of IMS dataset (iForest)*

2.1.10 ND Validation - LOF

The last algorithm to test is the LOF. The result is shown in **figure 2.14**, where we can see that the LOF model detects the novelty at 2003-11-16 07:49, which is a good result comparable with the K-means model. It doesn't have the same problem as the iForest, as there aren't as many discarded snapshots before the ND event.

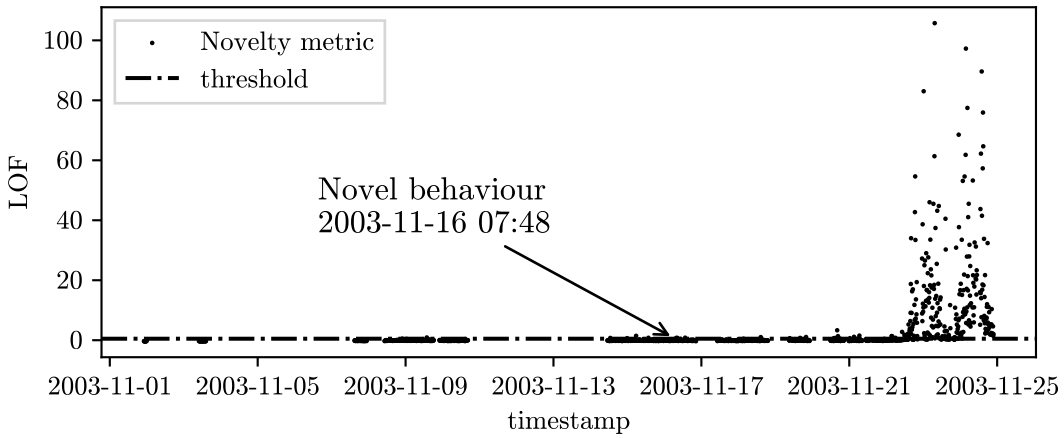

Figure 2.14: *Results of ND for the test n°1 of IMS dataset (LOF)*

2.1.11 Comparison of the results

Comparison between the models

Table 2.1: *Comparision of the results for the test n°1 of IMS dataset.*

Algorithm	**ND event**	**Lead Time** [min]
K-means	2003-11-16 07:46	**13913**
DBSCAN	2003-11-22 15:06	4833
GMM	2003-11-22 03:47	5513
BGMM	2003-11-22 03:45	5514
ν-SVM	2003-11-22 14:56	4844
iForest	2003-11-16 10:08	13771
LOF	2003-11-16 07:48	13912
P2P without any ML	2003-11-22 16:06	4774

In **table 2.1**, the results of all the previous tests are resumed, together with the result of performing ND without any machine learning algorithm, but just setting a threshold on the P2P value of the time-series, as it was previously shown in **??**. This last basic approach detects the novelty around the afternoon of 2003-11-22.

The ν-SVM and the DBSCAN models are not performing much better than not even using machine learning (at least on this dataset signal). The GMM and BGMM models are performing slightly better, but the margin is so low that the result may be biased by the threshold setting. The iForest, LOF and K-means models are performing better, they are all very close to detecting the novelty, around 14000 min = 9.7 days before the end of the test. The K-means model is the one performing the best, but just slightly better than the iForest and LOF models so, again, this small difference may not be significant. However, as discussed in the previous chapters, the K-means model will be used in the rest of the work, as it is also the most simple and interpretable model.

Comparison with another approach

As anticipated in the **??**, about the State of the Art, the signal of the same bearing (Bearing 3x) of this same test has been used in [6]. In their research, the authors used a different approach, based on regression, and obtained the result reported in **??**

Comment about the comparison

Every system that outputs a warning based on a trigger on a threshold is highly sensitive to the value of the threshold itself. This means that the comparison of the results is not straightforward, and quite opinable, because selecting a low threshold will make almost every system trigger earlier. The measure to take into consideration, in my opinion, is how many false positives are generated if the threshold is lowered, and how small the variance of the metric is. An high variance, on this dataset, means that the system is very sensitive while evaluating quite similar signals.

2.1.12 RUL Predictions validation - K-means

After the ND event, the MLA start predicting the future evolution of the novelty metric, and it superimpose the prediction curve to the same plot displayed to the user. The fitting procedure is the closed form solution of the LS problem applied to an exponential curve of **equation 1.2**, as described in **subsection 1.1.6**. The samples used for the regression of the curve are the last 230 before the current one. This parameter of the framework is configurable in the `.yaml` file.

Some good predictions are shown in **figure 2.15**. The RUL is the difference between the intercept of the prediction curve and another threshold, higher than the one used for ND, and the current time. In the figure, the blue line is a prediction made just a few hours after the ND event (the vertical dashed line marks the time of the prediction). The same concept applies to the other predictions performed in later times.

In some circumstances, the novelty metric starts decreasing slightly, on average, as can be seen around 2003-11-21. In this case, if the novelty metric has this behavior for several snapshots (≈ 230), the fitted curve will be a decreasing exponential, as shown in **figure 2.16**.

If this situation occurs, the intercept with the threshold does not exist, and the RUL prediction fails, so the interpretation of the RUL is left to the user. In some cases, the defect in a system can "self-heal" (for example a crack in a bearing can be polished with the use [7]). If this behavior is possible for the system, this situation can be interpreted as a sign that the system is going to return to normality. Otherwise, the user can retain the previous RUL prediction as the RUL of the system.

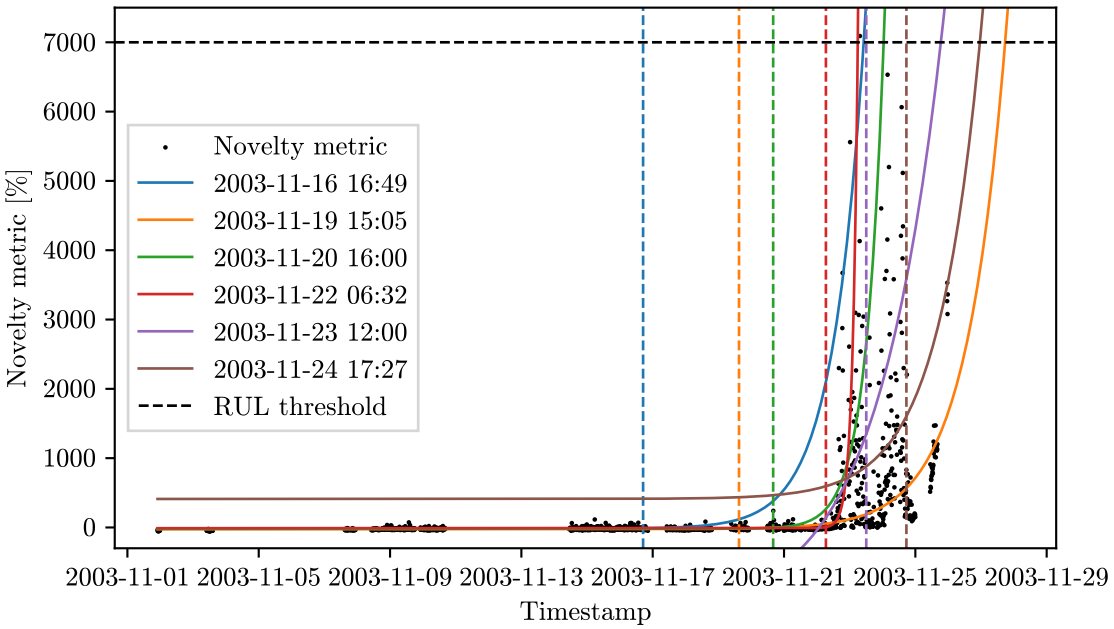

Figure 2.15: *RUL prediction at different instants after the ND event. The dashed lines are the instants of the predictions corresponding to the same color solid line prediction.*

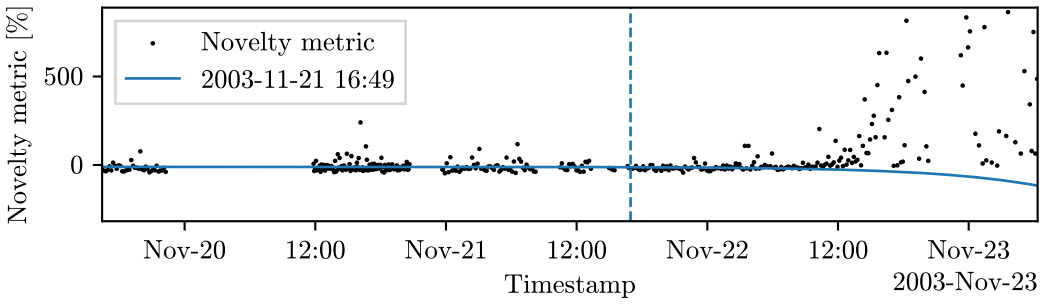

Figure 2.16: *Failed RUL prediction.*

2.1.13 Retraining, evaluating and predicting after ND event

If the user, after the ND event, performs an investigation that leads to the belief that the system is still healthy, the user can turn the MLA in *retrain* mode. In this case, the snapshots that are in the quarantine collection, are moved to the healthy collection (faulty if the instance is for FD and the investigation reveals that the fault is real). The model is then retrained with the new data with the same procedure used for the first training (silhouette and inertia scores are computed

and the user is asked to confirm the number of clusters).

Let's investigate what would have happened if the user declared the system healthy at 2003-11-23 00:00, in the previous scenario of "Bearing 3 x" signal in the IMS dataset. The MLA suggests that the best number of clusters is still two, so it has been retrained with the new data. The result of the updated model performing ND is shown in **figure 2.17**. The predictions of the RUL are shown in **figure 2.18**.

This test shows that in an increasingly decaying system retrained with data very close to the fault condition the MLA is able to detect the fault again. This comes at the cost of a later detection, and the first predictions after the ND event are not as good as the previous ones. Anyway, the RUL predictions still become more accurate as time passes, and the RUL prediction at the end of the test is still quite accurate (on the same day of the event).

Another consideration is about the RUL threshold: since the model has been retrained with "worse" data, the threshold for the RUL prediction should be set to a lower value, because now the clusters are either more in quantity, distorted or bigger, so it is unlikely that the novelty metric can still reach the same high values estimated before the retraining.

An intuition about why the sensitivity of the system is reduced after the retraining comes by examining the scatter plot of the snapshots in the feature space, shown in **figure 2.19**, where all the snapshots extracted from the dataset are displayed. The clusters are more elongated and much bigger. These shapes arise gradually from the original ones of **figure 2.4** so, by performing a retrain, both the effect of producing bigger clusters and one of the clusters being much more elongated play a role in reducing the sensitivity of the MLA.

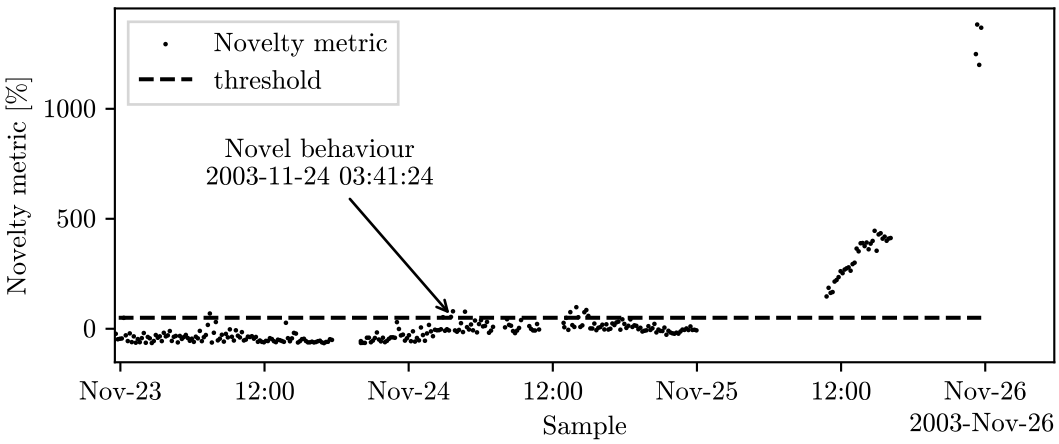

Figure 2.17: *Results of ND for the test n°1 of IMS dataset (K-means) - retrained model*

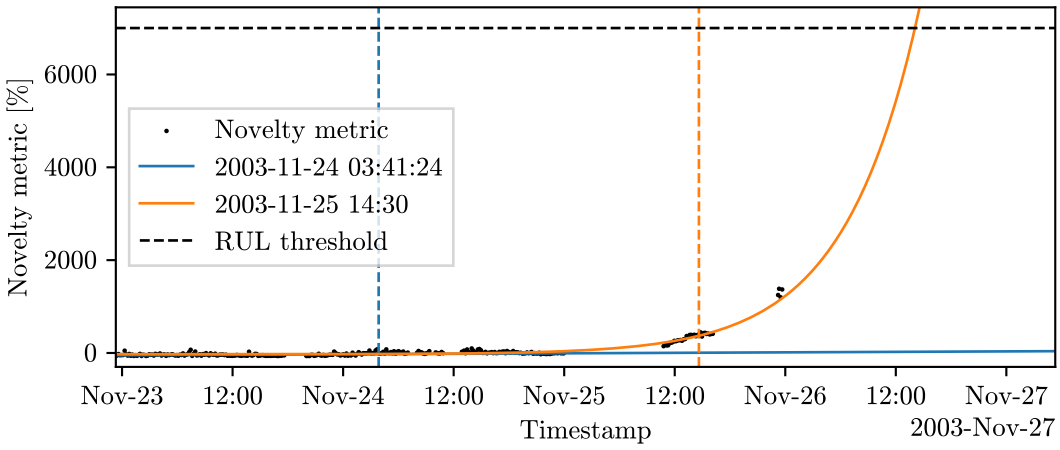

Figure 2.18: *RUL prediction at different instants after the ND event. The dashed lines are the instants of the predictions corresponding to the same color solid line prediction.*

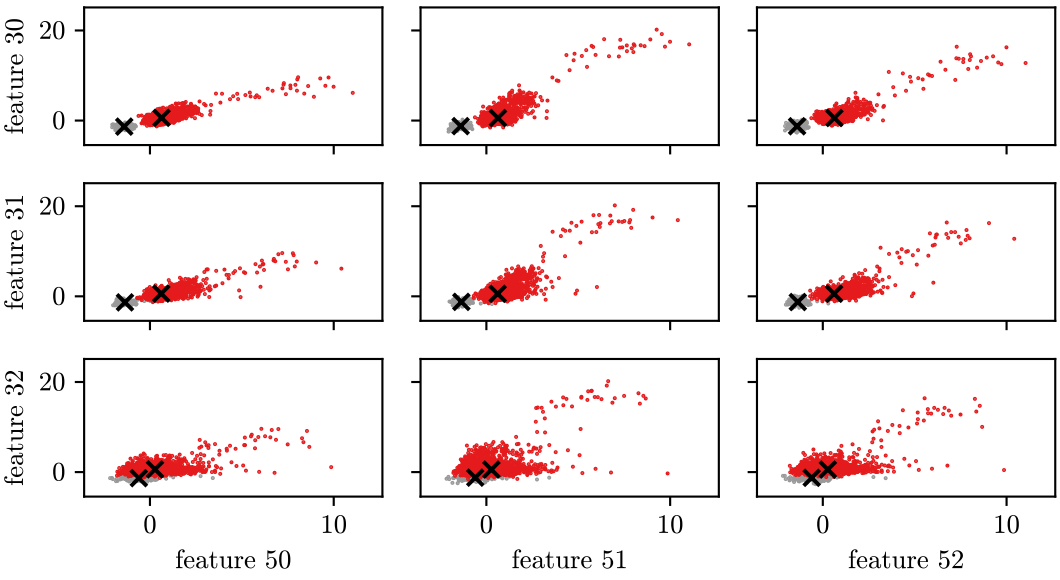

Figure 2.19: *Scatterplot of all the snapshot for the test n°1 of IMS dataset*

2.2 IMS dataset No.1 - All sensors

In the previous section, an extensive test of the framework has been performed on a single signal from the dataset (Bearing 3 x). So the warning given by the MLA was

indicating a problem in a specific component of the maintained system. Let's now test a configuration that takes into account all the signals of the dataset, so all eight signals from the four bearings are used for feature extraction. This configuration should be able to detect a generic novel behavior of the system or, better, a situation in which the system is abnormal as a whole (the signals may be normal but the combination of them may be abnormal). In this case, the configuration file has been set to use all the time-domain and all the frequency-domain features, and the MLA has been trained with the same procedure as before.

2.3 IMS dataset No.2 - Bearing 3x sensor

2.4 IMS dataset No.3 - Bearing 3x sensor

2.5 Experiments on a laboratory shaker - Test 1

After the PC implementation of the framework has been tested widely on the IMS dataset, the edge computing implementation had to be validated experimentally. The first test was done with a laboratory shaker, which is basically a powerful active speaker with a really wide band that can be attached with a bolt to a structure, to vibrate it.

In this case, an accelerometer, whose key specifications are shown in **table 2.2**, was used to capture the vibration signal. The accelerometer was attached to the shaker, with a custom 3D-printed fixture. This first test has the scope of checking the capability of the edge computing implementation to detect a new low amplitude harmonic in the signal. The signal is generated as a `.wav` file and fed to the shaker by a player. Both the input of the shaker and the output of the accelerometer were monitored with a digital oscilloscope. The setup is shown in **figure 2.20**.

Table 2.2: *Specifications of the ADXL335 Accelerometer*

Parameter	**Value**
Supply Voltage	1.8V to 3.6V
Sensing Range	±3g
Sensitivity	300 mV/g
Bandwidth	0.5 Hz to 1600 Hz
Output Type	Analog
Output Voltage Range	0V to V_{CC}
Operating Temperature	-40°C to +85°C
Package	3mm × 5mm × 1mm

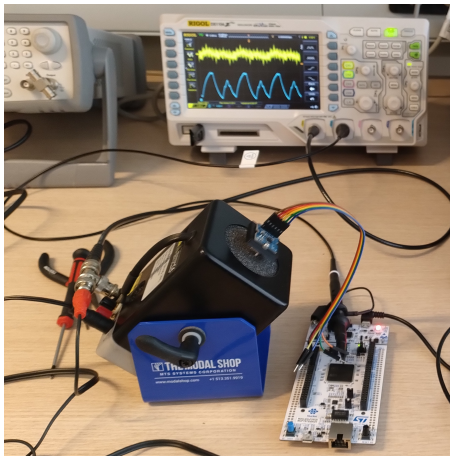

Figure 2.20: *Setup of the shaker tests.*

Table 2.3: *Harmonic coefficients for the shaker test. Wave 1 and Wave 2 are training signals, and Harmonic Injection is the signal to be detected.*

Signal Name	**Harmonic frequency** [Hz]						**Amplitude** $[\text{mV}]_{pp}$
	30	70	100	300	800	1400	
Wave 1	0.1	1.0	1.0	-	1.0	1.0	1000
Wave 2	0.1	0.8	1.0	-	3.0	0.6	1000
Harmonic Injection	0.1	1.0	1.0	0.1	1.0	1.0	1000

2.5.1 Training and evaluating

The framework was firstly set to gather the data, extract the features according to the configuration, and send the data to the PC for training. The training signals are two waves with different harmonic content and the test signal is very similar to one used for training, except for the presence of an additional harmonic with a small amplitude. The train and test signals harmonic content is reported in **table 2.3**. The amplitude of the vibration has been tuned at each test to ensure that the microcontroller was reading a signal of 1V_{pp}. The amplitude of the signals has been kept constant to test the capability of the framework to detect the frequency content of the signal in the feature extraction phase. The waveform of the test signals is shown with the one of one training signal in **figure 2.21**, to show the similarity of the two signals.

The setting of the framework can be resumed as follows:

- 67 features extracted from the signal ($2^6 = 64$ features from the wavelet decomposition, mean, P2P, and RMS);

- 110 samples for training for each signal.
- sampling frequency of 5kHz, for 1s of acquisition.

After the data gathering was completed, the training was done on the PC, as usual. The silhouette score correctly suggested 2 as the best number of clusters to be used. The PC part of the framework outputs the `model.h` file directly in the embedded project folder, so just a new upload of the firmware was needed to test the detection. The microcontroller was then set in *evaluation* mode and both the two training signals and the test signal were fed to the shaker.

2.5.2 Results

The result of the novelty detection is shown in **figure 2.22**. The result is consistent with the expected outcome, as the training signals produced a negative novelty metric, while the test signal produced a positive (and quite high) novelty metric. The spectrum of the signals used is shown graphically in **figure 2.23**.

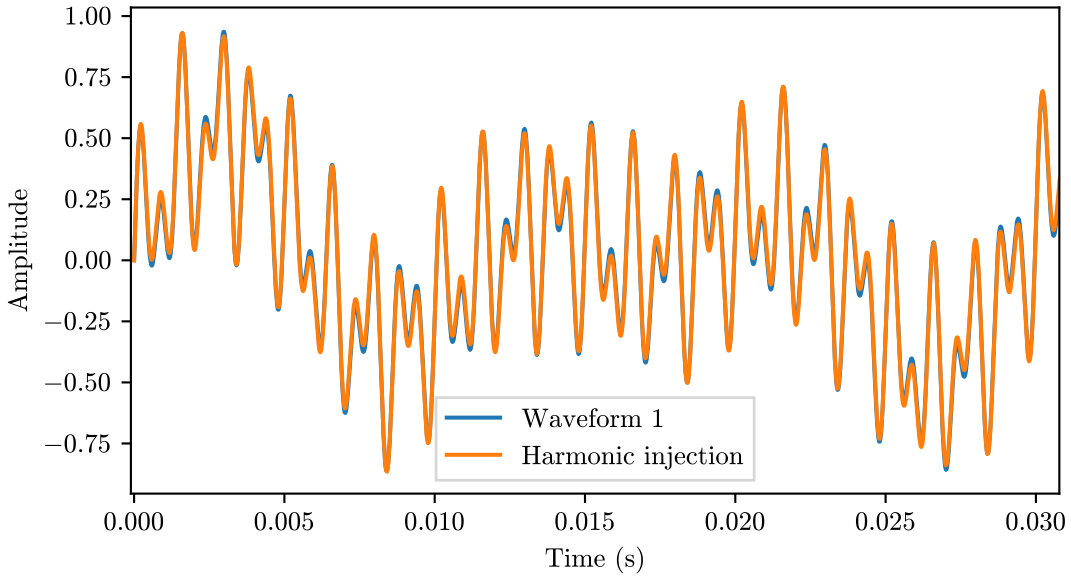

Figure 2.21: *Waveform comparison of the shaker test.*

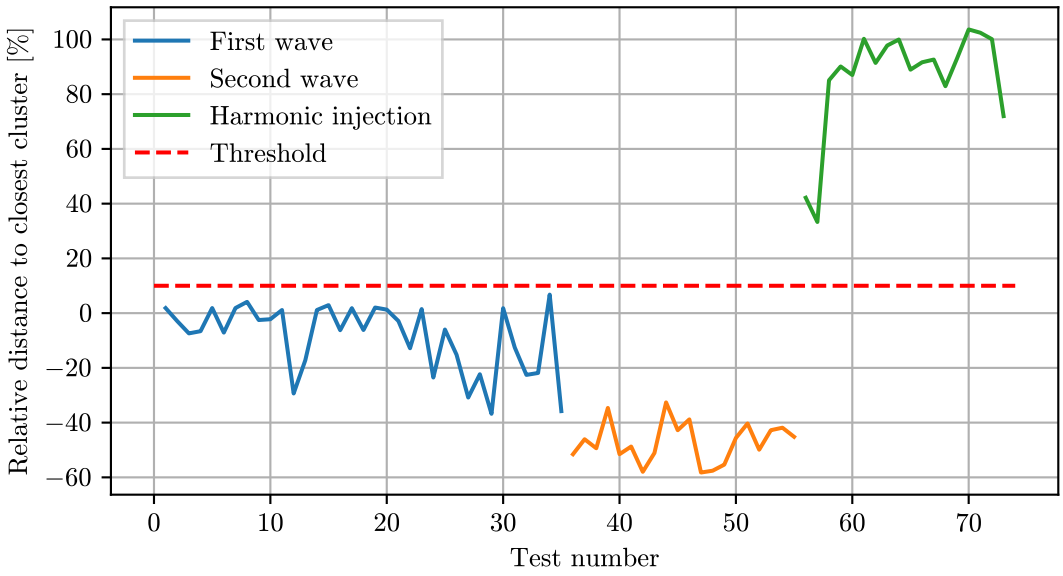

Figure 2.22: *Novelty detection result*

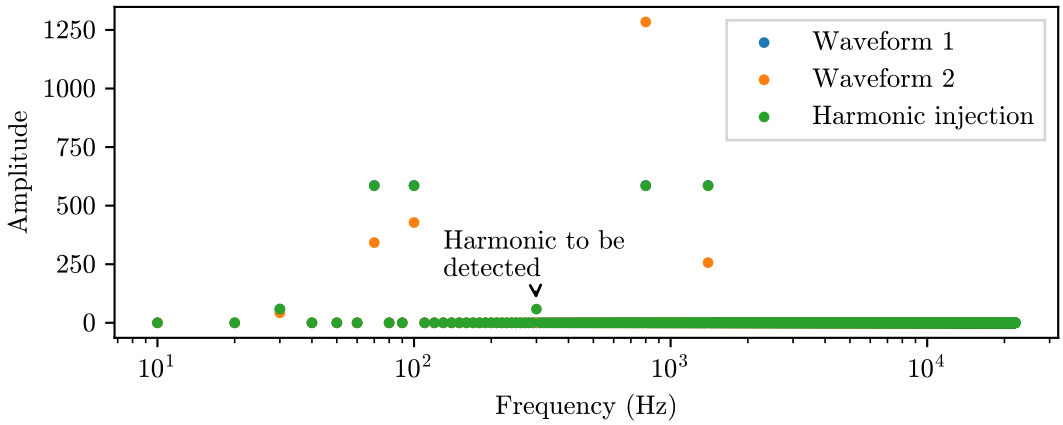

Figure 2.23: *Spectrum of the waveforms.*

2.6 Experiments on a laboratory shaker - Test 2

In the previous section, the first test on the shaker was presented. The test has shown the capability of the framework to detect unknown harmonics. A second test was done to evaluate the capability to detect time-domain variations and the effect of reducing the frequency resolution.

2.6.1 Training and evaluating

This new configuration has been set to use only 4 frequency-domain features and the same 3 time-domain features of the previous test, for a total of 7 features. The signal used for training and testing is resumed in **table 2.4**. The set is composed of the same signal at different amplitudes used for training and testing, plus another signal with different frequency content but the same amplitude as a training signal used for testing.

The training has been carried out in the same way as the previous test, the training of the K-means model has been done with 4 clusters, and loaded on the microcontroller.

Table 2.4: *Parameters of the second shaker test.*

Harmonic frequency [Hz]					**Amplitude** [mV_{pp}]	**No. of snapshots**	
10	30	60	70	100		Train	Test
-	0.1	-	1.0	1.0	580	100	10
-	0.1	-	1.0	1.0	1000	100	10
-	0.1	-	1.0	1.0	1980	100	10
-	0.1	-	1.0	1.0	1540	100	10
-	0.1	-	1.0	1.0	2000	-	20
-	0.1	-	1.0	1.0	0	-	10
-	0.1	-	1.0	1.0	800	-	10
-	0.1	-	1.0	1.0	200	-	10
-	0.1	-	1.0	1.0	1220	-	10
1.0	1.0	0.1	-	-	1540	-	10

2.6.2 Results

The result of the novelty detection is shown in **figure 2.24**. The first 4 lines have been correctly identified as normal, as they were in fact a repetition of the training signals. Then the purple and cyan line in the figure is the same training signal, but 20 mV higher in amplitude w.r.t. the training signal. The novelty metric overshoots the threshold in 5 samples out of 20. An increase of 2% in amplitude generates the ND event 25% of the times can be observed with this signal.

The brown, grey and light-green lines are the same signal, but with a bigger difference in amplitude w.r.t. the training signal. All the snapshots of these signals correctly generated a novelty metric above the threshold. The blue line is the signal with a different frequency content, and it has been correctly identified as a novelty event, this is the confirmation that even with just 4 frequency bins, the wavelet

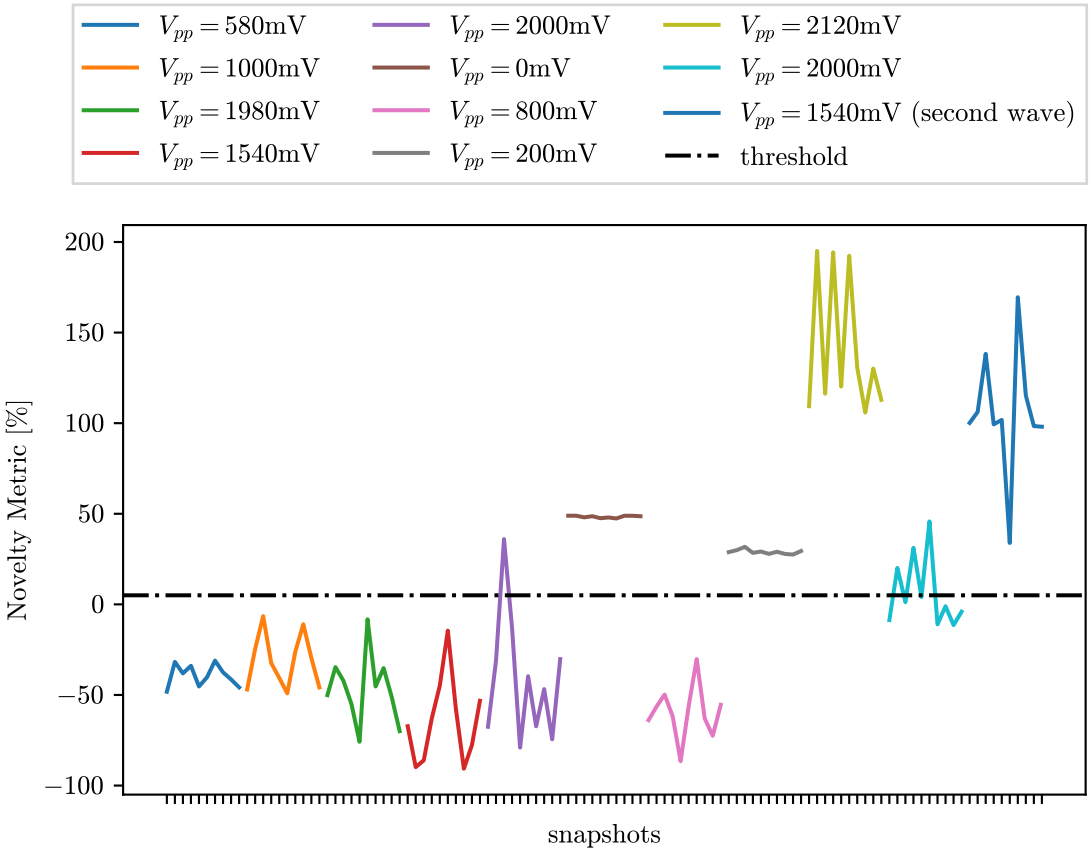

Figure 2.24: *Novelty detection result*

decomposition is still generating features that are informative.

The pink line is the test signal with amplitude of 800mV. It's evident that the novelty metric is below the threshold, and the signal has been classified as normal even if it is not in the training dataset. Let's investigate how this happened. The first consideration is that the 800mV amplitude is quite tight to both the 1000mV and 580mV signals used for training. Moreover, in this case, the total number of features is just 7. This allows plotting all the features against each other, to see why the ND event has not been detected. In **figure 2.25** the scatter plot of the features is shown. It's evident that, in this environment, even performing the standardization of the features, the clusters are still very elongated, resembling almost a line. To fit an elongated cluster in a hypersphere, it is inevitable that in some sections, the hypersphere will not closely surround the cluster, leaving "space" for false negative results. Another problem is that the k-means algorithm tends to split long clusters. In the figure, the red dots are the false negative results, and the gray shades are the hypersphere projection on the considered features plane. The

black dots are the training data. The effect of the elongated clusters is particularly evident in the plot of the "Feature 3" against "Feature 2", where the red dots are in between two clusters, that are modeled as one. On the other hand, looking at the plot of "Feature 1" against "Feature 4", a very long cluster has been split in two. This is an example of exploiting the limitations of the k-means algorithm anticipated in **??**. For completness, in **figure 2.25**, also the true positive results are shown, as magenta dots.

2.6.3 Possible improvements

The environment of this test is very challenging for the k-means algorithm. As discussed in **??**, there are algorithms that are not affected by the clusters' shapes. The candidate algorithms that may perform better in this situation are the LOF, the iForest and DBSCAN. A future work could be to implement these algorithms in the edge computing framework, despite being more demanding in computational power and memory, and test them in this environment.

As proof of concept, the LOF implementation in `python` has been used to perform ND on the same dataset used in this section in edge computing. The results are reported in **??**. The LOF algorithm has been able to correctly identify all the ND events, even the signals with just 20mV variation from the training dataset, and the 800mV signal that was problematic for the K-means. The LOF, however, generated a false positive on the 580mV signal. This false positive may be avoided by increasing the threshold, but this would also increase the false negative rate.

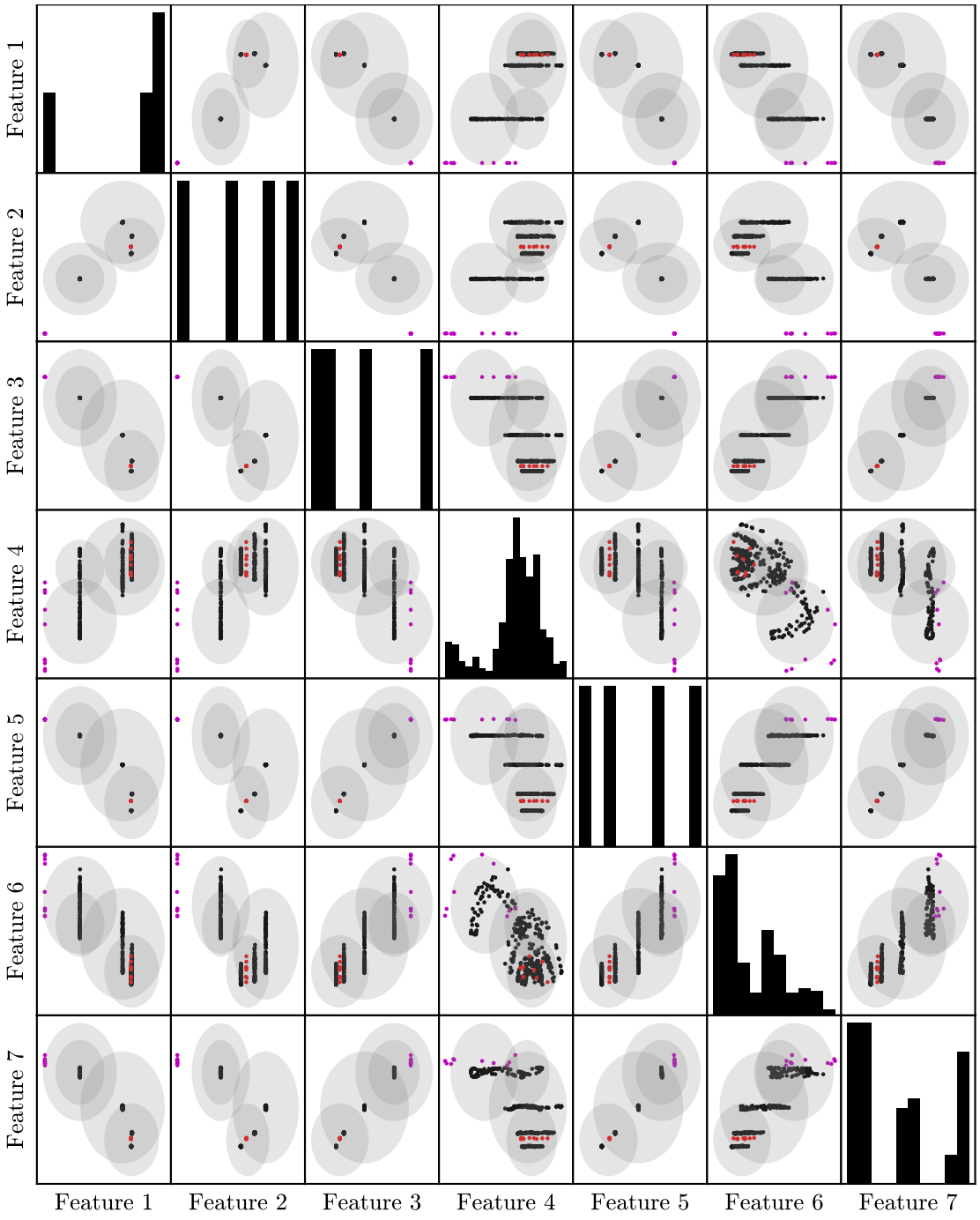

Figure 2.25: *False Negative and True Positive results. On the diagonal, there is a histogram of the feature values. The off-diagonal plots are the scatter plots of the features. The shades are the projection of the clusters on the considered plane. (Red: False Negative, Magenta: True Positive, Black: training data)*

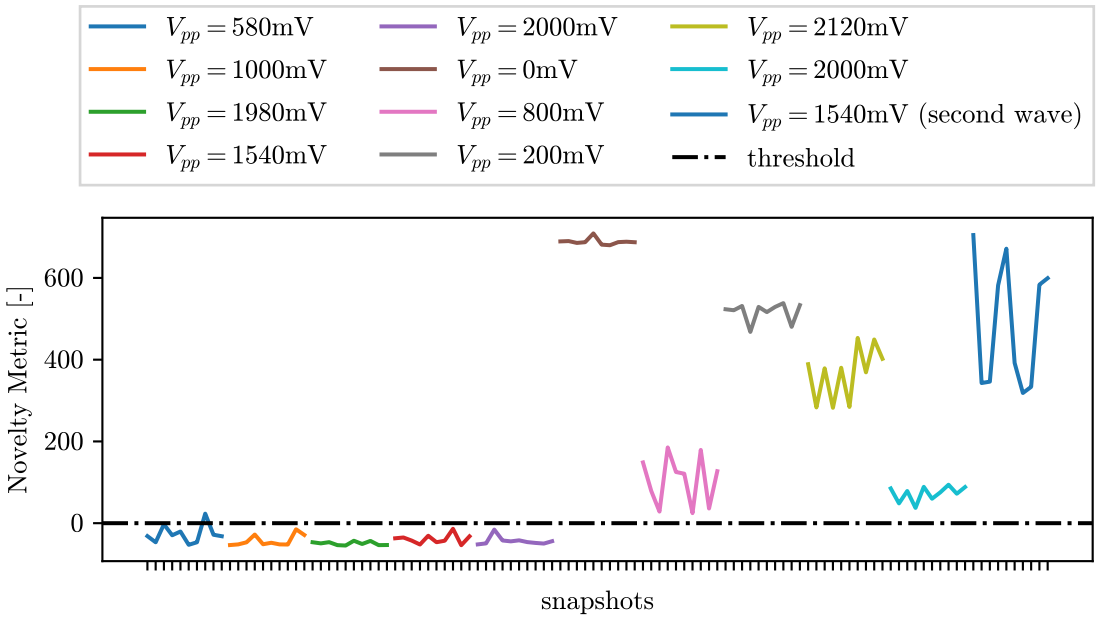

Figure 2.26: *LOF novelty detection result*

2.7 Experimental validation on a linear axis

The experimental validation reported in the previous **section 2.5** and **section 2.6** was carried out in a well-controlled environment with a shaker that was able to generate vibration according to specific references. To further test the framework, a real-world application is considered in this section. The setup consists of a machine equipped with a linear axis, that is used to move a platform. On the moving platform the same accelerometer described in **table 2.2** has been attached using a custom 3D-printed fixture.

The test consists of defining a set of movements to be actuated by the platform, the accelerometer is used to capture the characteristics of each movement. As done previously, some movement profiles are used for training and others for testing. The position reference is shown in **figure 2.27**, and the parameters of the profiles are resumed in **table 2.5**.

Figure 2.27: *Position reference for the linear axis test.*

Table 2.5: *Harmonic coefficients for the shaker test.*

Profile N.	**Speed** [ms^{-1}]	**Acceleration** [ms^{-2}]	**Jerk** [s]
1	0.8	6	0.02
2	0.4	3	0.02
3	0.4	6	0.02
4	0.6	8	0.02

2.7.1 Training

To perform the training, a loop has been implemented on the PC that manages the axis movements. The script cyclically actuates the axis to follow the reference profile and asks the microcontroller to start the acquisition of the accelerometer data. The received features are then stored in a file, and the process is repeated for each profile. The sampling frequency of the microcontroller is 5kHz, for a total of 6000 samples per profile.

Although not useful for the training, the microcontroller has been set not only to transmit the features to the PC but also the time-series, for visualization purposes. The time-series of the training set are shown in **figure 2.28**, and the features are shown in **figure 2.29**.

In the time-series set it is possible to see some outliers, for example, there is a record in which profile 1 started being actuated by the axis with a delay w.r.t. the

others. Profile 4, instead, has some outliers due to the axis sometimes overshooting the reference position.

The training set contains 100 snapshots for each profile, for a total of 400 snapshots. The K-means model is then trained for $n = 5$ clusters, according to the silhouette criterion.

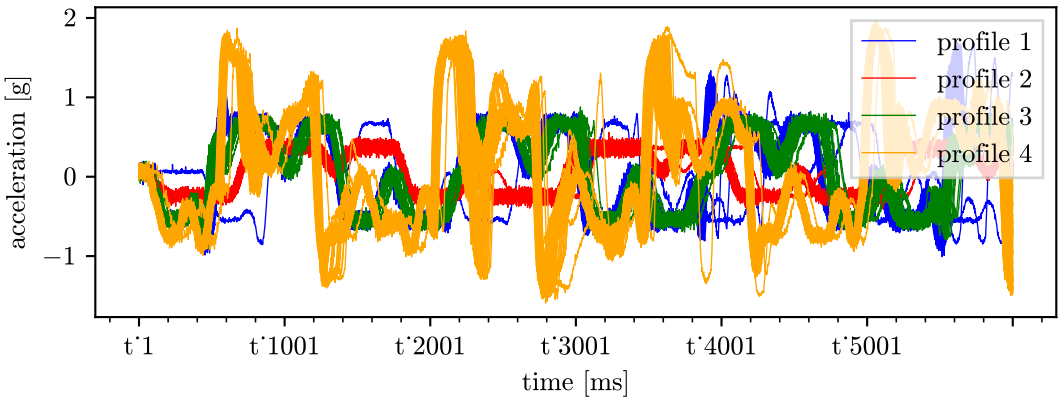

Figure 2.28: *Timeseries of the training set*

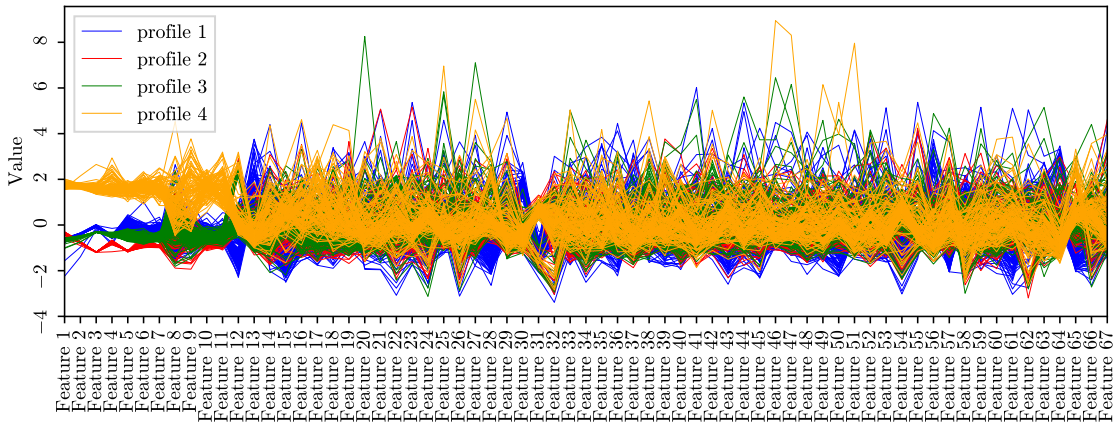

Figure 2.29: *features of the training set*

As done previously, the training is performed with the user confirming the correct number of clusters. And updating the model into the microcontroller.

2.7.2 Testing

The microcontroller is then set in *evaluate* mode and the ND is performed on a loop that repeats the movement of profile 2. The result of this first model (Model 1) is shown in **figure 2.30**. We can see that the model immediately falsely detects a novelty, despite profile 2 being part of the training set. The figure also shows a clearer view of the evolution of the novelty score, obtained by applying a moving average filter on the last 5 values of the novelty metric.

Let's investigate why this model gives almost all false positive results. Analyzing the features of the training set (**figure 2.29**), we can see that the first features, up to ≈ 12, are grouped by profile, but most of the remaining features are not, this arises the suspect that these features may not be significative of the movement, but maybe just representing noise. This is likely also because the necessary standardization procedure ensures that each feature will have unitary standard deviation, regardless of the magnitude of the feature itself w.r.t. the others. This may cause "noise" features to be amplified in the model. Moreover, it's clear that, being most of the features not significant, the model is not able to distinguish between the profiles.

2.7.3 Feature scaling

To address the problem of the non-significative features, a feature scaling method is proposed. The idea is to scale the features in such a way that the most significant features will have a higher weight in the model. To do that, a naive approach could be to visually select the important features (by eye it's evident that are the first few) and apply a small scaling factor to the others.

This approach goes against the principle of the framework being fully automatic to train. To address this problem, an unsupervised and automatic method is proposed. The idea is to scale the features in such a way that the most significant features will have a higher weight in the model. To do that, it's possible to exploit the fact that, at this point, the K-means model is already trained and this step provided labels for the training dataset. It's now possible to apply a *supervised* ML algorithm in a way that is transparent to the user, so the whole procedure remains *unsupervised*.

Luckily, the `sklearn` library provides a method

Figure 2.30: *Novelty detection on profile 2.*

Bibliography

[1] Fagor Automation. *IBARMIA - 5 AXES MACHINING CNC 8065*. Sept. 2017. URL: https://www.flickr.com/photos/fagorautomation/35762600461/ (cit. on p. 2).

[2] Victorchan An and William Meeker. «Estimation of Degradation-Based Reliability in Outdoor Environments». In: (Oct. 2001) (cit. on p. 4).

[3] Mingming Yan, Xingang Wang, Bingxiang Wang, Miaoxin Chang, and Isyaku Muhammad. «Bearing remaining useful life prediction using support vector machine and hybrid degradation tracking model». In: *ISA transactions* 98 (2020), pp. 471–482 (cit. on p. 4).

[4] Jean Jacquelin. *Régressions Et Équations Intégrales*. Première édition : 14 janvier 2009 - Mise à jour : 3 janvier 2014. 2009, pp. 16–18. URL: https://www.scribd.com/doc/14674814/Regressions-et-equations-integrales (cit. on p. 6).

[5] J. Lee, H. Qiu, G. Yu, J. Lin, and Rexnord Technical Services. «Bearing Data Set». In: *IMS, University of Cincinnati* (2007). Available at: https://www.nasa.gov/intelligent-systems-division/discovery-and-systems-health/pcoe/pcoe-data-set-repository/ (cit. on p. 23).

[6] Umberto Albertin, Giuseppe Pedone, Matilde Brossa, Giovanni Squillero, and Marcello Chiaberge. «A Real-Time Novelty Recognition Framework Based on Machine Learning for Fault Detection». In: *Algorithms* 16.2 (2023). ISSN: 1999-4893. DOI: 10.3390/a16020061. URL: https://www.mdpi.com/1999-4893/16/2/61 (cit. on p. 33).

[7] Hai Qiu, Jay Lee, Jing Lin, and Gang Yu. «Wavelet filter-based weak signature detection method and its application on rolling element bearing prognostics». In: *Journal of Sound and Vibration* 289.4 (2006), pp. 1066–1090. ISSN: 0022-460X. DOI: https://doi.org/10.1016/j.jsv.2005.03.007. URL: https://www.sciencedirect.com/science/article/pii/S0022460X0500221X (cit. on p. 34).