Project Number 723634

D2.4 Full Prototype of Predictive Analytics Platform

Version 1.0
15 February 2019
Final

Public Distribution

IKERLAN

Project Partners: **ATB, Electrolux, IKERLAN, OAS, ONA, The Open Group, University of York**

PROJECT PARTNER CONTACT INFORMATION

ATB Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de	**Electrolux Italia** Claudio Cenedese Corso Lino Zanussi 30 33080 Porcia Italy Tel: +39 0434 394907 E-mail: claudio.cenedese@electrolux.it
IKERLAN Trujillo Salvador P Jose Maria Arizmendiarrieta 20500 Mondragon Spain Tel: +34 943 712 400 E-mail: strujillo@ikerlan.es	**OAS** Karl Krone Caroline Herschel Strasse 1 28359 Bremen Germany Tel: +49 421 2206 0 E-mail: kkrone@oas.de
ONA Electroerosión Jose M. Ramos Eguzkitza, 1. Apdo 64 48200 Durango Spain Tel: +34 94 620 08 00 jramos@onaedm.com	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of York Leandro Soares Indrusiak Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325 570 E-mail: leandro.indrusiak@york.ac.uk	

DOCUMENT CONTROL

Version	Status	Date
0.2	Document outline and content guidelines	1 December 2018
0.4	Algorithm implementation details	25 January 2019
0.8	Updates and editing	1 February 2019
1.0	Final version	15 February 2019

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This document describes the Full Prototype of the software implementing the functionality as specified in D2.2, demonstrating basic functionality of the Predictive Analytics Platform. It includes a short description of the functionalities covered by the early prototype and their integration into the SAFIRE infrastructure.

This deliverable is an evolution from D2.3 (Early Prototype of Predictive Analytics Platform), including now the work performed to develop the full prototype. The main additions are:

- Detailed description of work performed for the Business Cases.
- Full description of the used technologies.
- Description of Scalability and High Availability solutions.
- Requirements coverage table and statistics have been updated.

1. INTRODUCTION

1.1 OVERVIEW

This document depicts the Full Prototype Analytics Platform based on:

- the first results from D1.1, Business Cases Requirements and Analysis,
- the results from D1.4, the SAFIRE Concept,
- the specification of Predictive Analytics Platform and
- the methodology for Predictive Analytics Platform

1.2 APPROACH APPLIED

For each of the main modules forming SAFIRE, a similar approach where a first step is to analyse the requirements collected at Business Case requirements and analysis phase, detail these and from there derive the data model, functional specification, external interfaces, and technical specification.

The general approach followed to write the current document can be seen in Figure 1.

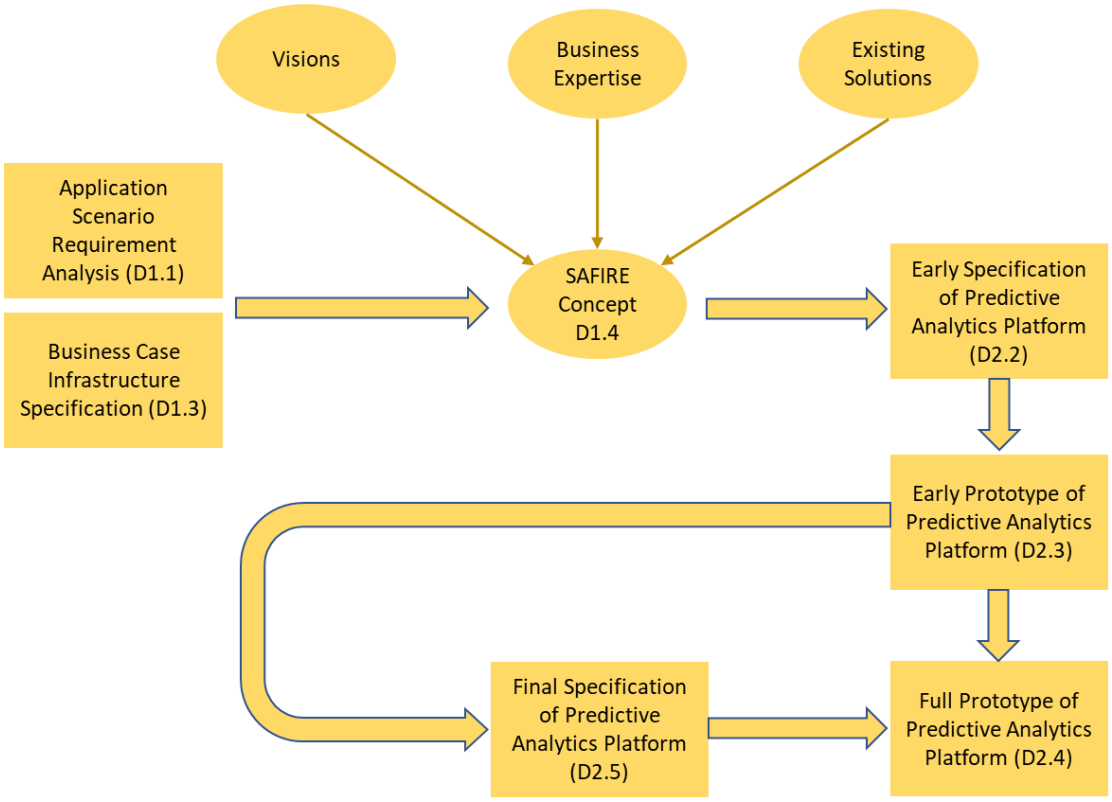


Figure 1: Approach followed for Full Prototype of Predictive Analytics Platform

1.3 DOCUMENT STRUCTURE

The document consists of:

- Section 1. Introduction, which describes the purpose of the document, and provides a brief overview of its contents.
- Section 2. Description of the Full Prototype (FP) implementation of the Predictive Analytics Platform.
- Section 3. Briefly describes the integration with other modules.
- Section 4. Short description on how to install and configure the Predictive Analytics Platform.
- Section 5. Describes the specific customisation for the SAFIRE business cases.
- Section 6. Presents the Software tools used for implementation
- Section 7. Conclusions and wrap up of the deliverable

2. PREDICTIVE ANALYTICS PLATFORM

The Predictive Analytics Platform allows to the SAFIRE users to do advanced analytics in real time, storing huge amounts of data and web visualization tools to easily query and visualize the stored data. Moreover, the Predictive Analytics Platform offers different web services for interacting with different modules. An architectural overview of the envisioned platform can be seen on Figure 2.

The Data Ingestion module is capable of keeping its current state in case the connection to the data source is lost, and retrying the connection after a reasonable time has passed without any loss of data. If the connection cannot be restored, a human operator is notified to take action.

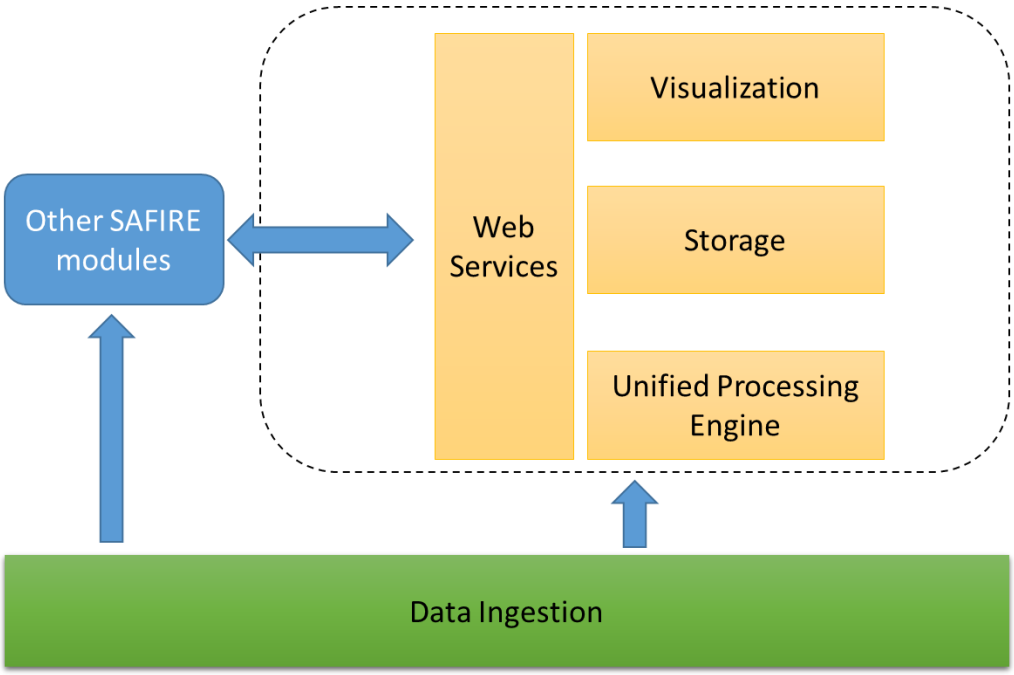


Figure 2: Conceptual Predictive Analytics Platform architecture

The modules of the platform are described next.

2.1 UNIFIED PROCESSING ENGINE

The Unified Processing Engine provides support for doing advanced analytics on both real-time and batch approaches. This module is based on Apache Spark. A Unified Big Data Framework. Moreover, as defining complex real-time analytics is difficult right now with that kind of frameworks a Complex Event Processing (CEP) engine has been included on the platform covering this use case. The CEP engine used on SAFIRE is called Espertech. Espertech, provides to the developer with Domain Specific Language (DSL) language based on SQL that helps to define complex real time analytics patterns.

For the cases when Apache Spark is used for real-time scenarios, a custom UI interface with a REST API for monitoring the different defined streaming queries has been developed. This UI interface can be seen on the Figure 3.

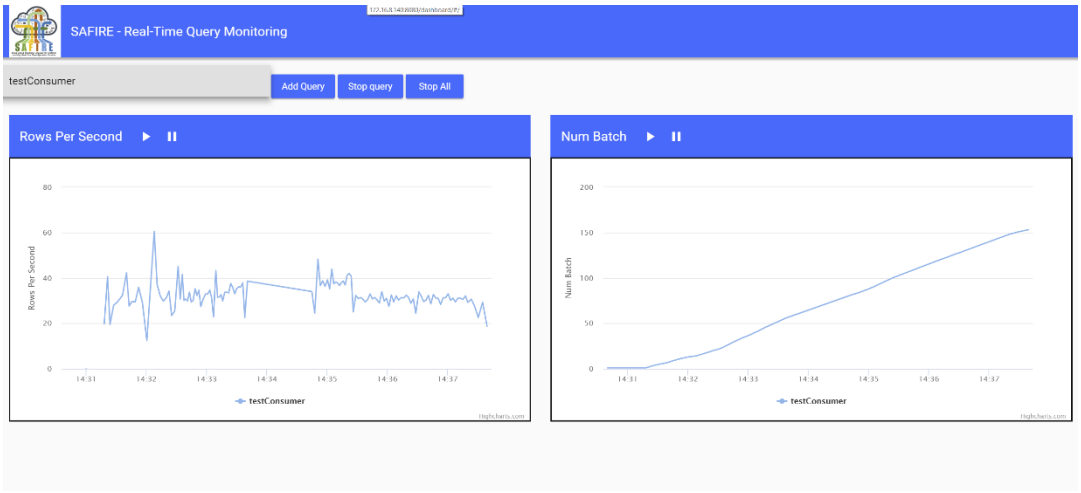


Figure 3 - Spark UI module for real-time metrics

2.2 STORAGE

The relational storage is covered using PostgreSQL, a leading Open Source Relational Database System. The relational database performs several of the data quality checks listed in D2.5 Final Specification of Predictive Analytics Platform. The No-SQL storage is covered by using Apache Cassandra Database, a key-value based database with horizontal scalability properties and with great integration with the Big Data landscape via different connectors. For each of the use cases of the platform, where to store the data and what to store where (relational data or non-relational data) must be decided.

2.3 VISUALIZATION

In order to be able to visualize the results of the analytics two tools are provided within SAFIRE Predictive Analytics Platform for different kind of users:

1. **Business Intelligence**: in order to provide support for Business intelligence dashboards Apache Superset tool is provided. provides an easy way to define different web-based dashboards with a lot of connectors to multiple databases.

2. **Data Scientist**: in order to provide support for data scientists that need to interact easily with the stored data in SAFIRE and need to execute interactive advanced analysis over huge quantities of data and visualize those analyses in an easy way, Apache Zeppelin is provided within SAFIRE to fulfil this task. Zeppelin is a web-based notebook that provides support for interactive analytics over Big Data easily.

2.4 SCALABILITY AND HIGH AVAILABILITY

High availability and scalability are supported through the use of a resource orchestrator such as Kubernetes (unified resource manager), Mesos or Yarn. Tools like Spark make use of this type of systems to be highly available. In addition, the deployed services (API) can use this type of managers to be able to be restarted in different nodes or with several instances, in order to distribute the load.

To achieve scalability, this type of managers has support to add and remove instances during execution in order to have more resources available. Regarding the availability of the algorithm, depending on the tool used and the algorithm itself, it will be possible to perform an approximation (raise more instances of the algorithm) or to parallelize it more (Spark MLLib has support for parallelizable algorithms).

2.5 IMPLEMENTED EP FUNCTIONALITIES

Some of the functionality already implemented (in the Early Prototype), has been refined in the Full Prototype. An overview of the functionality implemented, is listed in the following table using the requirements as a guide.

Table 1: Data Mining and Analytics Requirements

Req. No.	Requirement	Overall Priority	Status
U78	Supports data mining to extract useful patterns about operator behaviour	SHALL	Implemented
U79	Supports data mining to extract useful patterns about machine status	SHALL	Implemented
U80	Supports data mining to extract useful patterns about production process status	SHALL	Implemented
U81	Provides support for selection of sensors / systems to be analysed	SHALL	Implemented
U82	Provides support for selection of information sources to be analysed	SHALL	Implemented
U83	Provides support for data/sensor composition functionality	SHALL	Implemented
U84	Able to provide historical knowledge about system deviations or problems	SHOULD	Implemented

Req. No.	Requirement	Overall Priority	Status
U85	Able to provide decision support for production line selection	SHOULD	False
U86	Able to increase visibility of the production process	SHALL	False
U87	Supports analysis for algorithm definition for boiling/temperature control functionality	SHALL	True
U88	Supports sensitivity analysis to noise	SHALL	Implemented
U89	Supports main variation factor identification and robust strategy for minimising	SHOULD	False
U90	Supports computational resources estimation of machines	SHOULD	False
U91	Supports estimation of performance decrease for algorithm complexity reduction	SHOULD	False
U92	Supports process repeatability and stability characterisation	SHALL	False
U93	Supports Design of Experiments (DOE) and Analysis of Variance (ANOVA) analysis	SHOULD	Implemented

Table 2: Performance Requirements

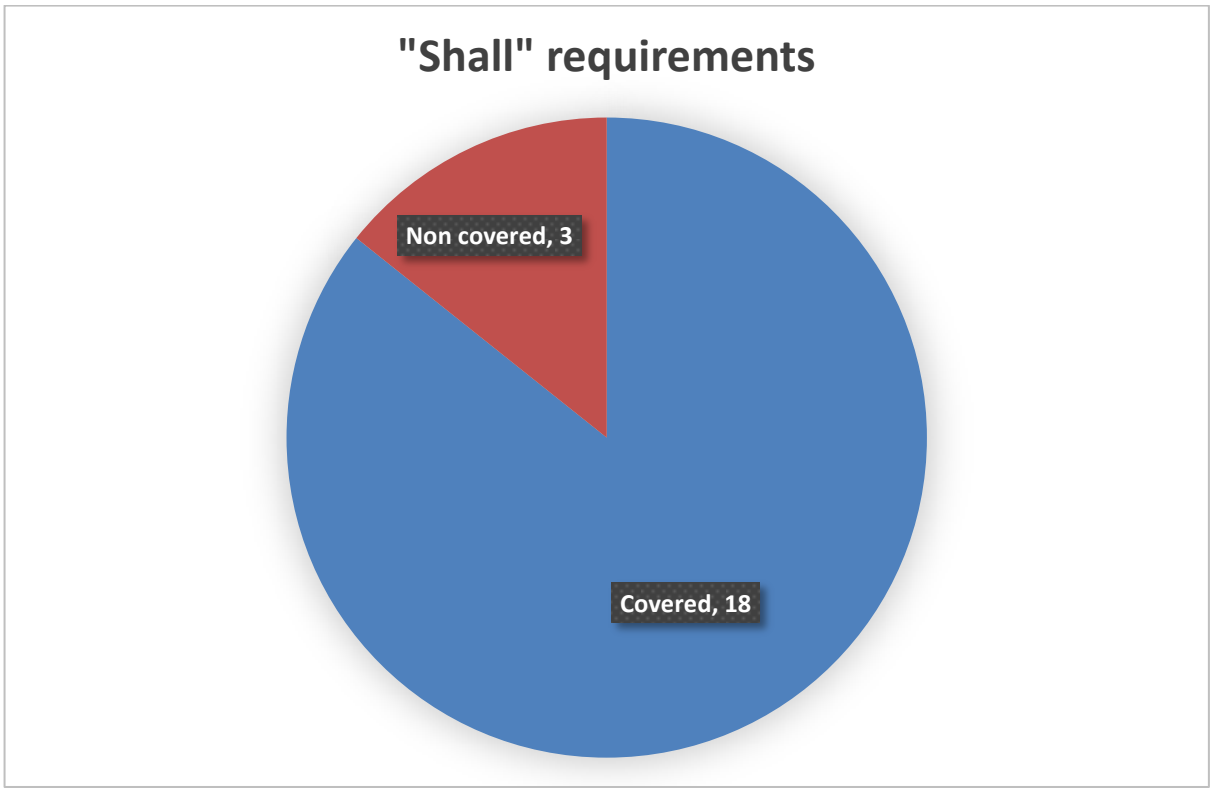
Req. No.	Requirement	Overall Priority	Status
U115	Does not negatively affect the usual production processes	SHALL	Implemented
U116	Support for scalability in the size of cloud and computing resources	SHALL	True
U117	Support for horizontal scalability to many machines	SHALL	True
U118	Capable of real-time data ingestion (registering data)	SHALL	Implemented
U119	Capable of batch processing of data (offline analysis)	SHALL	Implemented
U120	Capable of real-time data processing	SHALL	Implemented
U122	Able to analyse relevant data within a given timeframe	SHALL	Implemented, depending on the Analytics and the Computing Resources
U123	Capable of storing up to 5 TB/year/machine with resource recycling facilities	SHALL	Implemented
U124	Provides support for Machine Learning	SHALL	Implemented

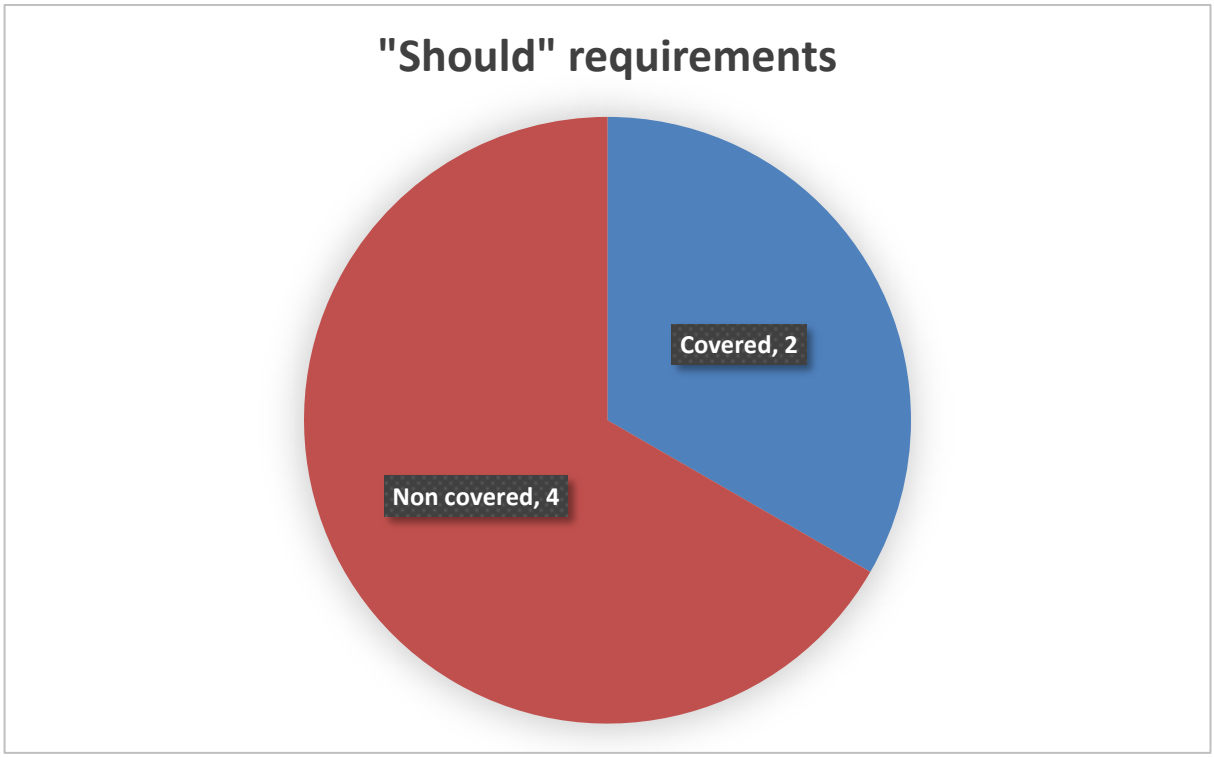
Req. No.	Requirement	Overall Priority	Status
	(Supervised / Unsupervised / Anomaly Detection)		
U125	Able to achieve required precision on cooking process estimation / optimisations	SHALL	Implemented

Table 3: Integration Requirements

Req. No.	Requirement	Overall Priority	Status
U130	Able to access data stored in a relational database	SHALL	Implemented

In the next charts a summary of the covered "shall" and "should" requirements is presented.





3. INTEGRATION WITH OTHER MODULES

The Full Prototype of the Predictive Analytics Platform is integrated with the following modules:

- **Data-Ingestion**: The data ingestion modules are NiFi templates that are in charge of sending data to SAFIRE Kafka cluster. The following data ingestion modules are available:
  - **OAS proNTo**: The data ingestion module in the OAS case connects to the Oracle database server of the proNTo system (simulated factory) and ingests the data required by SAFIRE modules into the Kafka cluster.
  - **ONA Cloud**: This data ingestion module in the ONA case connects to the REST API of ONA cloud and ingests the data required by SAFIRE modules into the Kafka cluster.
  - **ONA Machine**: This data ingestion module in the ONA case connects to the ONA machine using the XML based protocol, send it to the cloud via remote NiFi to NiFi connection and then, data required by SAFIRE modules is published into the Kafka cluster.
  - **Electrolux**: The data ingestion module in the Electrolux case connects to the data provided by the experimental cooker setup. Results are read from Matlab/CSV files and sent to the cloud using the MQTT IoT protocol for simulating a real scenario. Then, SAFIRE modules ingest data and send it to the Kafka cluster.

For the full prototype, integration with other SAFIRE modules will be developed using a Web service approach (e.g. asking for historical data stored in the Predictive Analytics Platform).

4. INSTALLATION, CONFIGURATION AND USAGE

This section describes the installation, configuration and usage of the Full Prototype of the Predictive Analytics Platform. The business case specific customisation is described in Section 5.

4.1 INSTALLATION

The Predictive Analytics Platform can be downloaded from

https://gitlab.atb-bremen.de/SAFIRE/safire-predictive-analytics

The steps to install will be defined in a file in that repository. Note that some steps can need a valid AWS account and can incur in AWS costs.

4.2 CONFIGURATION

As the Predictive Analytics platform has a huge list of Frameworks to configure. Each one with different possibilities, please refer to the official documentation of each framework where specific configuration is needed.

5. BUSINESS CASE SPECIFIC CUSTOMISATION

5.1 ELECTROLUX

5.1.1 Predictive Analytics

Electrolux Business Case has been tested by implementing the following two services:

- *Boiling status detection* of a pot without using direct physical sensors inside the pot.
- *Temperature estimation* of a pot without using direct physical sensors inside the pot.

The prediction service doesn't know nor the amount of water in the pot, neither the power applied to the pot, but only the currents in the coil and the temperature of the coil itself. The models have trained off-line but can be invoked in real-time (order of seconds) so that the detection (boiling or temperature) can be done in real-time.

The data flow needed for EP-support of the Electrolux business case in the Early Prototype was simulated with a custom simulator that was reading offline data from different Electrolux experiments (.csv files) and sending it to the standard communication for the Internet of Things MQTT.

From that point, a simple NiFi template that can be seen on Appendix 8.1.3 was used to ingest data for the SAFIRE Platform. An overview of can be seen on Figure 4.

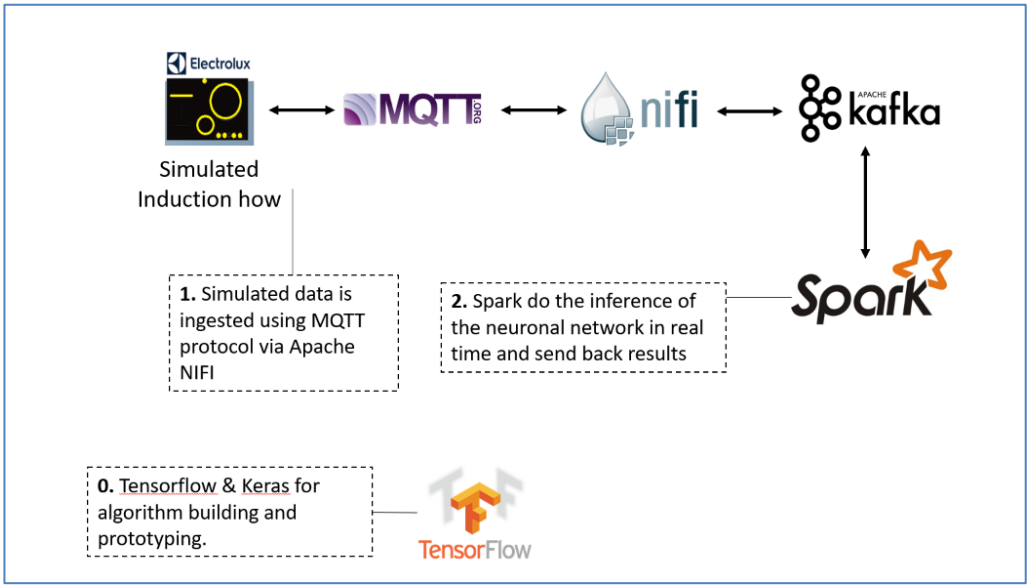


Figure 4 - Detailed Data Flow for the Electrolux scenario in the Early Prototype

For Final Prototype data is gathered from Matlab (c) that is connected to a cooking and sends the data to SAFIRE via MQTT, mosquito, NiFi and Kafka (and finally to Spark as previous case) as shown in Figure 5.

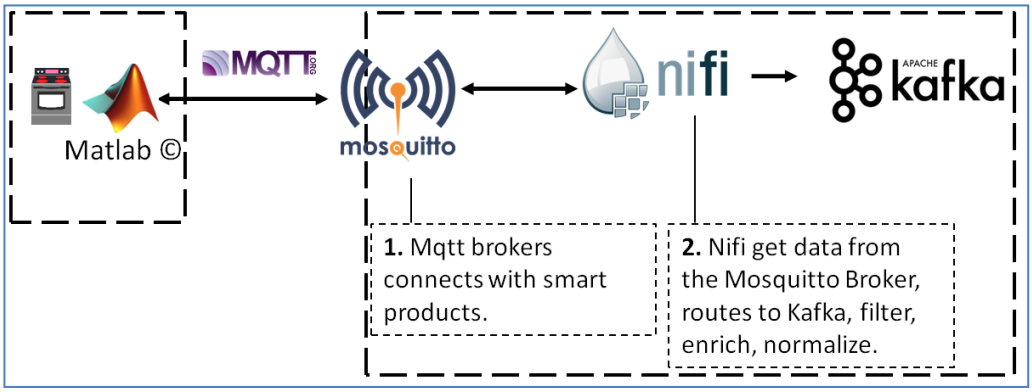


Figure 5 - Detailed Data Flow for the Electrolux scenario in the Final Prototype

The data model used for Electrolux as the experiments contains different timestamps with the data from all the currents of the induction hob. Moreover, each "row" also contains whether the water is boiling or not.

Prediction Execution Flow

Real time prediction is executed by a Predictive Analytics service in the cloud (developed in Java and deployed as a *.jar). When the cooking (or matlab(c)) invokes the service (via message into MQTT or directly invoking the service via REST Web Service), the service (1) loads a trained predictive model (spark model, keras model,

etc), (2) evaluates the input with the model and (3) sends back to the caller the prediction (via the same path as the invocation).

5.1.2 Boling Status Detection.

The goal is to be able to detect the boiling status of a pot without using direct physical sensors inside the pot. Some promising advantages of boiling detection include:

- Water boiling detection (even, customised to each person).
- Boil maintenance.
- Milk boiling.
- Oil boiling.
- Food cooking status estimation (i.e. spaghetti cooking status, French fries, etc).

From sate of the art it is known that the thermal status of a pot can be evaluated by indirectly measuring electrical parameters (such as currents in a coil). In the case of Electrolux, a patent pending fast sweep process allows multiple *Current vs. Frequency* measures at the same time with minimal interaction with the cooking process. This idea has the advantage of having more data (more currents) available for detection.

Error! Reference source not found. Figure 6 shows a plot with a boiling process. Graph on the left shows the temperature of the water (starting at 20º Celsius and reaching 100º) and graph of the right shows the plotting of six currents (in amperes). Typically, behaviour of the currents (after initial transition of about 60 seconds) is a *decreasing* phase, followed by a *plateau* (flat) phase, and followed by a very little *increasing* that indicates that the pot is boiling.

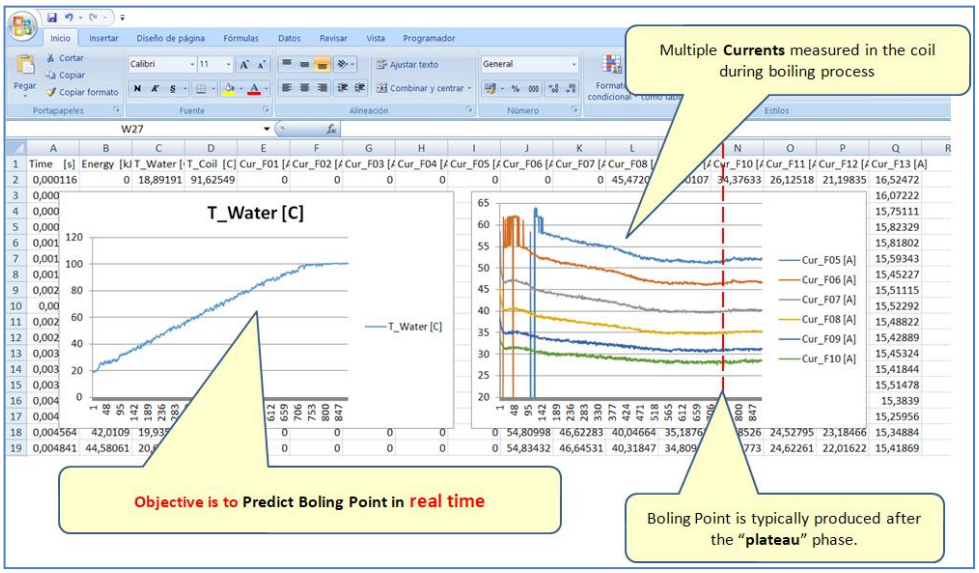


Figure 6 - Plot of currents during a boiling process

Sometimes, as in Figure 7 current shapes resulting from a boiling process show a clear pattern of *decrease-plateau-little increase* shape. Therefore, boiling point identification is *relatively* easy.

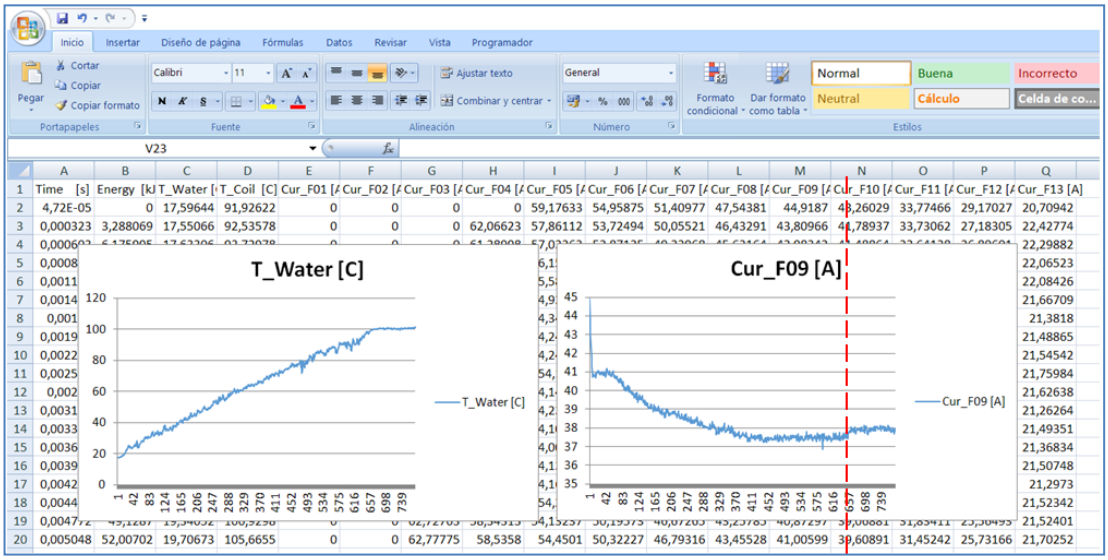


Figure 7: Boiling process with currents pattern easily showing the boiling point.

Unfortunately sometimes, as in Figure 8Figure 8Figure 7, current shapes resulting from a boiling process don't show a clear pattern of *decrease-plateau-little increase* shape. In Figure 8Figure 8 it is not easy to recognize the end of the *plateau* phase. In this case, boiling point identification is *quite* difficult.

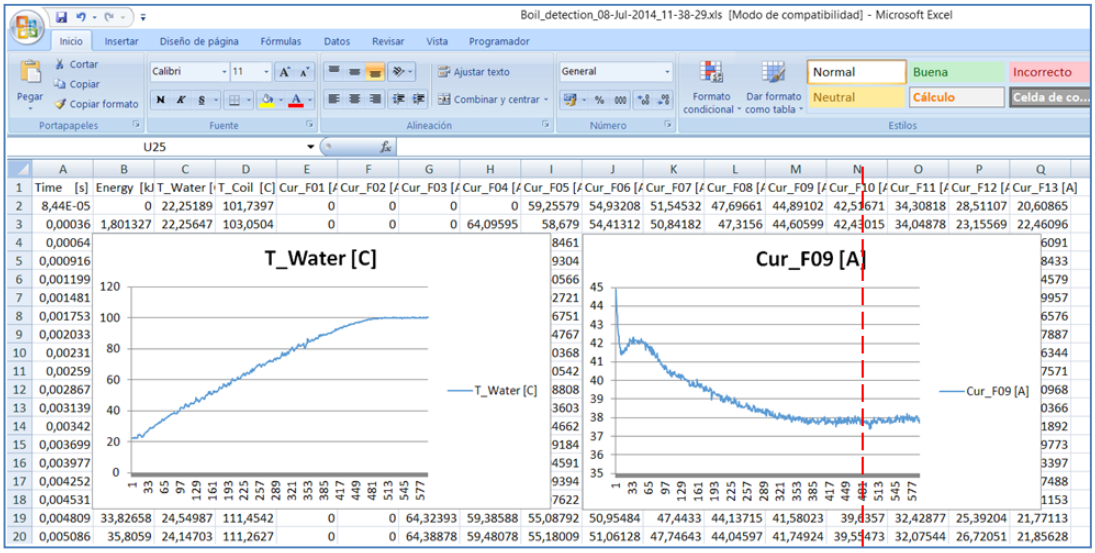


Figure 8: Boiling process with current pattern with unclear boiling point.

Having multiple currents alleviates the problem described in Figure 8Figure 8. A close look at the figure reveals that, although current F08, F09, F10 don't show a clear boiling point, currents F05, F06, F07 shows a much clearer boiling point. Thus, the use of multiple currents may be an advantage.

Predictive Analytics has been used in this BC as follows:

- ***Training Phase*** – A neural network has been used with the help of 144 boiling experiments carried out by Electrolux. For each experiment, Electrolux has monitored, second by second, the value (in Amperes) of a set of currents. In addition to this, a temperature sensor in the pot has measured the temperature second by second. With this data, a machine learning algorithm, in this case a Neural Network, has been trained to learn to recognise the boiling point of the pot:
    - 96 of the 144 experiments (66.6%) have been used as training set.
    - 48 of the 144 experiments (33.3%) have been used as development set.
- ***Testing Phase –*** Later, the neural network has been tested. For each experiment, the testing has been done as follows:
    - The neural network receives the currents second by second (simulating a real time boiling process).
    - Second by second, the neural network, with the current data received so far, decides if the pot is boiling or not.

Next sections show details of the neural networks training processes and the results achieved.

5.1.2.1 Neural Network architecture and training

Sample generation for training

A neural network consists of a stacked number of layers composed by neurons. The network usually has one input layer (of fixed size), some hidden layers, and an output layer. The key point here is that the input layer is a *fixed size* layer and therefore, receives a current pattern of a fix number of points.

As it has been mentioned earlier, the network receives the currents second by second. This means that network is first activated after the first *valid* 100 seconds (the very first 90 seconds are discarded as they are very noisy, so, from that point on, the values are considered *valid*), but also, this means that after 100 seconds, the network has to deal with an increasingly larger collection of values (representing all the values of a given current received so far). Just an example, after 4 *valid* minutes of starting its job, the network has 4x60=240 values (per current).

However, as mentioned at the beginning of this section, a network has a fixed input size and cannot handle variable length inputs. To solve this problem, input signals are down

sampled to 100 points. In other words, each time a new value is received (second by second), the whole signal received so far is down sampled to 100 points.

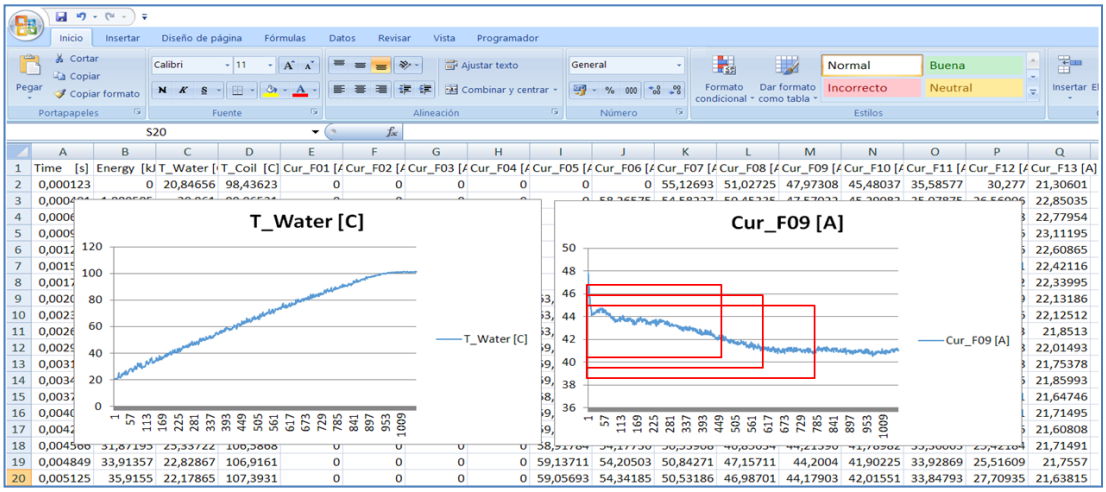


Figure 9: Sample generation

In Figure 9Figure 9 (graph on the right) red squares represents time windows. As time passes, the whole signal (down sampled to 100 values) is passed to the network.

In addition to down sampling the signal received so far, it is convenient to eliminate the noise of the signal. Some methods have been tried but finally, a down sampling and a noise elimination is carried by a performing *a polynomial regression* (degree 3) of all points received so far and. Then, the regressed signal is re-sampled in equally spaced 100 points. After down sampling, the values are normalized (mean 0, variance 1.0).

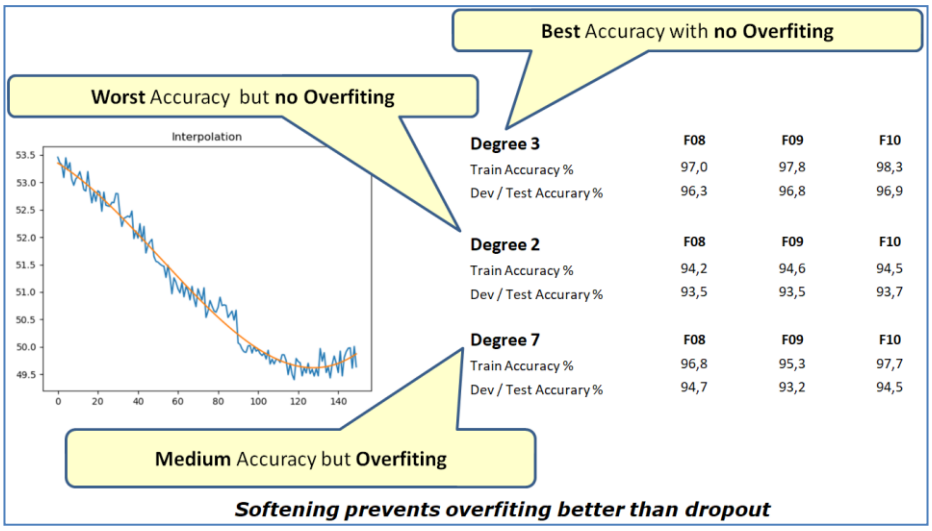


Figure 10: Samples softened and resized to 100 points with polynomial regression of different degrees

Several polynomial degrees have been tried and finally degree 3 has been found the best. In addition to eliminating the noise, polynomial interpolation helps the neural network to generalise better, avoiding overfitting.

With 144 experiments, this process generates a total of about 45.000 samples that are labelled 0 (not boiling, in this case water temp < 98.0) or 1 (boiling, water tem >= 98.0). These samples are given to the networks for training.

Networks architectures experimented

Several architectures have been tested so far:

- **Fully Connected** (Multilayer Perceptron) - a simple but powerful network in which all neurons of a layer are connected to all neurons of next layer.
- **LSTM** (Long Short Term Memory) - networks with memory from past values.
- **Convolutional** NN - network which first layers consist of signal processing convolutional filters.
- **LSTM + Convolutional** - combination of LSTM and Convolutional.

In addition to those architectures, two alternatives have been tried:

- Training **one single** network receiving **one single** current (usually F09 that is, normally a very clean signal).
- Training **one single** network receiving **multiple currents** (in this case, currents F08, F09, F10, as they are usually the cleanest signals).
- Training **three single independent** networks receiving each one just **one single current** (F08, F09, F10, respectively) and using a kind of **voting** mechanism to decide if the pot is boiling (all the three must agree in that the pot is boiling).

The third option (**three networks with a voting mechanism**) has been found more accurate and robust. Figure 11Figure 11 shows training accuracy for all the three networks (one per current). Accuracy represents the number of samples that are correctly labelled.

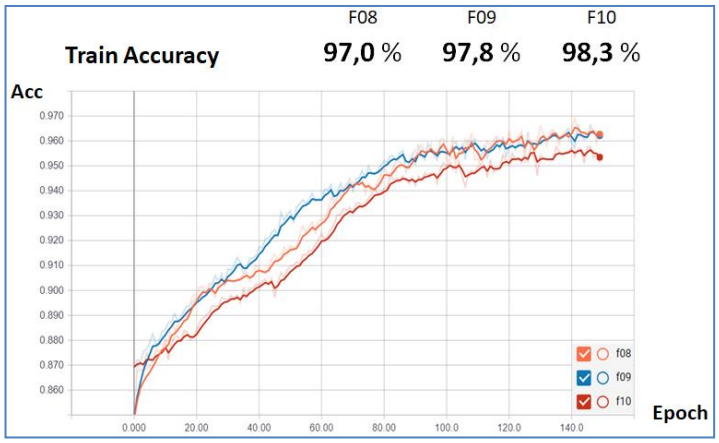


Figure 11: Training accuracy of the three neural networks (one network per current)

Testing procedure

After training with samples coming from 66.6% of the experiments, validation and test procedure consists of evaluating the accuracy of the network with the remaining samples, it is, 33.3% of the total available samples. To simulate a real situation, signal values are collected second by second. Each second, the signal received so far is processed (down sampled, regressed to 100 points and normalized) and passed to the network to predict boiling. Accuracy represents the number of samples that are correctly labelled.

Figure 12Figure 12 shows accuracy of validation and test sets as the number of training epochs increases (in our case, validation and test sets are the same, as there are few samples).

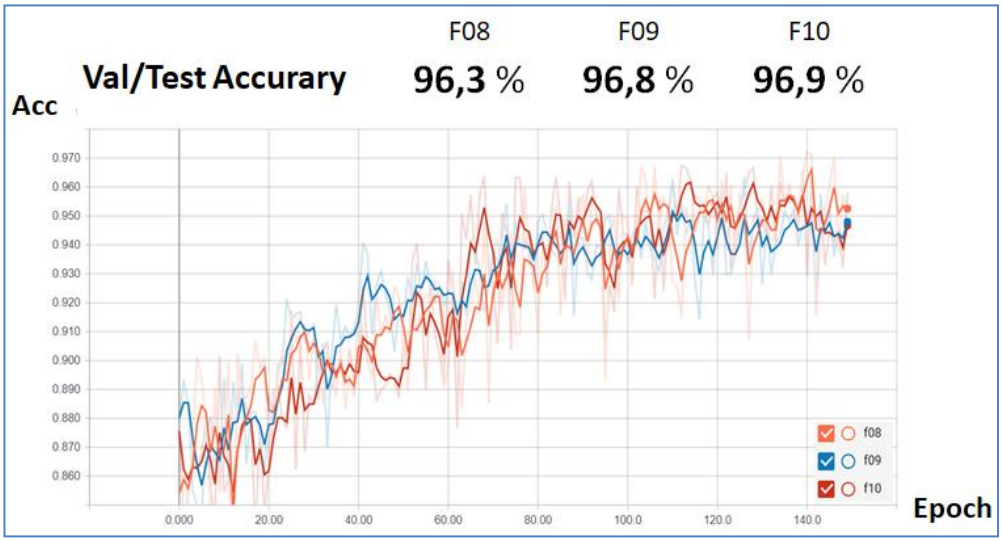


Figure 12: Validation/Test sets accuracy of the three neural networks (one network per current)

5.1.2.2 Results: Accuracy of Detection in Time

Accuracy in Time

More important than accuracy of results in terms of % of samples correctly labelled is the accuracy of boiling detection in time, in other words, the error of detection (in seconds) with respect to the *exact* boiling time.

Results of experiment carried so far show the following results:

- Training **three single network** receiving **each one just one single current**:
    - Three currents F08, F09, F10 respectively.
    - Fully Connected Neural Network architectures.
    - Voting mechanism (all three NN must classify sample as boiling).
    - Mean time error in validation/test data **12.2**s.
    - Mean temperature difference in validation/test **0.84**ºC.
    - One case in validation/test data does not identify boiling point.
    - Three cases in validation/test data with large errors (55s to 91s).
    - Excluding those latter three cases, mean error in validation/test is **8.3**s.

- Training **one single** network receiving **one single** current:
    - Cleanest current used (F09):
    - Fully Connected Neural Network architecture.
    - Mean time error in validation/test **18.82**s.
    - Mean temperature difference in validation/test **1.22**ºC.
    - One case in validation/test data does not identify boiling point.
    - Five cases in validation/test data with large errors (65s to 45s).
    - Excluding those latter five cases mean error in validation/test is **14.59**s.
- Training **one single** network receiving **three** currents at once:

- These are the worst results, probably due to the fact that there are too few samples for such a complex network (a network processing three currents at the same time becomes cumbersome).

Best network architecture

As mentioned before several architectures were tried but finally the simplest has been found to be the best (it is simple but, at the same time, has full potential of behaving as some of the others architectures). Best architecture so far was:

- X = **Dense**(100, input_dim=200, activation='relu')(X_input) – This one layer with 100 input neurons that are fully connected to the next layer.
- X = **Dense**(40, activation='relu')(X) – second layer has 40 neurons that are fully connected to the next layer.
- X = **Dense**(20, activation='relu')(X) – third layer has 20 neurons that are fully connected to the next layer.
- X = **Dense**(10, activation='relu')(X) - fourth layer has 10 neurons that are fully connected to the next layer.
- X = **Dense**(1, activation='**sigmoid**')(X) – final layer has one single neuron with a sigmoid activation function. Its output is interpreted as a probability.

Conclusions

As a conclusion the better results are achieved by using multiple currents and training three independent networks (trained separately each one with one current) and implementing a voting mechanism (the pot is boiling when all the three networks classify the sample as boiling).

It was expected that using more currents would improve detection accuracy, but it has been discovered that training three independent networks and using a voting mechanism, is a more *robust* solution because sometimes it has been observed that one network is predicting that the pot is boiling but the other two don't. This happens when one of the currents has some random noise that may confuse the network. However, as there are two more networks, the chance of having a noise that confuses all the three networks at the same time is less probable, resulting into a more robust detection algorithm.

Regarding the solution of one "big" network processing all signal currents at the same time, it has been found that for such a complex network, many more samples would be needed for training, so results are not conclusive (in our case, with the samples available, the worst of all three alternatives).

5.1.3 Temperature Estimation

The goal is to be able to estimate the temperature status of the water in the pot without using direct physical sensors inside the pot.

In the previous test case (boiling) from state of the art it is known that the thermal status of a pot can be evaluated by indirectly measuring electrical parameters (such as currents in a coil). However, in the case of temperature estimation, to our knowledge, there are no previous experiences of temperature estimation out from currents' profiles and, in the case of Electrolux, from coil temperature change profile.

Error! Reference source not found. Figure 13 shows a plot with a boiling process. Graph on the left shows the temperature of the coil (starting at about 50º Celsius and reaching more than 160º) and graph of the right shows the plotting of six currents (in amperes). In previous test case, behaviour of the currents is known (*decreasing* phase, *plateau*, very little *increasing*). In addition to this information, the network is fed with the coil temperature (the profile shown in the graph is very typical).

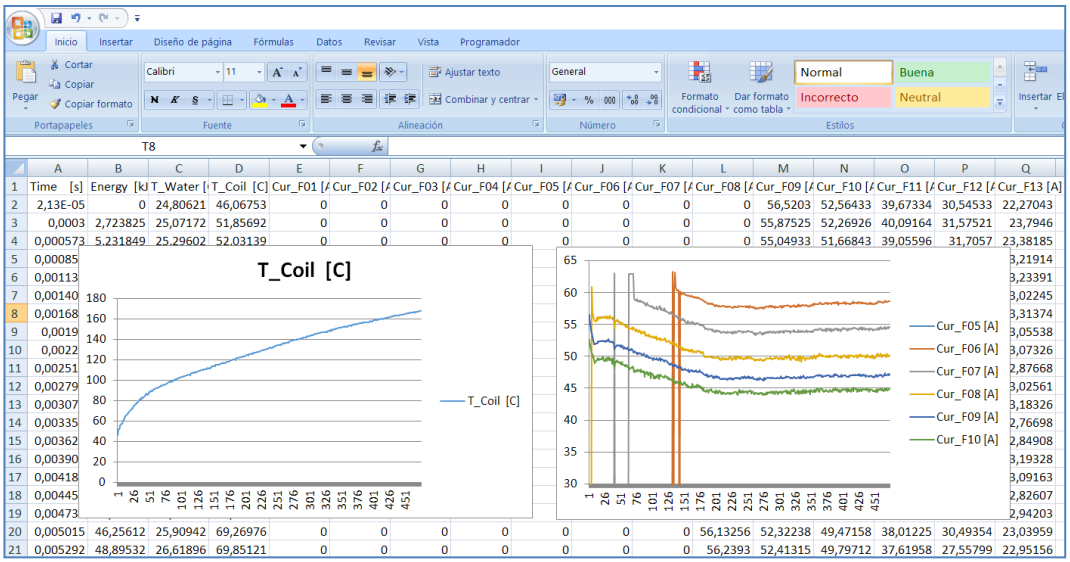


Figure 13: Plot of coil temperature and currents during a boiling process.

The temperature of the coil has been analysed in the samples and it has been found that the temperature increase follows always a similar pattern, an initial phase with rapid temperature increase followed with an increase that decays with the time. This temperature profile, along with the currents profile, helps the network to estimate the temperature.

Predictive Analytics has been used in this BC as follows:

- ***Training Phase*** – A neural network has been used with the help of 144 boiling experiments carried out by Electrolux. For each experiment, Electrolux has monitored, second by second, the value (in Amperes) of a set of currents. In addition to this, a temperature sensor in the pot has measured the temperature second by second. With this data, a machine learning algorithm, in this case a Neural Network, has been trained to learn to estimate the temperature of the water in the pot:
    - 96 of the 144 experiments (66.6%) have been used for training.
    - 48 of the 144 experiments (33.3%) have been used for testing.
- ***Testing Phase –*** Later, the neural network has been tested. For each experiment, the testing has been done as follows:
    - The neural network receives, second by second, both the temperature of the coil and the currents.
    - The neural network estimates, second by second, the temperature of the water in the pot.

Next sections show details of the neural networks training processes and the results achieved.

5.1.3.1 Neural Network architecture and training

Sample generation for training

Sample generation is the same as in the case of boiling estimation but (a) samples are labelled with the temperature of the water and (b) samples are enriched with samples of the coil temperature, as shown in Figure 14.

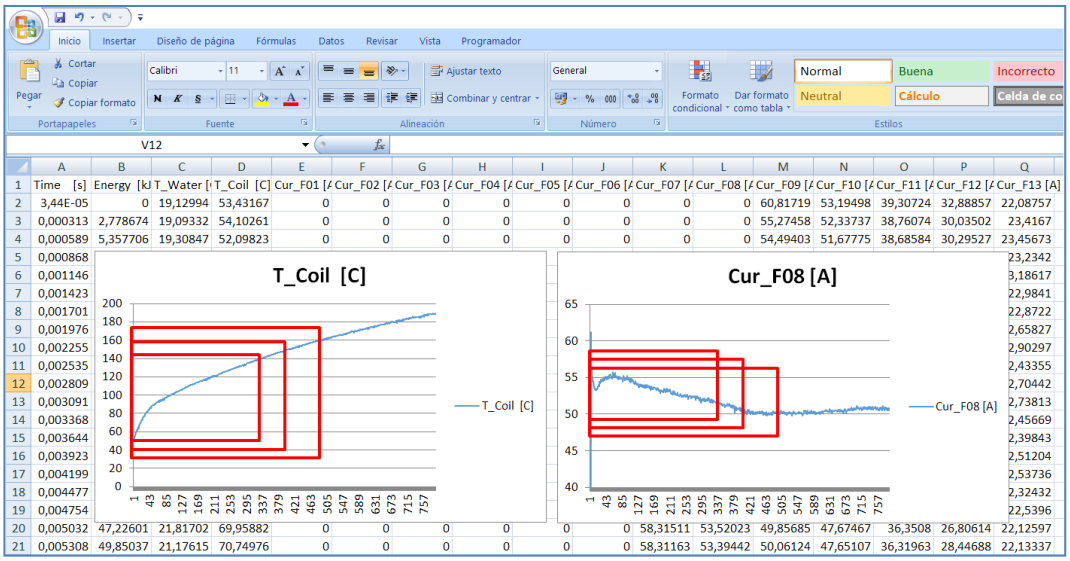


Figure 14 – Plots of coil temperature and currents signal

As in the case of boiling estimation, a network has a fixed input size and cannot handle variable length inputs. To solve this problem, all input signals are down sampled to a fix number of points.

As the problem of temperature estimation is more difficult than boiling detection, samples are down sampled to 200 points (instead of 100 points). After down sampling and smoothing, the values are normalized (mean 0, variance 1.0). In addition to down sampling, to eliminate the noise (as in the case of boiling estimation) the signal is smoothed with *a polynomial regression* (degree 3).

However, coil temperature is not filtered as it is a very clean signal with very low noise. Nevertheless, the signal values are normalized between 0.0 and 1.0 before passing to the network.

Networks architectures experimented

Several architectures have been tested so far:

- **Fully Connected** (Multilayer Perceptron) - a simple but powerful network in which all neurons of a layer are connected to all neurons of next layer.
- **GRU** (Gate Recurrent Units) - networks with memory from past values.

Following the experience gained with previous test case, boiling estimation, the approach taken is training **three single independent** networks receiving each one just **one single current** (F08, F09, F10, respectively) and the **coil temperature** and taken the medium of the three temperatures estimated by the three networks.

Testing procedure

After training with samples coming from 66.6% of the experiments, validation and test procedure consists of evaluating the accuracy of the network with the remaining samples, it is, 33.3% of the total available samples. In this case, accuracy represents the medium square deviation from the actual temperature of samples and the temperature predicted by the network.

5.1.3.2 Results: Accuracy of temperature estimation

Accuracy in temperature

Accuracy of results is given as a medium square deviation of the actual temperature (labels of the samples) and the temperature predicted. Best results obtained for the three single independent networks are the following:

- **Network for F08 + Coil**
    - *Trained Samples*
        - *Mean Square Deviation:* ***0,0027***
        - *Mean Absolute Error:* ***3,8*** *Cº*
    - *Test Samples*
        - *Mean Square Deviation:* ***0,0049***
        - *Mean Absolute Error:****4,89*** *Cº*
- **Network for F09 + Coil**
    - *Trained Samples*
        - *Mean Square Deviation:* ***0,0025***
        - *Mean Absolute Error:* ***3,64*** *Cº*
    - *Test Samples*
        - *Mean Square Deviation:* ***0,0047***
        - *Mean Absolute Error:* ***4,72*** *Cº*
- **Network for F10 + Coil**
    - *Trained Samples*
        - *Mean Square Deviation:* ***0,0024***
        - *Mean Absolute Error:* ***3,63*** *Cº*

- *Test Samples*
  - *Mean Square Deviation:* ***0,0049***
  - *Mean Absolute Error:* ***4,84*** *C°*

The following Figure 15, Figure 16, Figure 17, Figure 18 and Figure 19 show a variety of examples of temperature estimation for ***Test Samples***, it is, ***non-trained samples***. **Orange** line represents the actual temperature of water while **blue** line represents the temperature estimated by the network (mean of three networks' estimations). Scale Y axis of the graphs are labelled with 5º steps.

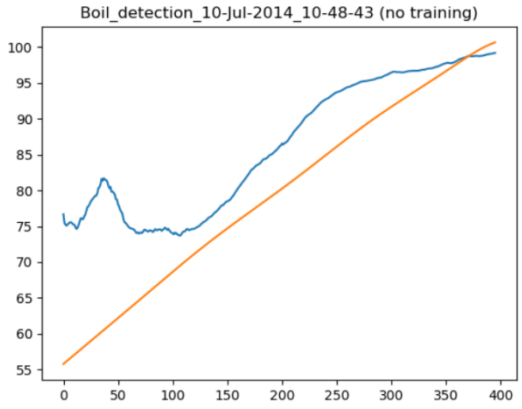


Figure 15 – Temperature estimation for case 10-jul-2014 10:48:43

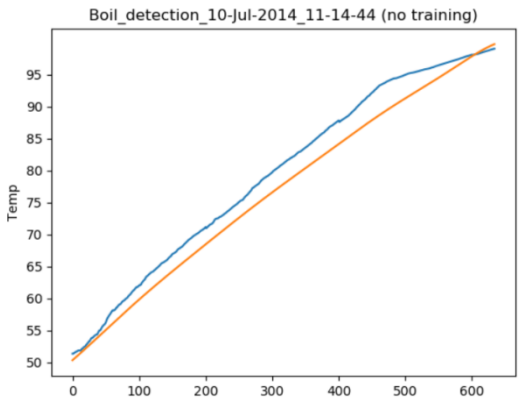


Figure 16 - Temperature estimation for case 10-jul-2014 11:44:44

Figure 17 - Temperature estimation for case 10-jul-2014 12:35:40

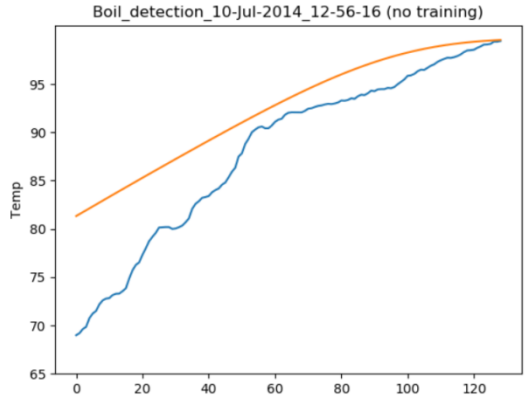


Figure 18 - Temperature estimation for case 10-jul-2014 12:56:16

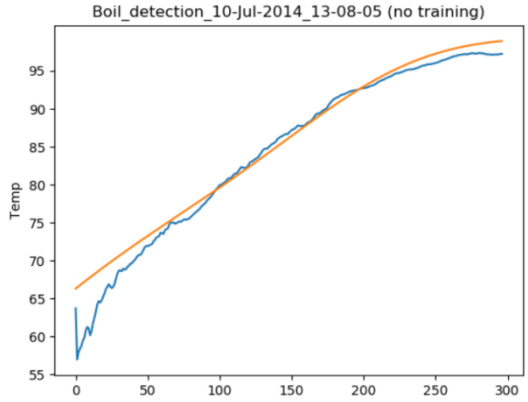


Figure 19 - Temperature estimation for case 10-jul-2014 13:08:05

It is worth noting that the temperature estimation error at the end is less pronounced than at the beginning. This is to be expected as at the end of the boiling process more signal is available to the network, therefore, more accuracy is to be expected.

Best network architecture

As mentioned before several architectures were tried but, so far, an architecture composed by two subnetworks (one to process currents and another to process coil temperature), followed by a subnetwork that combines both subnetworks, has been found to be the best. Best architecture so far was:

Subnetwork to process **Signals Currents**

- X_input = Input(shape = X_input_shape)
- X = **Dense**(50, input_dim=200, activation='relu')(X_input) – Layer with 50 neurons fully connected to the next layer-
- X = **Dropout**(0,2)(X) – Dropout to avoid overfitting
- X = **Dense**(20, input_dim=200, activation='relu')(X) – Layer with 20 neurons fully connected to the next layer-
- X = **Dropout**(0,2)(X) – Dropout to avoid overfitting
- X = **Dense**(4, activation='relu')(X)

Subnetwork to process **Coil Temperature**

- C_input = Input(shape = C_input_shape)
- C = **Dense**(50, input_dim=200, activation='relu')(C_input) – Layer with 50 neurons fully connected to the next layer-
- C = **Dropout**(0,2)(C) – Dropout to avoid overfitting
- C = **Dense**(20, input_dim=200, activation='relu')(C_input) – Layer with 20 neurons fully connected to the next layer-
- C = **Dropout**(0,2)(C) – Dropout to avoid overfitting
- C = **Dense**(4, activation='relu')(C)

Subnetwork to combine and process both subnetworks

- XC = **Concatenate**()([X, C])
- XC = **Dense**(8, activation='relu')(XC)
- XC = **Dense**(4, activation='relu')(XC)
- XC = **Dense**(2, activation='relu')(XC)
- XC = **Dense**(1, activation='linear')(XC)

Note that final network has just one neuron that outputs the estimated temperature.

5.1.4 Conclusions

As a conclusion the better results are achieved by using multiple currents and coil temperature values. Network architecture composed by two subnetworks (one to process currents and another to process coil temperature) followed by a subnetwork that combines both subnetworks.

Temperature accuracy, in the mean case, is less than 5º C, which is a good result, particularly because at the end of the boiling process (when accuracy is needed), the accuracy is even better.

5.2 ONA

The ONA ingests data provided the ONA Industrial Cloud (OIC). The OIC is provided by SAVVY Data Systems and the data contained in it can be accessed using a REST API. The ingestion module uses this API to access the data.

The available data consists of metadata and the actual machinery data. The metadata can be divided into the following types.

- **Location**: The location data includes the different actual physical locations available. For each location, the unique location ID, name of the enterprise, name of the location, location coordinates, and time zone.

- **Machine**: Machine data refers to the individual machines that are being monitored. The available data consists of the unique machine ID, its name, whether it is active or not, the ID of the physical location (the same ID as in the location metadata), and the timestamp of last received information.

- **Group**: This type of metadata relates to the capture groups of indicators. For each group the unique ID, group name, description, collection frequency, data size, whether it is active or not, the timestamp of last modification date, and the machine and location IDs of the machine a particular capture group belongs to.

- **Indicator**: This metadata includes information of each of the monitored variables. For each variable the unique ID, variable name, description, origin, whether it is active or not, and the IDs of the group, machine and location they belong to are provided. If the indicator also has a minimal, maximal, and optimal values, these are also provided.

On the other hand, the actual machinery indicator data can be consumed in two ways: by means of individual requests, or through a stream of data. Individual requests have to include the start timestamp and end timestamp, and the response will only contain information generated in that timeframe. However, each request has a maximal timeframe and repeatedly requesting data while moving the timeframe is discouraged. If continuous monitoring of the indicators is desired, the streaming data should be used. In this case, a single request is made for a specific machine (or a list of machines) and the

server will respond with the latest available data. The client should read the response and keep the connection open for further reading, as the server will keep dumping new data into the connection until it is disconnected. Thus, the ONA Monitor reads indicator data through the data stream.

The Early Prototype of the ONA monitor reads data from the REST API and processes it using Apache NiFi. The NiFi dataflow then persists this data into a PostgreSQL relational database and also published on Apache Kafka for further distribution.

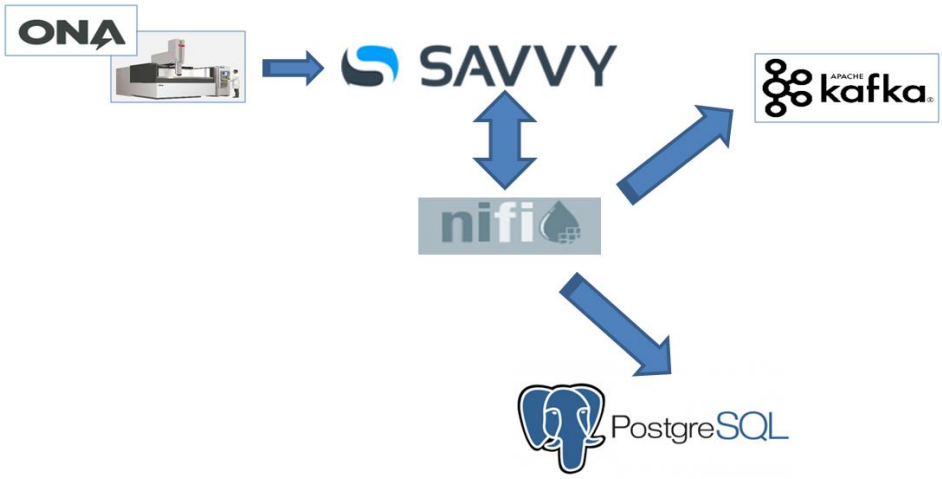


Figure 20 - Data transfer from the REST API into the ONA Monitor.

The metadata dataflow, depicted in Figure 21, ingests metadata from the REST API. Each type of metadata is provided by a different endpoint and is acquired in the following way. First, the system polls for available locations, and the response indicates the IDs for the locations the user has permission to read from. For each available location metadata is requested, and then the list of available machines on that location is requested. Then, each machine is polled, the machine metadata is stored, and also the capture groups for that machine are requested. Then, for each capture group, the group metadata is requested, followed by a request of the indicators belonging to that group. Finally, for each indicator, the metadata is requested. The server responds using the JSON objects as shown on **Error! Reference source not found.**. All this data is stored n the PostgreSQL relational database with a structure based on the objects returned by the server. Figure 21 shows the data model for the metadata. Metadata information ingestion is done periodically to check for new information, but not too frequently to comply with the API's anti-abuse policy.

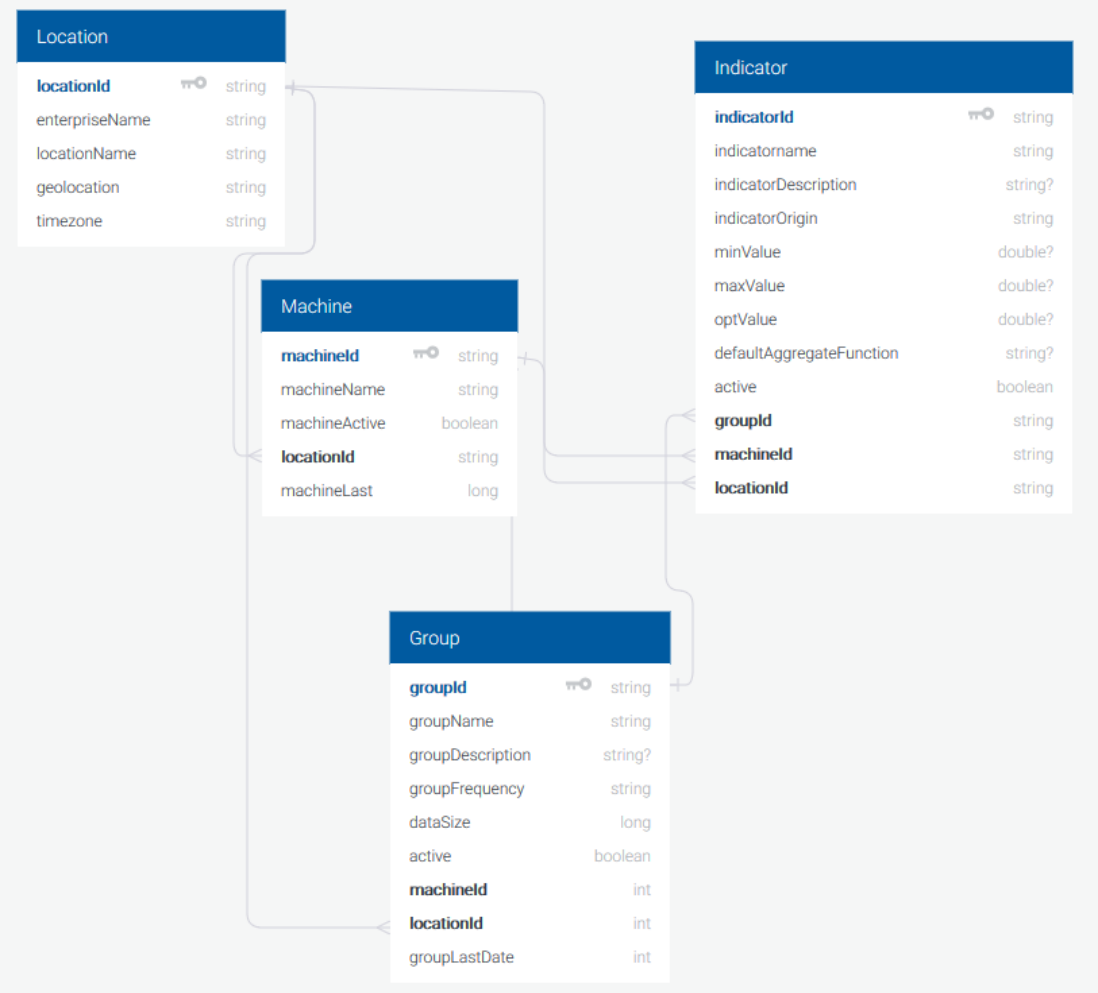


Figure 21 - Metadata data model.

```
{
        "locationId": "E1L1",
        "enterpriseName": "Empresa de demo",
        "locationName": "Taller",
        "geolocation": "43.301227464575554,-2.0148110389709473",
        "timezone": "Europe/Madrid"
}
```

Figure 22 - Example of REST API response for location metadata.

On the other hand, the streaming dataflow works in a different way. It only makes one request each time it is executed but keeps the connection open to keep receiving data, and then the data is processed much more thoroughly. The dataflow has four main parts:

- The API client group. Shown in Figure 35 (Appendix).
- The Stream Splitter group. Shown in Figure 36 (Appendix).

- The PostgreSQL insertion group. Shown in Figure 37 (Appendix).
- The Kafka redistribution group. Shown in Figure 38 (Appendix).

The API client group is the interface with the REST API. This group contains processors that prepare and execute the connection with the API. First the authorization parameters are calculated and set. Then, the ONAStreamProcessor, a custom processor developed for this project, opens a connection to the API and keeps it open to read new data as the server sends it. It works by not only triggering the next node in the dataflow when data is received and transferred, but also makes sure that the processor calls itself to be ready when new data arrives.

Each time a data package is received, it only regards a single machine and capture group, but can contain data of many indicators.

The stream splitter group makes some slight processing of the data, but only the modifications necessary to both insert it into PostgreSQL and republishing into Kafka.

The PostgreSQL insertion group first splits the data so that each flowfile of the dataflow only includes a single indicator and its value. Then, the flowfiles are filtered so that only the files containing the variables we are interested in continue in the process, and the rest are dropped. This filtering benefits the overall system as it frees resources from processing and storing unnecessary data. Then the data is modified so that it ends up with a format conforming to a JSON object that contains all necessary data (machineId, locationId, indicatorId, value, and timestamp) to convert it into an SQL query and insert it into the database.

The Kafka redistribution group publishes the data without altering its structure. This means that, unlike in the PostgreSQL insertion group, data is sent in a message in a similar structure as it is received, and not split until each flowfile contains a single indicator. However, the flowfiles do undergo some processing, in order to drop flowfiles with no real data, and normalize indicator and machine names. Before republishing the data into Kafka, the JSON objects are serialized into the avro binary format, as this format is more efficient for digital transfer and storage. The schema for the avro records is shown on Figure 23.

```
{
	"type": "record",
	"name": "onaRecord",
	"fields": [{
			"name": "machine",
			"type": "string"
		}, {
			"name": "group",
			"type": "string"
		}, {
			"name": "timestamp",
			"type": "string"
		}, {
			"name": "data",
			"type": {
				"type": "array",
				"items": {
					"type": "record",
					"name": "data",
					"fields": [{
							"name": "indicator",
							"type": "string"
						}, {
							"name": "value",
							"type": "string"
						}
					]
				}
			}
		}
	]
}
```

Figure 23 - Avro Schema for Kafka messages.

Another developed approach for obtaining data from ONA machines can be seen on Appendix 8.1.2.

5.2.1 Complex Event Processing

The Complex Event Processing (CEP) engine is in charge of processing the raw data and creating complex indicators based on individual events received by the engine, and acting upon them. It has been implemented using the Espertech engine.

The CEP engine ingests data from a Kafka broker, and persists the generated complex indicators in a PostgreSQL database. Afterwards, a data visualization framework, Apache Superset, feeds off of the persisted data to display the complex indicators for human consumption through a web interface. Figure 24 shows the flow of the data from

ingestion to visualization. The CEP engine itself is wrapped within a Java program. This program ingests the data from the broker, feeds it to the Espertech engine and subscribes to the output of the Espertech engine, forwarding its results to the database. The rules for the engine are described in the Event Processing Language (EPL) language, a SQL-like Doamin Specific Language (DSL) developed for Espertech.

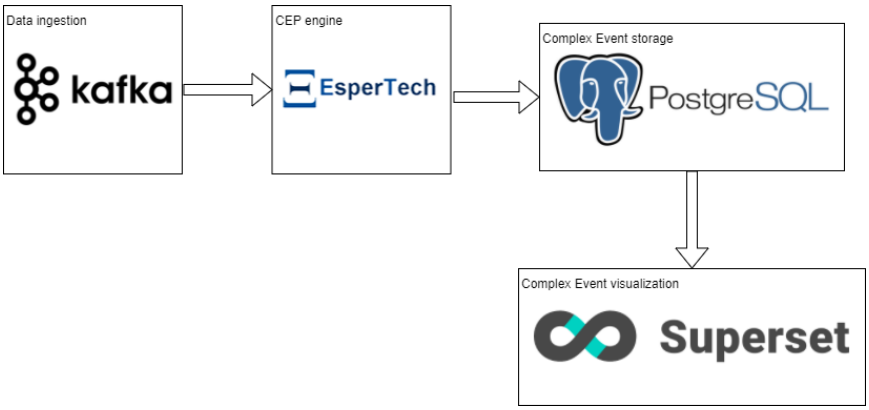


Figure 24 – Complex Event Processing flow

The events received by the CEP engine contain four pieces of information:

- Timestamp
- Machine ID
- Indicator ID
- Indicator value

From the same message stream events related to different machines are received, in chronological order. One of the first steps performed by the CEP engine is to pivot the received events by their machine ID and the (approximate) timestamp. Thus, it is the CEP engine's job to turn simple events like:

Timestamp	Machine ID	Indicator ID	Indicator Value
2018/12/01 10:00:00.245	1	A	84
2018/12/01 10:00:00.255	1	C	43
2018/12/01 10:00:00.305	2	B	21
2018/12/01 10:00:00.551	2	A	86
2018/12/01 10:00:00.601	1	B	0
2018/12/01 10:00:00.817	2	C	100

Into complex events like:

Timestamp	Machine ID	A	B	C
2018/12/01 10:00:00	1	84	0	43
2018/12/01 10:00:00	2	86	21	100

The CEP engine only monitors certain variables provided by the SAVVY API. The list of the monitored variables is the following:

- Ambient Temperature (ambientTemperature, measured by location, independent of the machine)
- Conductivity (conductivity)
- Speed (speed)
- Remaining Spool Percent (spoolRemainingWirePercent)
- Remaining Spool Length (spoolRemainingWireLength)
- Spool Type (spoolType)

- Theoretical Max. Speed (technological_theoreticalSpeed)
- Current Wire Thickness (thickness)
- Wire Speed (wireSpeed)
- Wire Type (wireType)
- Wire Diameter (wireDiameter)
- Red Semaphore: Flag 1 (semRojoFlag1)
- Red Semaphore: Flag 2 (semRojoFlag2)
- Red Semaphore: Flag 3 (semRojoFlag3)
- Red Semaphore: Flag 4 (semRojoFlag4)
- Amber Semaphore: Flag 1 (semAmbarFlag1)
- Amber Semaphore: Flag 2 (semAmbarFlag2)
- Amber Semaphore: Flag 3 (semAmbarFlag3)
- Amber Semaphore: Flag 4 (semAmbarFlag4)
- Amber Semaphore: Flag 5 (semAmbarFlag5)
- Green/Gray Semaphore: Condition 2 (semVerdeCond2)
- Grey Semaphore: Flag 1 (semGrisVerdeCond1)
- Grey Semaphore: Flag 1 (semGrisCond2)
- Grey Semaphore: Flag 1 (semGrisCond3)

The CEP engine keeps the state of each monitored machine in the current timestamp. Even though the event timestamp has a millisecond-level granularity, the timestamp of the state is kept at second-level. This gives the system some flexibility, and, since data from a single machine is received, at most, once per second there is no possibility of data overlap within the same second.

The state of a machine includes all of the monitored variables. When a new event is received, the engine checks whether the timestamp of the current state for that machine and the timestamp of the event coincide. If they do, the value of the indicator is registered in the state. If the timestamps do not match, the old state is published (even if values for all indicators were not registered) and the current state of the machine is reset.

Once the state is published, the computing of complex indicators begins. This is done in several steps, because some complex indicators require other complex indicators being computed in an earlier stage.

The produced warnings are:

- Whether the machine is running or not
- Whether the conductivity has been above a pre-established threshold or not
- Whether the conductivity has been below a pre-established threshold or not
- Whether the remaining spool percent has been above a pre-established percentage threshold or not
- Whether the change in temperature in the previous hour was above a pre-established threshold or not
- Whether the change in temperature in the previous 24 hours was above a pre-established threshold or not

The rules created to obtain these warnings are listed in Appendix 9.2.

These warnings are visualized in a Superset dashboard by means of chronograms. Figure 25 shows a Superset dashboard used in SAFIRE using chronograms. In these chronograms the X axis represents time. Each bar along the Y axis shows relates to a different warning and the colour or the bar in a period of time indicates for how much of that period the warning was active, with a darker colour representing more time with the active warning.

Superset dashboards are set-up using a mix of its web interface and database-dependent sentences. For relational databases, such as in this case, the used database-dependent sequences are written in SQL. The superset dashboard has some parametrizable options such as the machine (or machines) to visualise and the time range (by default the last 24h).

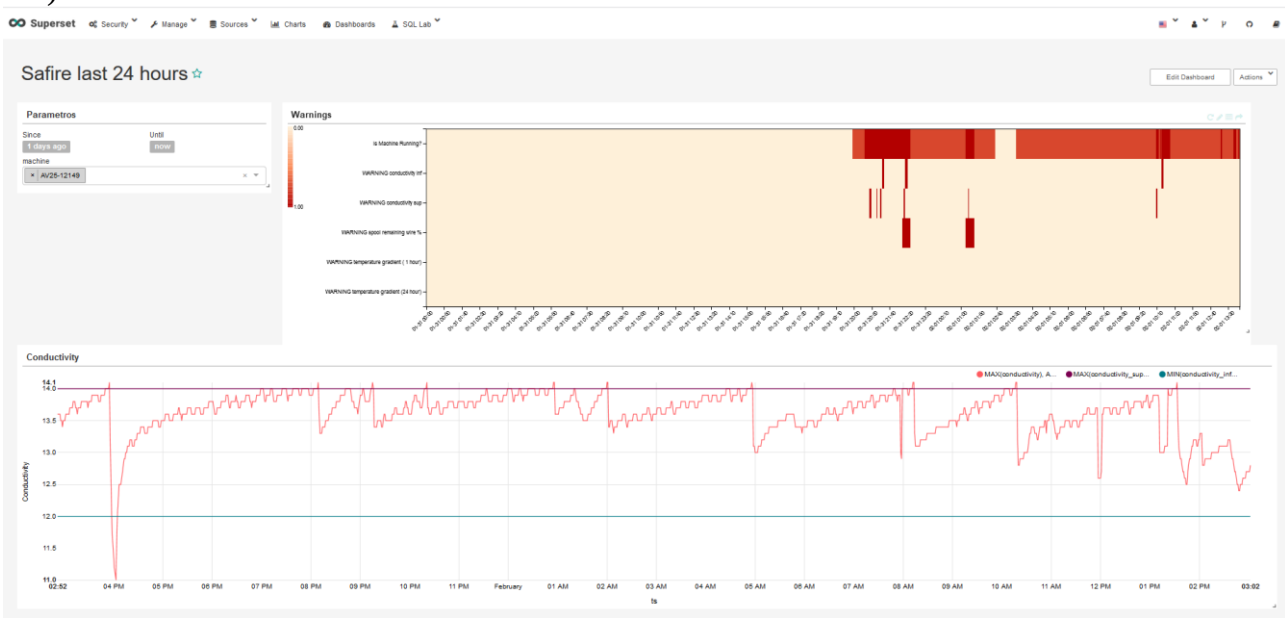


Figure 25 – Chronograms in Superset

5.2.2 Predictive Analytics

In the context of Wire Electrical Discharge Machining (WEDM), one the goals of this BC application is to be able to predict, in advance, the event of change of thickness of the machined part. The ability able to know in advance that a change of thickness is coming helps improving the cutting process.

WEDN process works by generating short electrical discharges between the cutting wire of the machine and the part to be machined through a dielectric fluid (deionised water).

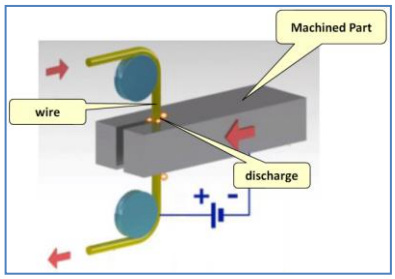


Figure 26: WEDM cutting process

Each discharge generates a little crater of few micrometres in the part, thus, drawing the shape of the part. Figure 27Figure 27 shows the voltage profile of several discharges.

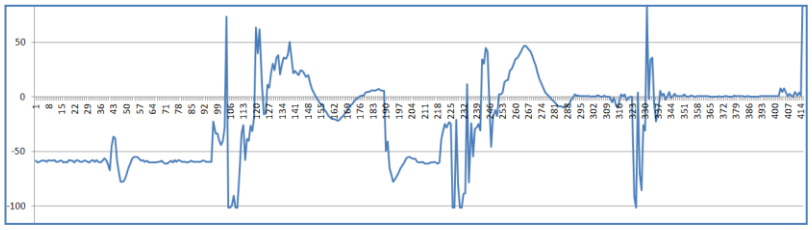


Figure 27: Voltage profile of several discharges

During the process of cutting a part, as the wire approaches a change in the thickness of the part, the discharge pattern changes (because the pressure of the dielectric fluid is lost or changes).

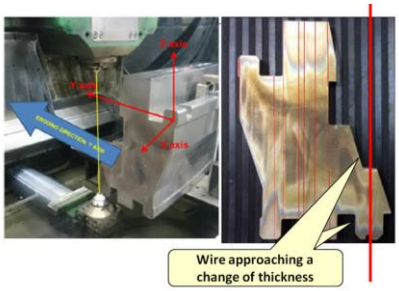


Figure 28: WEDM cutting with changing thickness parts

When the thickness of the part to be machined changes, to avoid degrading the cutting process, some process parameters need to be adapted. It would be very useful to be able to detect in advance this kind of events as the machine is cutting.

The key observation to develop such a detection system is that some features of the discharge voltage pattern may indicate that a change of thickness is approaching.

5.2.3 Part's Thickness Estimation

5.2.3.1 Sample generation for training

A number of experiments have been conducted by Ona registering the discharge voltage with an oscilloscope with a resolution of 100ns. During the experiments, five different zones (each 1 mm width) have been defined as shown in Figure 29Figure 29: Zone 1 is 5 mm away from the change of thickness point (these are optimal conditions), Zone 2 is 4 mm away, Zone 3 is 3 mm away, Zone 4 is 2 mm away and, finally Zone5 is just 1 mm away. The experiment design has followed a previous experience reported in [1] in which a neural networks approach is successfully applied to the case.

In each zone, a number of cutting experiments were conducted (0,8mm cutting of the total 1mm width of the zone, as the remaining 0,2mm are used to reset the oscilloscope). Each cutting experiment is divided in 2s sequences, registering the voltage with a sampling rate of 100ns, giving a total of 20.000 sample values per 2s sequence (voltage ranges from 120V to -120V). Each 2s sequence is labelled with a number 1 to 5 according to the zone in which the wire was cutting.

A total of 567 sequences have been recorded, each one of 2 ms (with 20.000 voltage values) and each sequence is labelled with the zone value (1 to 5).

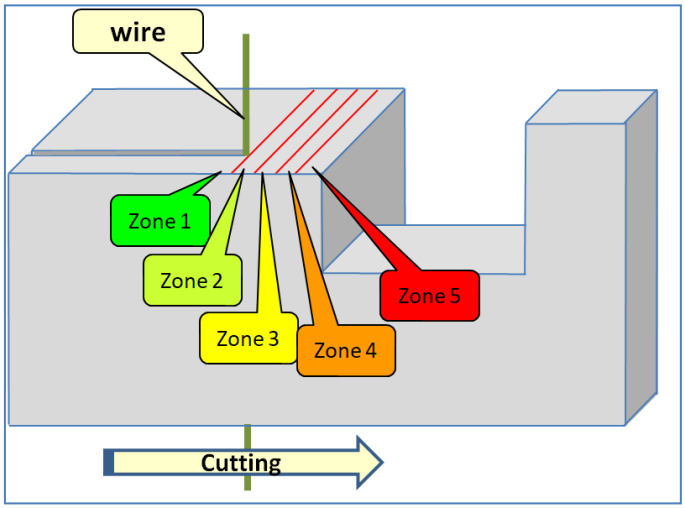


Figure 29: Zone definition during cutting experiments

Predictive Analytics has been used in this BC as follows:

- ***Training Phase*** – A variety of machine learning algorithms have been trained with 70% of the sequences (with a balance between zones).
- ***Testing Phase –*** Later, the algorithms have been tested with the remaining 30% of sequences.

5.2.3.2 Feature Extraction for each sample

For each sample, a set of features is extracted. The set of feature values for each sample (that will be used later to train machine learning algorithms) are the following:

- **Sparks per Second** category:
  - *Global Average Sparks/Sec* – Average number of sparks per second found in the sample.
  - *Td Low Average Sparks/Sec* – Average number of sparks/sec with low ignition delay time (td < 0.5 µs).
  - *Td Ok Average Sparks/Sec* – Average number of sparks/sec with ignition delay time ok (0.5 µs <= td <= 10 µs).
  - *Td High Average Sparks/Sec* – Average number of sparks/sec with high ignition delay time (10 µs < td).
- **Max Peak, Duration and Energy** category:
  - *Average Max Peak* – Average pulse voltage max peak.
  - *Average Duration* – Average pulse duration.
  - *Average raw Energy* – Average raw energy discharge by the pulse, computed as absolute area inside the peak.
- **Delay Time** category:
  - *Td Low Average Delay Time* – Average delay time of sparks with low ignition delay time (td < 0.5 µs).
  - *Td Ok Average Delay Time* – Average delay time of sparks with ignition delay time ok (0.5 µs <= td <= 10 µs).
  - *Td High Average Delay Time* – Average delay time of sparks with ignition high delay time (10 µs < td).
- **Ionization Phase Voltage** category:
  - *Td Low Average Ionization Phase Voltage* – Average voltage during ionization phase for sparks with low ignition delay time (td < 0.5 µs).
  - *Td Ok Average Ionization Phase Voltage* – Average voltage during ionization phase for sparks with ignition delay time ok (0.5 µs <= td <= 10 µs).
  - *Td High Average Ionization Phase Voltage* – Average voltage during ionization phase for sparks with high ignition delay time (10 µs < td).

Figure 30, Figure 31 and Figure 32 below show concepts involved in the features (ignition delay time, ionization phase, pulse duration, pulse max peak, etc). For a detailed description of these concepts see [2].

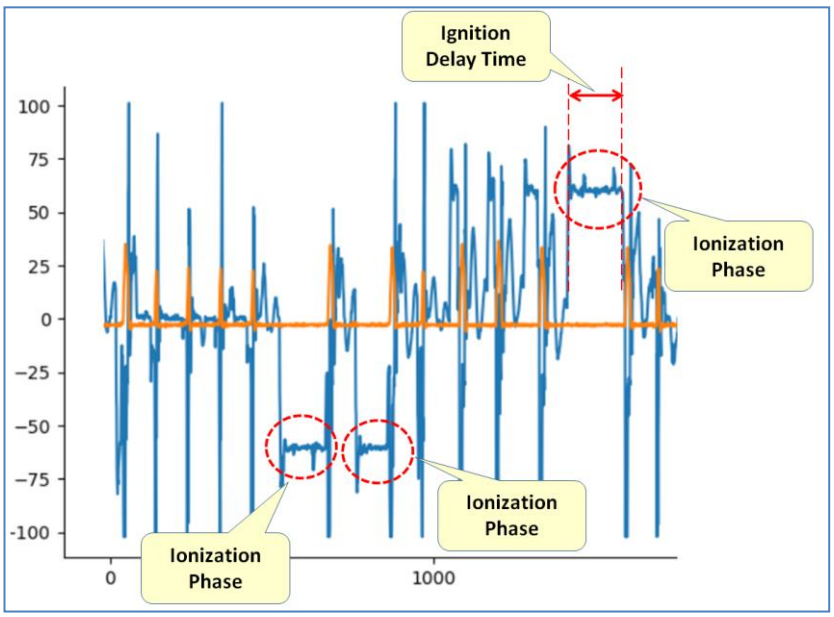


Figure 30 – Ignition delay time and ionization phase details

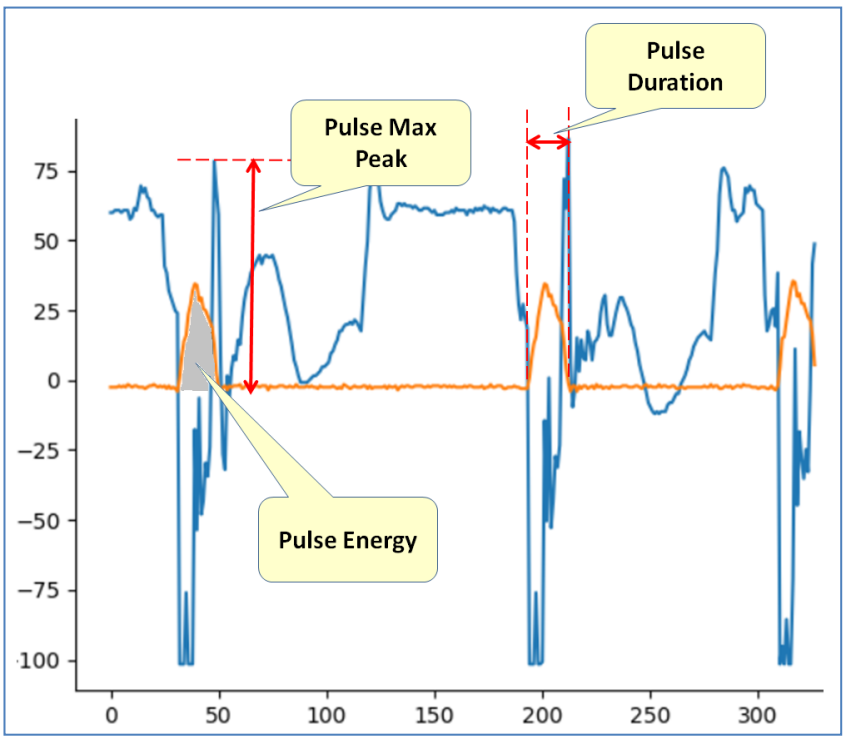


Figure 31 – Pulse Voltage max peak, Duration and Energy

Feature	Td	Zone 0	Zone 1	Zone 2	Zone 3	Zone 4	Zone 5	R. Pow 1/5	R. Pow 3/5
Avg Sparks/sec	all	20742.699	21102.610	21031.532	19653.333	15776.614	12857.018	1.641	1.529
	low	15242.699	15587.782	15483.671	14257.949	10928.883	8563.450	1.820	1.665
	medium	2132.931	2183.274	2177.365	2086.154	1725.131	1420.175	1.537	1.469
	high	3367.069	3331.554	3370.495	3309.231	3122.600	2873.392	1.159	1.152
Ratio Sparks/All	low	0.735	0.739	0.736	0.725	0.693	0.666	1.109	1.089
	medium	0.103	0.103	0.104	0.106	0.109	0.110	1.068	1.041
	high	0.162	0.158	0.160	0.168	0.198	0.223	1.416	1.327
Avg Current Pulse Max Peak A	all	1.107	1.058	1.056	1.066	1.094	1.127	1.065	1.057
Avg Current Pulse Duration µs	all	1.213	1.238	1.236	1.245	1.274	1.286	1.039	1.033
Avg Current Pulse Energy ∑ i*v	all	500.633	486.842	485.259	495.129	519.923	540.804	1.111	1.092
Avg Current Pulse Delay Time µs	low	0.660	0.677	0.662	0.671	0.653	0.630	1.076	1.066
	medium	7.161	7.170	7.158	7.178	7.156	7.234	1.009	1.008
	high	25.852	25.122	25.141	25.363	27.132	29.522	1.175	1.164
Avg Current Pulse Ionization Phase Voltage Avg V	low	10.825	11.099	10.823	10.903	10.663	10.251	1.083	1.064
	medium	53.303	53.081	53.030	53.071	53.144	53.221	1.003	1.003
	high	57.974	57.692	57.710	57.705	57.795	57.967	1.005	1.005

Figure 32 – Averages of Feature values for the different Zones

Signal Features directly served by Machine's Controller

ONA machine's controller is able to generate directly the features described above with a period of milliseconds. Therefore it is possible to add a local machine learning module in the machine that detects in real-time that a part-thickness change is approaching. The module can be installed in an industrial PC inside the machine getting data generated by the machine's controller.

5.2.3.3 Machine Learning Algorithms tested

Predictive Analytics has been used in this BC with the following Spark's machine learning algorithms:

- *Logistic Multi Class Regression*
- *Random Forest Trees*
- *Decision Trees*

Python PySpark Source code of the algorithms and their parameters can be found in Appendix 9.3.1.1.

5.2.3.4 Testing procedure

After training with 70% of the samples the test will be done with the remaining 30% of the samples. The machine learning trained models will try to classify the sample in one of the five zones and a confusion matrix will be computed to analyse the results.

5.2.3.5 *Results: Accuracy of Zone Detection*

Better results have been produced by *Logistic Multi Class Regression* classification. Accuracy achieved by each of the four algorithms can be found in Table 2. For samples in each zone type, the accuracy in being identified as correct or as other type is given.

	LogisticReg	***RandomForest***	***DecisionTree***
Class 1 Samples			
Classified as 5	*0.0 %*	*0.0 %*	*0.0 %*
Classified as 4	*10.4 %*	*6.6 %*	*8.3 %*
Classified as 3	*28.8 %*	*23.0 %*	*38.6 %*
Classified as 2	*56.0 %*	*41.0 %*	*37.1 %*
Classified as 1	*4.8 %*	*29.5 %*	*16.0 %*
Class 2 Samples			
Classified as 5	*0.0 %*	*0.0 %*	*0.0 %*
Classified as 4	*3.7 %*	*6.2 %*	*3.1 %*
Classified as 3	*27.4 %*	*31.0 %*	*37.5 %*
Classified as 2	*66.7 %*	*40.7 %*	*40.6 %*
Classified as 1	*2.2 %*	*22.1 %*	*18.7 %*
Class 3 Samples			
Classified as 5	*2.1 %*	*1.2 %*	*0.7 %*
Classified as 4	*20.3 %*	*19.4 %*	*13.8 %*
Classified as 3	*46.2 %*	*39.4 %*	*57.2 %*
Classified as 2	*29.4 %*	*25.0 %*	*22.5 %*
Classified as 1	*2.1 %*	*15.0 %*	*5.8 %*
Class 4 Samples			
Classified as 5	*31.0 %*	*29.7 %*	*31.7 %*
Classified as 4	*60.0 %*	*52.3 %*	*47.0 %*
Classified as 3	*7.9 %*	*12.8 %*	*20.7 %*
Classified as 2	*1.3 %*	*3.0 %*	*0.6 %*
Classified as 1	*0.0 %*	*2.3 %*	*0.0 %*
Class 5 Samples			
Classified as 5	*93.4 %*	*91.7 %*	*88.7 %*
Classified as 4	*6.7 %*	*8.3 %*	*11.3 %*
Classified as 3	*0.0 %*	*0.0 %*	*0.0 %*
Classified as 2	*0.0 %*	*0.0 %*	*0.0 %*
Classified as 1	*0.0 %*	*0.0 %*	*0.0 %*

Table 2: Accuracy of Machine Learning algorithms in ONA case.

5.2.4 Conclusions

Experiments conducted so far show that with *Logistic Multi Class Regression* algorithm zone 5 can be detected in advance and is never mixed with zones 1, 2 and 3. Zone 5 is correctly classified 93.4% of the times (1 mm before the change) but 6.7% of the times is classified as zone 4.

Logistic Multi Class Regression was the better algorithm in samples of zones 1, 2, 4 and 5, but *Decision Tree* algorithm performed better for samples in zone 3.

This accuracy will improve with further experiments in which values of the features will be provided directly by the machine's control. Due to the fact that these values are calculated directly by the micro controller, are more reliable than those calculated out from the signals taken with an oscilloscope. Therefore it is expected an improvement in the classification accuracy.

5.3 OAS

The use case is currently under development. Therefore, the changes that would have to be made to support a new case will be depicted.

First, it is be necessary to see how to extract data from proNTo to SAFIRE platform. proNTO is the process control system used by OAS and it stores data in a Microsoft SQL Server database. In order to extract data from it a custom NiFi template should be developed. This template will connect and interact to the database in order to extract interesting events on different tables. The approach can be seen graphically on Figure 33.

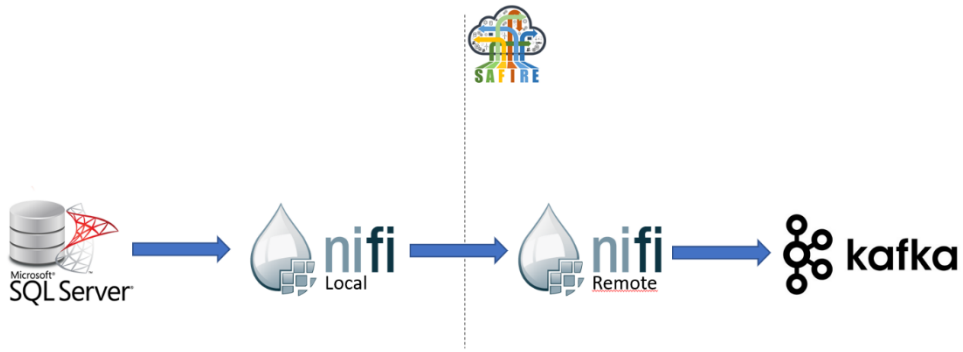


Figure 33 – Local acquisition system to be developed for OAS

The local NiFi will send data to a remote NiFi running on SAFIRE platform. The remote NiFi will perform basic transformations to the incoming data and send the data to a Kafka cluster.

After the data is ingested the data should be analysed and custom analytics will be developed using the tools provided by the Predictive Analytics module.

6. SOFTWARE TOOLS USED FOR IMPLEMENTATION

For the implementation of the Full Prototype several different development tools and IDE[1] have been used. For the overall development of all system modules and components, the Apache Netbeans IDE has been used. The different software tools used, together with their version, link and functionality they are being used for, are listed in the following Table .

Table 4: Overview of used key software tools Table

Functionality	Software	Version	Link
IDE	Apache Netbeans IntelliJ	>= 8.2 >= 2018.1.5	https://netbeans.apache.org/
Build-Management tool	Maven	>=3.5.3	https://maven.apache.org
Version Control	GITlab SVN	>= 2.3	
Issue Management	Jira	>= 6.3	
Infrastructure Automation	Terraform	>=0.11.7	https://www.terraform.io/
Infrastructure Provisioning	Ansible	>=2.4.3.0	https://www.ansible.com/
Programming Language	Java	>= 1.8.0_xx	http://www.java.com
Web Application Framework	Spring	>= 4.1	https://spring.io/
Infrastructure Provisioning	Ansible	>=2.4	https://www.ansible.com/
Infrastructure Automation	Terraform	>=0.11.7	https://www.terraform.io/
Runtime Environment / Application Server	Jetty	>= 8.0	https://www.eclipse.org/jetty/
Unified Big Data Engine	Apache Spark	>= 2.3.0	https://spark.apache.org/
Web based data science	Apache Zeppelin	>= 0.7.3	https://zeppelin.apache.org/
Business intelligence dashboards	Apache Superset	>=0.25.6	https://superset.apache.org/
Complex Event Processing	Espertech	>=7.1.0	http://www.espertech.com/
JPA-based persistence	Hibernate	>= 4.3	http://hibernate.org/
Database for testing	H2 Database	1.3	http://www.h2database.com
Relational Database	PostgreSQL	>= 9.6	https://www.postgresql.org/
No-SQL Database	Apache Cassandra	>=2.2	https://cassandra.apache.org
Data processing and distribution	Apache Nifi Apache Kafka	>=1.6.0 >=1.1.0	https://nifi.apache.org https://kafka.apache.org
Container virtualization	Docker	>=18.03.1-ce	https://www.docker.com

[1] Integrated Development Environment

7. CONCLUSIONS

This document presented the work done by SAFIRE in WP2, in particular in T2.3: Early and Full Prototype of Predictive Analytics Platform, specifically it documents the work on Full Prototype implementation.

Following the requirements and specification for SAFIRE Full Prototype defined in accordance with SAFIRE Concept and Business Case requirements and analysis and the following requirements definition, as well as the data model, external interfaces and functional and technical specifications, the Full Prototype was developed. This document serves as brief description of this Full Prototype implementation given that the result of this task is the developed Software.

8. REFERENCES

[1]. Sanchez JA, Conde A, Arriandiaga A, Wang J, Plaza S. Unexpected Event Prediction in Wire Electrical Discharge Machining Using Deep Learning Techniques. Materials. 2018;11(7):1100. doi:10.3390/ma11071100.

[2]. Alessandra Caggiano,*, Roberto Teti, Roberto Perez, Paul Xirouchakis. Wire EDM Monitoring for Zero-Defect Manufacturing based on Advanced Sensor Signal Processing. 9th CIRP Conference on Intelligent Computation in Manufacturing Engineering - CIRP ICME '14. Procedia CIRP 33 (2015) 315 – 320.

9. APPENDIX

9.1 NIFI DATAFLOWS

In this section, the different Dataflows used for data ingestion are depicted. The code for the XML templates and the different custom NiFi processors can be accessed via de private repository of SAFIRE.

9.1.1 ONA Cloud

This section displays the NiFi dataflows used to ingest the ONA API data.

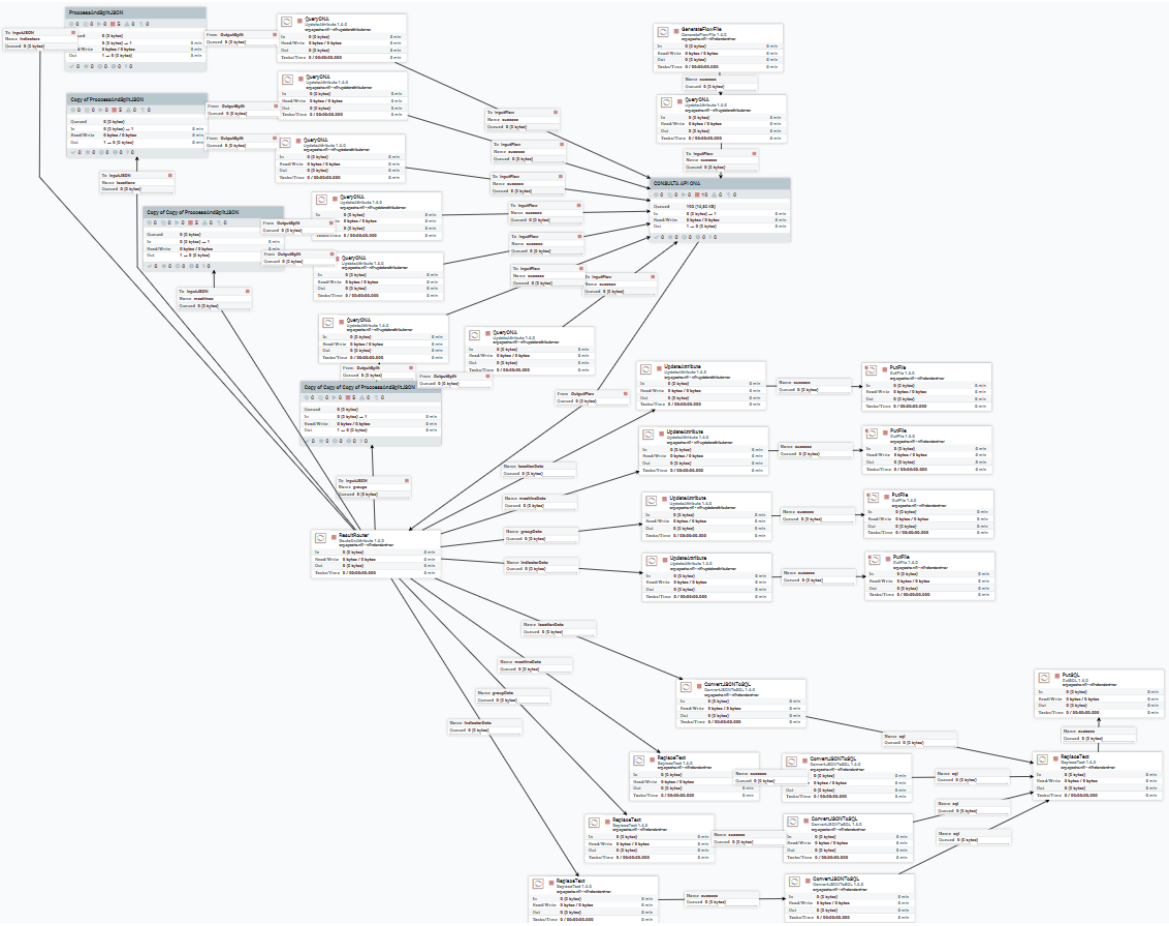


Figure 34 - NiFi dataflow for metadata ingestion.

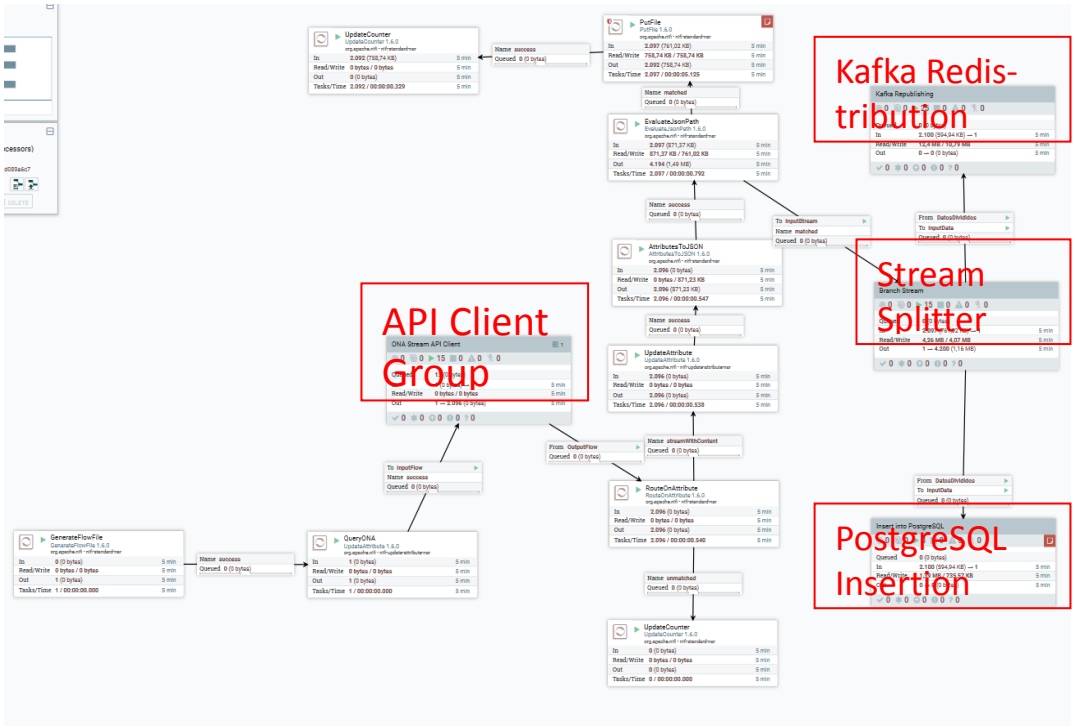


Figure 7 - NiFi dataflow for stream data ingestion.

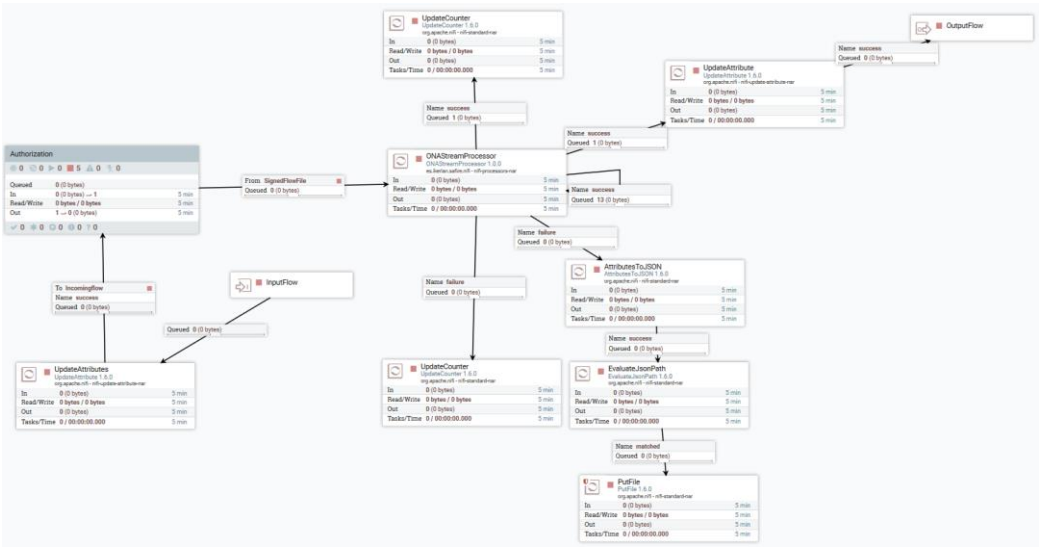


Figure 35 - NiFi dataflow for the API Client Group.

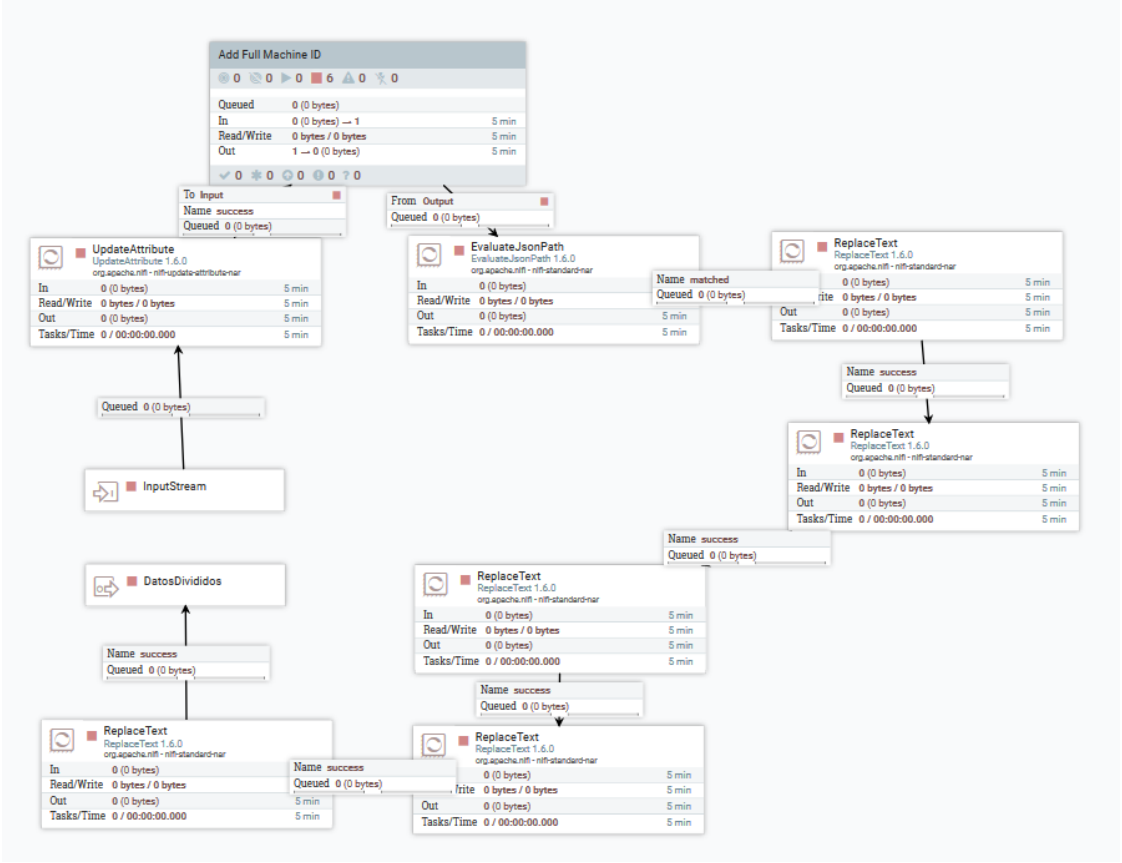


Figure 36 - NiFi dataflow for the Stream Splitter Group.

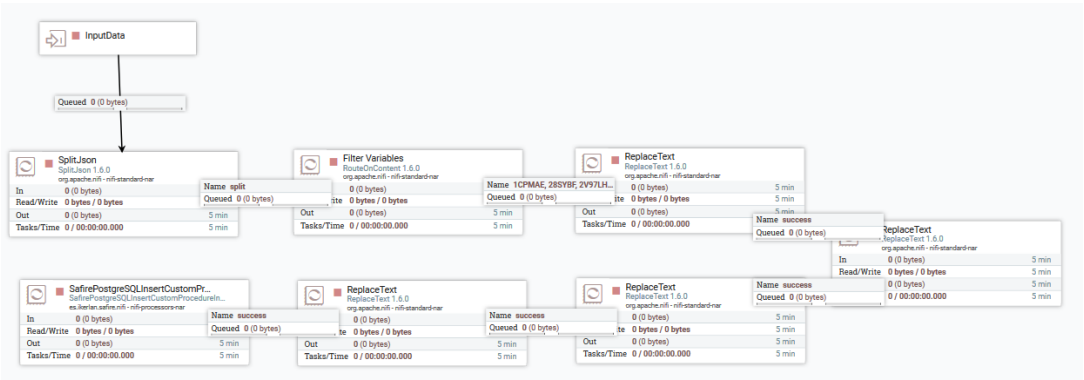


Figure 37 - NiFi dataflow for the PostgreSQL Insertion Group.

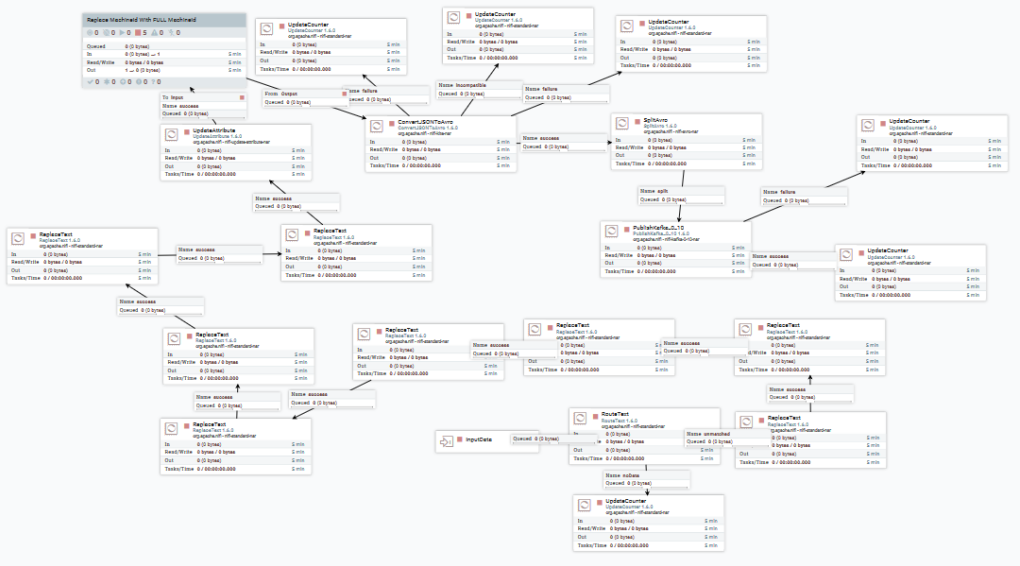


Figure 38 - NiFi dataflow for the Kafka Redistribution Group.

```
package es.ikerlan.safire.nifi.processors;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.nifi.annotation.behavior.*;
import org.apache.nifi.annotation.documentation.CapabilityDescription;
import org.apache.nifi.annotation.documentation.SeeAlso;
import org.apache.nifi.annotation.documentation.Tags;
import org.apache.nifi.annotation.lifecycle.OnScheduled;
import org.apache.nifi.annotation.lifecycle.OnStopped;
import org.apache.nifi.annotation.lifecycle.OnUnscheduled;
import org.apache.nifi.components.PropertyDescriptor;
import org.apache.nifi.flowfile.FlowFile;
import org.apache.nifi.processor.*;
import org.apache.nifi.processor.exception.ProcessException;

import javax.net.ssl.HttpsURLConnection;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.SocketTimeoutException;
import java.net.URL;
import java.net.URLConnection;
import java.util.*;

@Tags({"example"})
@CapabilityDescription("Provide a description")
@SeeAlso({})
@ReadsAttributes({@ReadsAttribute(attribute="", description="")})
@WritesAttributes({@WritesAttribute(attribute="", description="")})
@TriggerSerially
public class ONAStreamProcessor2 extends AbstractProcessor {

    HttpsURLConnection conexion;
```

```
    InputStream inputStream;
    InputStreamReader inputStreamReader;
    BufferedReader bufferedReader;
    boolean sStop;

    boolean error;

    public static final Relationship REL_SUCCESS = new Relationship.Builder()
            .name("success")
            .description("The flow file with the answer received from socket will be
transferred to this relation")
            .build();

    public static final Relationship REL_FAILURE = new Relationship.Builder()
            .name("failure")
            .description("The flow file with the original request that failed will be
transferred to this relation")
            .build();

    public static final Relationship REL_ORIGINAL = new Relationship.Builder()
            .name("original")
            .description("The original request flow file received by the processor")
            .build();

    private List<PropertyDescriptor> descriptors;

    private Set<Relationship> relationships;

    static Log msgLog = LogFactory.getLog(ONAStreamProcessor2.class);


    @Override
    protected void init(final ProcessorInitializationContext context) {
        msgLog.info("init");
        final Set<Relationship> relationships = new HashSet<Relationship>();
        relationships.add(REL_SUCCESS);
        relationships.add(REL_FAILURE);
        relationships.add(REL_ORIGINAL);

        this.relationships = Collections.unmodifiableSet(relationships);
    }

    @Override
    public Set<Relationship> getRelationships() {
        return this.relationships;
    }


    @OnScheduled
    public void onScheduled(final ProcessContext context) {
        msgLog.info("onScheduled");
        sStop = false;
        error = false;

        conexion = null;
        inputStream = null;
        inputStreamReader = null;
        bufferedReader = null;

        //crearConexion y abrirInputStream no se pueden mover aquí, donde sería lo
lógico,
            //ya que necesitan parametros que llegan en el flowfile
    }
```

```
    @Override
    public void onTrigger(final ProcessContext context, final ProcessSession session)
throws ProcessException {
        try {

            msgLog.info("onTrigger");

            if (session == null) {
                msgLog.info("!!!!!!!!!!!!!Session is null!! Should this happen?");
                return;
            }

            FlowFile flowFile = session.get();
            session.transfer(flowFile,REL_ORIGINAL);

            if (sStop) {
                msgLog.info("Stop already triggered. Don't do anything");
                session.remove(flowFile);
                return;
            }


            if (flowFile == null) {
                msgLog.info("!!!!!!!!!!!!!Flowfile is null!! Should this happen?");

                //flowFile = session.create();
                return;
            }

            Map<String, String> attributes = flowFile.getAttributes();




            String flowFileContent;
            try {
                if (conexion == null && !sStop) {
                    crearConexion(attributes);
                }

                if (inputStream == null && !sStop) {
                    abrirInputStream();
                    inputStreamReader = new InputStreamReader(inputStream);
                    bufferedReader = new BufferedReader(inputStreamReader);
                    msgLog.info("Stream readers initialized");
                    //flowFile =
session.putAttribute(flowFile,"XM2CSequence",String.valueOf(Long.valueOf(attributes.g
et("XM2CSequence"))+1));
                }

                if (!sStop) {

                    if (!error) {
                        while(!sStop){
                            FlowFile newFlowFile = leerLinea(flowFile, session);
                            newFlowFile =
session.putAllAttributes(newFlowFile,attributes);
                            session.transfer(newFlowFile, REL_SUCCESS);
                            session.commit();
                        }

                    } else {
```

```
                        String linea = null;
                        if ((linea = bufferedReader.readLine()) != null) {
                            flowFile = session.putAttribute(flowFile, "error_data",
linea);
                        }
                        msgLog.info("Error: " + linea);
                        session.transfer(flowFile, REL_FAILURE);
                        sStop = true;
                    }
                }



                String cause = null;
                if (error) {
                    cause = "error";
                } else {
                    cause = "user";
                }
                msgLog.info(String.format("Stop triggered by %s. Disconnecting",
cause));

                disconnect();


            } catch (MalformedURLException e) {
                msgLog.error(e);
            } catch (SocketTimeoutException e) {
                msgLog.error(e);
            } catch (IOException e) {
                msgLog.error(e);
            }
        } catch (Exception ex) {
            msgLog.error(ex.toString(), ex);
        }
    }

    void crearConexion(Map<String,String> attributes) throws
MalformedURLException,IOException,SocketTimeoutException {
        msgLog.info(System.identityHashCode(this)+" Opening connection. Attributes:
"+attributes);
        URL url = new URL(attributes.get("REMOTE_URL"));
        URLConnection conexionUrl = url.openConnection();
        if (!(conexionUrl instanceof HttpsURLConnection)) {
            throw new IOException("La URL no es una direccion HTTPS valida");
        }
        conexion = (HttpsURLConnection) conexionUrl;
        conexion.setAllowUserInteraction(false);
        conexion.setInstanceFollowRedirects(true);
        conexion.setRequestMethod(attributes.get("HTTP_TYPE"));

        conexion.setRequestProperty("Content-Type", attributes.get("CONTENT_TYPE"));
        conexion.setRequestProperty("X-M2C-Sequence",
attributes.get("XM2CSequence"));
        conexion.setRequestProperty("Authorization",
attributes.get("authorization_signature"));
        conexion.setReadTimeout(500000 * 1000);
    }

    void abrirInputStream() throws IOException{
        int responseCode;
```

```
        responseCode = conexion.getResponseCode();
        msgLog.info("Response code " + responseCode);
        if (responseCode == 200) { // Respuesta correcta del servidor
            msgLog.info("getInputStream");
            inputStream = conexion.getInputStream();
        }
        else {          // Respuesta error del servidor
            msgLog.info("getErrorStream");
            inputStream = conexion.getErrorStream();
            error = true;
        }
    }

    FlowFile leerLinea(FlowFile flowFile,ProcessSession session) throws IOException{
        String linea, datos;
        int numChar, actual;
        char[] buffer;
        FlowFile newFlowFile = null;

        if ((linea = bufferedReader.readLine()) != null) {
            msgLog.info("Reading Line "+flowFile+" "+session);
            numChar = Integer.parseInt(linea);
            buffer = new char[numChar];
            msgLog.info("Line will be "+numChar+" characters long");
            actual = bufferedReader.read(buffer);
            datos = new String(buffer);
            msgLog.info("Line was "+actual+" characters long. Data: "+(actual < 100 ?
datos : datos.substring(0,100)+"..."));
            msgLog.info("Received indicators: "+foundIndicators(datos));

            newFlowFile = session.create();
            // Update the name of the flowFile with data
            newFlowFile = session.putAttribute(newFlowFile,"streamRead",datos);
            // Update the name of the flowFile with the timeStamp
            newFlowFile =
session.putAttribute(newFlowFile,"filename",String.valueOf(System.currentTimeMillis()
));
        }
        else{
            msgLog.info("null line. Close stream?");
        }

        return newFlowFile;
    }

    private List<String> foundIndicators(String linea){
        List<String> ret = null;
        int hasiera,amaiera = -1;

        ret = new ArrayList<>();

        while((hasiera = linea.indexOf("\"I_",amaiera+1)) > -1){
            hasiera = hasiera+1;
            amaiera = linea.indexOf("\"",hasiera);
            ret.add(linea.substring(hasiera,amaiera));
        }

        return ret;
    }

    void disconnect()throws  IOException{
        msgLog.info("DISCONNECT");
        if(bufferedReader != null){
```

```
            bufferedReader.close();
            inputStream = null;
            inputStreamReader = null;
            bufferedReader = null;
        }

        if(conexion != null){
            conexion.disconnect();
            conexion = null;
        }
    }

    @OnStopped
    public void onStopped(){

/*        sStop = true;
        try {
            disconnect();
        } catch (IOException e) {
            msgLog.error(e);
        }
        */
        msgLog.info("OnStopped");
    }
    @OnUnscheduled
    public void onUnscheduled(){
        sStop = true;
        try {
            disconnect();
        } catch (IOException e) {
            msgLog.error(e);
        }
        msgLog.info("onUnscheduled");
    }
}
```

Code 2 - Custom processor for getting data to ONA Cloud API.

```
package es.ikerlan.safire.nifi.processors;

import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.commons.lang.exception.ExceptionUtils;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.nifi.annotation.behavior.*;
import org.apache.nifi.annotation.documentation.CapabilityDescription;
import org.apache.nifi.annotation.documentation.SeeAlso;
import org.apache.nifi.annotation.documentation.Tags;
import org.apache.nifi.annotation.lifecycle.OnScheduled;
import org.apache.nifi.annotation.lifecycle.OnStopped;
import org.apache.nifi.annotation.lifecycle.OnUnscheduled;
import org.apache.nifi.components.PropertyDescriptor;
import org.apache.nifi.dbcp.DBCPService;
import org.apache.nifi.flowfile.FlowFile;
import org.apache.nifi.processor.*;
import org.apache.nifi.processor.exception.ProcessException;
import org.apache.nifi.processor.io.InputStreamCallback;
import org.postgresql.ds.PGSimpleDataSource;
import org.apache.commons.io.IOUtils;
```

```
import javax.sql.DataSource;
import java.io.IOException;
import java.io.InputStream;
import java.nio.charset.StandardCharsets;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.*;

@Tags({"example"})
@CapabilityDescription("Provide a description")
@SeeAlso({})
@ReadsAttributes({@ReadsAttribute(attribute="", description="")})
@WritesAttributes({@WritesAttribute(attribute="", description="")})
@InputRequirement(InputRequirement.Requirement.INPUT_REQUIRED)
public class SafirePostgreSQLInsertCustomProcedureInvocator extends AbstractProcessor
{


    static final PropertyDescriptor CONNECTION_POOL = new
PropertyDescriptor.Builder()
            .name("JDBC Connection Pool")
            .description("Specifies the JDBC Connection Pool to use in order to
convert the JSON message to a SQL statement. "
                    + "The Connection Pool is necessary in order to determine the
appropriate database column types.")
            .identifiesControllerService(DBCPService.class)
            .required(true)
            .build();

    public static final Relationship REL_SUCCESS = new Relationship.Builder()
            .name("success")
            .description("The flow file with the answer received from socket will be
transferred to this relation")
            .build();

    public static final Relationship REL_FAILURE = new Relationship.Builder()
            .name("failure")
            .description("The flow file with the original request that failed will be
transferred to this relation")
            .build();

    private List<PropertyDescriptor> descriptors;

    private Set<Relationship> relationships;

    static Log msgLog =
LogFactory.getLog(SafirePostgreSQLInsertCustomProcedureInvocator.class);


    @Override
    protected List<PropertyDescriptor> getSupportedPropertyDescriptors() {
        final List<PropertyDescriptor> properties = new ArrayList<>();
        properties.add(CONNECTION_POOL);
        return properties;
    }

    @Override
    protected void init(final ProcessorInitializationContext context) {
        msgLog.info("init");
        final Set<Relationship> relationships = new HashSet<Relationship>();
        relationships.add(REL_SUCCESS);
        relationships.add(REL_FAILURE);
```

```
        this.relationships = Collections.unmodifiableSet(relationships);

    }

    @Override
    public Set<Relationship> getRelationships() {
        return this.relationships;
    }

    @Override

    public void onTrigger(final ProcessContext context, final ProcessSession session)
throws ProcessException {
        msgLog.info("onTrigger2");
        final FlowFile flowFile;

        flowFile = session.get();

        if(flowFile == null){
            //This can easily happen when using multiple processing units. All are
called, only one can acquire FlowFile.
            return;
        }

        try{
            msgLog.info("11111111111111111111111111111111111111111111111111111111111111
"+(session != null)+" "+(flowFile != null));
            InputStream in = session.read(flowFile);
            ObjectMapper mapper = null;
            JsonNode root = null;

msgLog.info("22222222222222222222222222222222222222222222222222222222222222"+(in !=
null));

            try {
                mapper = new ObjectMapper();

msgLog.info(String.format("33333333333333333333333333333333333333333333333333 %b
%b",mapper != null,in != null));

                root = mapper.readTree(IOUtils.toString(in, StandardCharsets.UTF_8));

msgLog.info("44444444444444444444444444444444444444444444444444444444444444");

                in.close();

msgLog.info("55555555555555555555555555555555555555555555555555555555555555");

            }
            catch(IOException ex){
                msgLog.info(ex.getMessage(),ex);
                session.transfer(flowFile,REL_FAILURE);
            }
            msgLog.info("inputStream closed");

            String machine, group, indicator;
            Double value = null;
            Long timestamp = null;


            msgLog.info(root.toString());
```

```
msgLog.info("6666666666666666666666666666666666666666666666666666666666666666");

            machine  = root.get("machine").textValue();
            group  = root.get("group").textValue();
            indicator  = root.get("indicator").textValue();
            value  = root.get("value").doubleValue();
            //timestamp = Long.valueOf(root.get("timestamp").textValue());
            String timestampo = root.get("timestamp").textValue();

            msgLog.info("77777777777777777777777777777777777777777777777777");


            try {
                //
                // Prepare a call to the stored procedure 'demoSp'
                // with two parameters
                //
                // Notice the use of JDBC-escape syntax ({call ...})
                //
                //           onaschema.add_stream(machine text, "group" text,
indicator text, value double precision, "timestamp" text )
                msgLog.info(String.format("data: %s %s %s %f
%s",machine,group,indicator,value,timestampo));
                //call.setString(String,String) NO ESTÁ IMPLEMENADO PARA LA LIBRERIA
DE POSTGRESQL. CONSTRUIR EL CALL A PELO
                //se necesita usar Locale.US para que el punto decimal sea un punto y
no una coma.
                String call = String.format(Locale.US,"{call
onaschema.add_stream('%s', '%s', '%s', %f, '%s') }"
                        ,machine.replace("'","\\'")
                        ,group.replace("'","\\'")
                        ,indicator.replace("'","\\'")
                        ,value
                        ,timestampo);
                msgLog.info("call: "+call);
                msgLog.info("Connection? "+conn);
                CallableStatement cStmt = conn.prepareCall(call
                );

                cStmt.execute();
                session.transfer(flowFile,REL_SUCCESS);
            }
            catch (Exception ex) {
                msgLog.error(ex.getMessage(),ex);
                session.transfer(flowFile,REL_FAILURE);
            }
        }
        catch(NullPointerException npex){
            msgLog.info("StackTrace elements? "+npex.getStackTrace().length);
            msgLog.error(ExceptionUtils.getStackTrace(npex));
            throw npex;
        }
        session.commit();
    }

    PGSimpleDataSource ds = null;
    Connection conn = null;
    @OnScheduled
    public void onScheduled(final ProcessContext context) {
        msgLog.info("onScheduled <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<");

        try {
```

```
            conn =
context.getProperty(CONNECTION_POOL).asControllerService(DBCPService.class).getConnec
tion();
            msgLog.info("Connection set");
        } catch (Exception e) {
            msgLog.error(e);
            throw new RuntimeException(e);
        }
    }

    @OnUnscheduled
    void onUnscheduled(){
        msgLog.info("onUnscheduled <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<");
    }

    @OnStopped
    void onstopped(){
        msgLog.info("onStopped <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<");
    }

}
```

Code 3 - Custom processor for storing data to PostgreSQL database

9.1.2 ONA Link

Another method to acquire Data from ONA machines is to acquire data directly from them using the ONA Link protocol. This protocol is based on XML and can be accessed via direct telnet connection with the machines. With this method, a local instance of NiFi will run in the same network than the ONA machine, then, the local NiFi will extract data from the machine and will send it to a Remote NiFi that will be running on SAFIRE platform.

The overall approach can be seen on Figure 39.

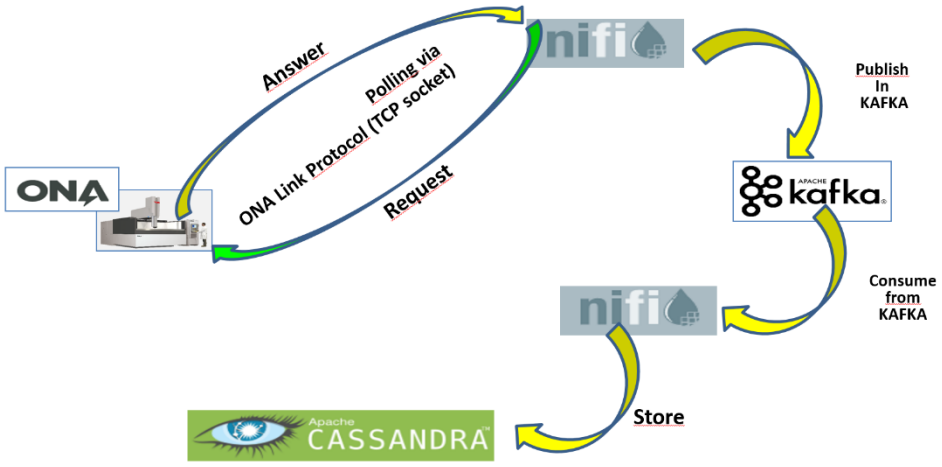


Figure 39 - NiFi dataflow for getting data from ONA Link.

Following, the different process groups and NiFi dataflows for acquiring data using the local approach can be seen.

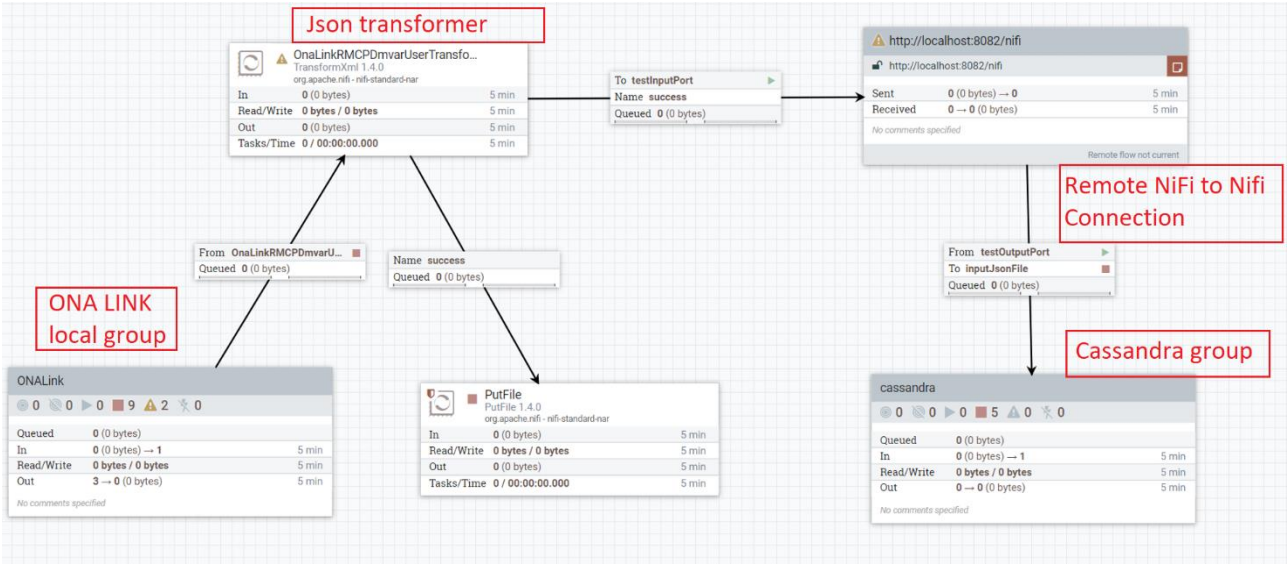


Figure 40 - General NiFi dataflow for obtaining data from ONA Link Protocol.

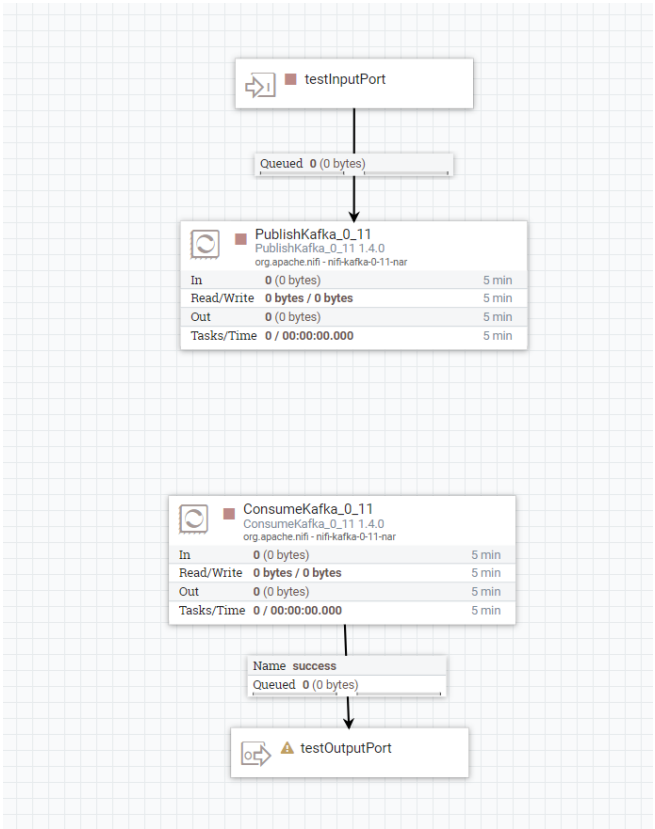


Figure 41 - NiFi dataflow for the Kafka Group.

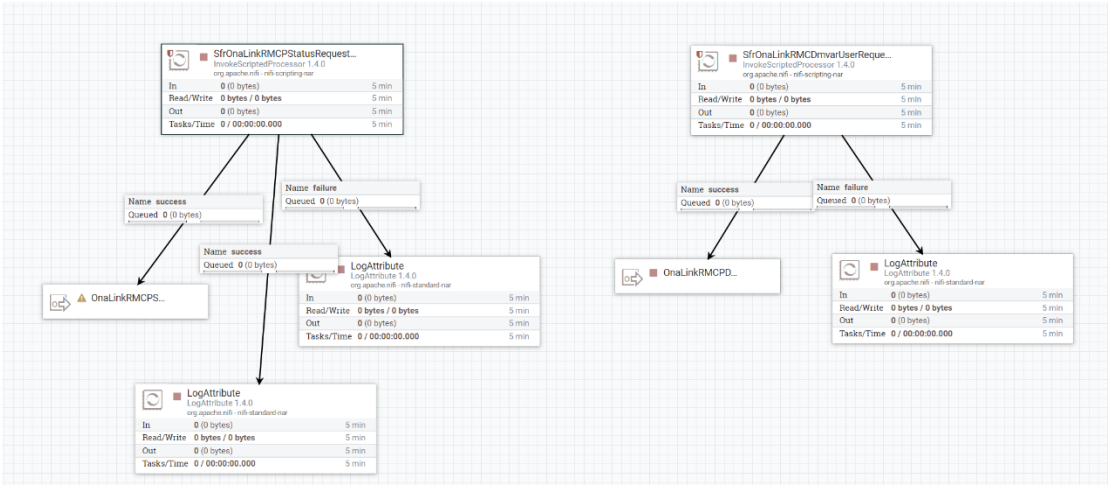


Figure 42 - NiFi dataflow for the Ona Link Group.

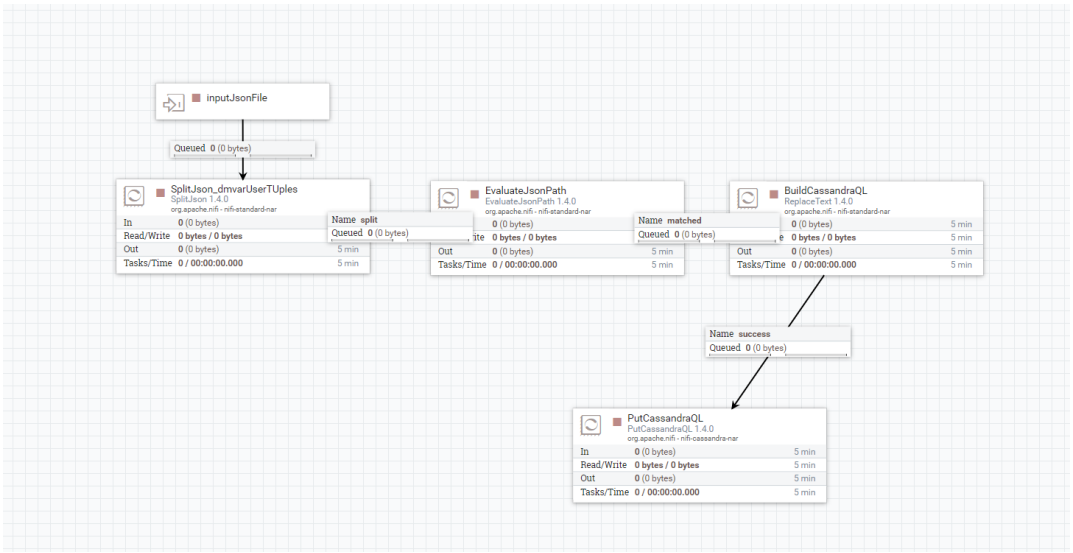


Figure 43 - NiFi dataflow for the Cassandra Group.

Next, the NiFi processor code for getting machine status data using the ONA Link protocol is shown.

```
import java.net.Socket
import java.io.FileWriter;
import java.net.InetSocketAddress;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;
import java.sql.Timestamp;
import java.util.List;

import org.apache.nifi.processor.io.OutputStreamCallback
import org.apache.commons.io.IOUtils
import org.apache.commons.logging.Log;
```

```
import org.apache.commons.logging.LogFactory;

class ClsSfrOnaLinkRMCPStatusRequestClient implements Processor {

    static SocketChannel GlbOnaRMSocketChannel = null;

    def REL_SUCCESS = new Relationship.Builder()
            .name('success')
            .description('The flow file with the answer received from socket will be
transferred to this relation')
            .build();

    def REL_FAILURE = new Relationship.Builder()
            .name('failure')
            .description('The flow file with the original request that failed will be
transferred to this relation')
            .build();

    def ONA_RM_HOSTNAME = new PropertyDescriptor.Builder()
            .name('Hostname').description('Host name or Ip address to be connected
to')

.required(true).expressionLanguageSupported(false).addValidator(Validator.VALID).buil
d()

    def ONA_RM_PORT = new PropertyDescriptor.Builder()
            .name('Port').description('Port number to be connected to')

.required(true).expressionLanguageSupported(false).addValidator(Validator.VALID).buil
d()

    def ONA_RMCP_MSG_STATUS_REQUEST = new PropertyDescriptor.Builder()
            .name('Status Request Message').description('ONA Link Protocol Message to
request State of machine')

.required(true).expressionLanguageSupported(false).addValidator(Validator.VALID).buil
d()

    static                      Log                      msgLog                      =
LogFactory.getLog(ClsSfrOnaLinkRMCPStatusRequestClient.class);

    @Override
    void initialize(ProcessorInitializationContext context) {}

    @Override
    Set<Relationship> getRelationships() { return [REL_SUCCESS, REL_FAILURE] as Set }

    @Override
    void  onTrigger(ProcessContext  context,  ProcessSessionFactory  sessionFactory)
throws ProcessException {

      // Create session and flow file
      def session = sessionFactory.createSession();
      def flowFile = session.create();

      // Variables to store the content to added to the flowfile
      // and to know the relation to which the flowFile has to be sent.
      String flowFileContent = '';
      boolean success = true;

      // Extract paramaters
      String onaRMHostname = '???';
      String onaRMPort = '???';
```

```
      String onaRMCPStatusRequestMessage = '???';

      // Try to catch var data
      try {

        long timeStart;
        long timeEnd;


        // ---------------------------------------------------------------------
        // Get parameters
        // ---------------------------------------------------------------------

        onaRMHostname = context.getProperty(ONA_RM_HOSTNAME)?.getValue();
        onaRMPort = context.getProperty(ONA_RM_PORT)?.getValue();
        onaRMCPStatusRequestMessage                                                    =
context.getProperty(ONA_RMCP_MSG_STATUS_REQUEST)?.getValue();

        // Check no empty values
        if (onaRMHostname == '') {
            throw new Exception("Cannot Start Processor: Hostname not specified.");
        };
        if (onaRMPort == '') {
            throw new Exception("Cannot Start Processor: Port not specified.");
        };
        if (onaRMCPStatusRequestMessage == '') {
            throw new Exception("Cannot Start Processor: Status Request message not
specified.");
        };

        // Check onaRMPort is a number
        if (!onaRMPort.isInteger()) {
            throw new Exception("Cannot Start Processor: Port is not a positive
Integer.");
        };


        // ---------------------------------------------------------------------
        // Socket connection
        // ---------------------------------------------------------------------

        // If SocketChanel not created or is not connected
        if (GlbOnaRMSocketChannel == null || !GlbOnaRMSocketChannel.isConnected()) {

            // Create the socket channel if needed
            if (GlbOnaRMSocketChannel == null) {
                GlbOnaRMSocketChannel = SocketChannel.open();
                GlbOnaRMSocketChannel.configureBlocking(false);
            }

            // Try connect
            msgLog.info('SocketChannel connecting ...');
            GlbOnaRMSocketChannel.connect(new         InetSocketAddress(onaRMHostname,
onaRMPort.toInteger()));

            // Wait at most 5000 ms for connection
            timeStart = System.currentTimeMillis();
            while (!GlbOnaRMSocketChannel.finishConnect()) {
                timeEnd = System.currentTimeMillis();
                if ((timeEnd - timeStart) > 5000) {
                    throw new Exception("SocketChannel Connection timeout.");
                }
            }
```

```
            msgLog.info('SocketChannel connected.');
        }

        // ------------------------------------------------------------------------
        // Status request
        // ------------------------------------------------------------------------

        CharBuffer bufferSendStatusRequest;
        ByteBuffer bufferReceiveStatusRequest;

        // Write status request into channel.
        // Add \r\n to the message request as required by ONA Protocol
        bufferSendStatusRequest   =   CharBuffer.wrap(onaRMCPStatusRequestMessage   +
"\r\n");
        while (bufferSendStatusRequest.hasRemaining()) {

GlbOnaRMSocketChannel.write(Charset.defaultCharset().encode(bufferSendStatusRequest))
;
        }
        msgLog.info('ONA    RMCP    Status    Request    sent    to    server:    '    +
onaRMCPStatusRequestMessage);

        // Loop until a response is received or timeout
        bufferReceiveStatusRequest = ByteBuffer.allocate(1024);
        timeStart = System.currentTimeMillis();
        while (true) {

            // See if any message has been received
            while (GlbOnaRMSocketChannel.read(bufferReceiveStatusRequest)  > 0) {
                bufferReceiveStatusRequest.flip();
                flowFileContent                                                 +=
Charset.defaultCharset().decode(bufferReceiveStatusRequest);
            }

            // If message received assign a time stamp
            if (flowFileContent.length() > 0) {
                 break;
            }

            // Wait at most 5000 ms for answer
            timeEnd = System.currentTimeMillis();
            if ((timeEnd - timeStart) > 5000) {
               throw new Exception("ONA RMCP Status Request timeout.");
            }
        }
        msgLog.info('ONA  RMCP  Status  Request  answer  received  from  server:  '  +
flowFileContent);


        // ------------------------------------------------------------------------
        // Catch the exception
        // ------------------------------------------------------------------------
        } catch(e) {
            msgLog.info("ClsSfrOnaLinkRMCPStatusRequestClient    Exception:    "    +
e.getMessage());
            flowFileContent  =  "ClsSfrOnaLinkRMCPStatusRequestClient  Exception:  "  +
e.getMessage()
            success = false
        }


        // ------------------------------------------------------------------------
        // Transfer
```

```
        // --------------------------------------------------------------------

        // Update the name of the flowFile with the timeStamp
        flowFile = session.putAttribute(flowFile, 'filename', "ONALinkLogData" + "_"
+       onaRMHostname       +       "_"       +       onaRMPort       +       "_"       +
String.valueOf(System.currentTimeMillis()));

        // Add content of the answer message to the flow file
        flowFile       =       session.write(flowFile,       {       outStream       ->
outStream.write(flowFileContent.getBytes("UTF-8"))} as OutputStreamCallback);

        // Uoadte content of flowFile
        if (success) {
            msgLog.info('SUCCESS: FlowFile Content: ' + flowFileContent);
            session.transfer(flowFile, REL_SUCCESS);
        } else {
            msgLog.info('FAILURE: FlowFile Content: ' + flowFileContent);
            session.transfer(flowFile, REL_FAILURE);
        }

        // Commit the transaction
        session.commit();
    }

    @Override
    Collection<ValidationResult> validate(ValidationContext context) { return null }

    @Override
    PropertyDescriptor getPropertyDescriptor(String name) {
        switch(name) {
            case 'Hostname': return ONA_RM_HOSTNAME
            case 'Port': return ONA_RM_PORT
            case 'Status Request Message': return ONA_RMCP_MSG_STATUS_REQUEST
            default: return null
        }
    }

    @Override
    void  onPropertyModified(PropertyDescriptor  descriptor,  String  oldValue,  String
newValue) { }

    @Override
    List<PropertyDescriptor>  getPropertyDescriptors()  {  return  [ONA_RM_HOSTNAME,
ONA_RM_PORT, ONA_RMCP_MSG_STATUS_REQUEST] as List }

    @Override
    String    getIdentifier()    {    return    'ClsSfrOnaLinkRMCPStatusRequestClient-
InvokeScriptedProcessor' }

}

processor = new ClsSfrOnaLinkRMCPStatusRequestClient()
```

Code 4. Groovy Script for getting Status data from ONA machines

Finally, the NiFi processor code for obtaining machine variable data using the ONA Link protocol is shown.

```
import java.net.Socket
import java.io.FileWriter;
import java.net.InetSocketAddress;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;
import java.sql.Timestamp;
import java.util.List;

import org.apache.nifi.processor.io.OutputStreamCallback
import org.apache.commons.io.IOUtils
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

class ClsSfrOnaLinkRMCPStatusRequestClient implements Processor {

    static SocketChannel GlbOnaRMSocketChannel = null;

    def REL_SUCCESS = new Relationship.Builder()
            .name('success')
            .description('The flow file with the answer received from socket will be
transferred to this relation')
            .build();

    def REL_FAILURE = new Relationship.Builder()
            .name('failure')
            .description('The flow file with the original request that failed will be
transferred to this relation')
            .build();

    def ONA_RM_HOSTNAME = new PropertyDescriptor.Builder()
            .name('Hostname').description('Host name or Ip address to be connected
to')

.required(true).expressionLanguageSupported(false).addValidator(Validator.VALID).buil
d()

    def ONA_RM_PORT = new PropertyDescriptor.Builder()
            .name('Port').description('Port number to be connected to')

.required(true).expressionLanguageSupported(false).addValidator(Validator.VALID).buil
d()

    def ONA_RMCP_MSG_STATUS_REQUEST = new PropertyDescriptor.Builder()
            .name('Status Request Message').description('ONA Link Protocol Message to
request State of machine')

.required(true).expressionLanguageSupported(false).addValidator(Validator.VALID).buil
d()

    static Log msgLog =
LogFactory.getLog(ClsSfrOnaLinkRMCPStatusRequestClient.class);

    @Override
    void initialize(ProcessorInitializationContext context) {}

    @Override
    Set<Relationship> getRelationships() { return [REL_SUCCESS, REL_FAILURE] as Set }

    @Override
```

```
    void onTrigger(ProcessContext context, ProcessSessionFactory sessionFactory)
throws ProcessException {

      // Create session and flow file
      def session = sessionFactory.createSession();
      def flowFile = session.create();

      // Variables to store the content to added to the flowfile
      // and to know the relation to which the flowFile has to be sent.
      String flowFileContent = '';
      boolean success = true;

      // Extract paramaters
      String onaRMHostname = '???';
      String onaRMPort = '???';
      String onaRMCPStatusRequestMessage = '???';

      // Try to catch var data
      try {

        long timeStart;
        long timeEnd;


        // ------------------------------------------------------------------------
        // Get parameters
        // ------------------------------------------------------------------------

        onaRMHostname = context.getProperty(ONA_RM_HOSTNAME)?.getValue();
        onaRMPort = context.getProperty(ONA_RM_PORT)?.getValue();
        onaRMCPStatusRequestMessage =
context.getProperty(ONA_RMCP_MSG_STATUS_REQUEST)?.getValue();

        // Check no empty values
        if (onaRMHostname == '') {
            throw new Exception("Cannot Start Processor: Hostname not specified.");
        };
        if (onaRMPort == '') {
            throw new Exception("Cannot Start Processor: Port not specified.");
        };
        if (onaRMCPStatusRequestMessage == '') {
            throw new Exception("Cannot Start Processor: Status Request message not
specified.");
        };

        // Check onaRMPort is a number
        if (!onaRMPort.isInteger()) {
            throw new Exception("Cannot Start Processor: Port is not a positive
Integer.");
        };


        // ------------------------------------------------------------------------
        // Socket connection
        // ------------------------------------------------------------------------

        // If SocketChanel not created or is not connected
        if (GlbOnaRMSocketChannel == null || !GlbOnaRMSocketChannel.isConnected()) {

            // Create the socket channel if needed
            if (GlbOnaRMSocketChannel == null) {
                GlbOnaRMSocketChannel = SocketChannel.open();
                GlbOnaRMSocketChannel.configureBlocking(false);
```

```
            }

            // Try connect
            msgLog.info('SocketChannel connecting ...');
            GlbOnaRMSocketChannel.connect(new InetSocketAddress(onaRMHostname,
onaRMPort.toInteger()));

            // Wait at most 5000 ms for connection
            timeStart = System.currentTimeMillis();
            while (!GlbOnaRMSocketChannel.finishConnect()) {
                timeEnd = System.currentTimeMillis();
                if ((timeEnd - timeStart) > 5000) {
                    throw new Exception("SocketChannel Connection timeout.");
                }
            }
            msgLog.info('SocketChannel connected.');
        }

        // ------------------------------------------------------------------------
        // Status request
        // ------------------------------------------------------------------------

        CharBuffer bufferSendStatusRequest;
        ByteBuffer bufferReceiveStatusRequest;

        // Write status request into channel.
        // Add \r\n to the message request as required by ONA Protocol
        bufferSendStatusRequest = CharBuffer.wrap(onaRMCPStatusRequestMessage +
"\r\n");
        while (bufferSendStatusRequest.hasRemaining()) {

GlbOnaRMSocketChannel.write(Charset.defaultCharset().encode(bufferSendStatusRequest))
;
        }
        msgLog.info('ONA RMCP Status Request sent to server: ' +
onaRMCPStatusRequestMessage);

        // Loop until a response is received or timeout
        bufferReceiveStatusRequest = ByteBuffer.allocate(1024);
        timeStart = System.currentTimeMillis();
        while (true) {

            // See if any message has been received
            while (GlbOnaRMSocketChannel.read(bufferReceiveStatusRequest)  > 0) {
                bufferReceiveStatusRequest.flip();
                flowFileContent +=
Charset.defaultCharset().decode(bufferReceiveStatusRequest);
            }

            // If message received assign a time stamp
            if (flowFileContent.length() > 0) {
                 break;
            }

            // Wait at most 5000 ms for answer
            timeEnd = System.currentTimeMillis();
            if ((timeEnd - timeStart) > 5000) {
               throw new Exception("ONA RMCP Status Request timeout.");
            }
        }
        msgLog.info('ONA RMCP Status Request answer received from server: ' +
flowFileContent);
```

```
        // ---------------------------------------------------------------------
        // Catch the exception
        // ---------------------------------------------------------------------
        } catch(e) {
            msgLog.info("ClsSfrOnaLinkRMCPStatusRequestClient Exception: " +
e.getMessage());
            flowFileContent = "ClsSfrOnaLinkRMCPStatusRequestClient Exception: " +
e.getMessage()
            success = false
        }


        // ---------------------------------------------------------------------
        // Transfer
        // ---------------------------------------------------------------------

        // Update the name of the flowFile with the timeStamp
        flowFile = session.putAttribute(flowFile, 'filename', "ONALinkLogData" + "_"
+ onaRMHostname + "_" + onaRMPort + "_" +
String.valueOf(System.currentTimeMillis()));

        // Add content of the answer message to the flow file
        flowFile = session.write(flowFile, { outStream ->
outStream.write(flowFileContent.getBytes("UTF-8"))} as OutputStreamCallback);

        // Uoadte content of flowFile
        if (success) {
            msgLog.info('SUCCESS: FlowFile Content: ' + flowFileContent);
            session.transfer(flowFile, REL_SUCCESS);
        } else {
            msgLog.info('FAILURE: FlowFile Content: ' + flowFileContent);
            session.transfer(flowFile, REL_FAILURE);
        }

        // Commit the transaction
        session.commit();
    }

    @Override
    Collection<ValidationResult> validate(ValidationContext context) { return null }

    @Override
    PropertyDescriptor getPropertyDescriptor(String name) {
        switch(name) {
            case 'Hostname': return ONA_RM_HOSTNAME
            case 'Port': return ONA_RM_PORT
            case 'Status Request Message': return ONA_RMCP_MSG_STATUS_REQUEST
            default: return null
        }
    }

    @Override
    void onPropertyModified(PropertyDescriptor descriptor, String oldValue, String
newValue) { }

    @Override
    List<PropertyDescriptor> getPropertyDescriptors() { return [ONA_RM_HOSTNAME,
ONA_RM_PORT, ONA_RMCP_MSG_STATUS_REQUEST] as List }

    @Override
    String getIdentifier() { return 'ClsSfrOnaLinkRMCPStatusRequestClient-
InvokeScriptedProcessor' }
```

```
}

processor = new ClsSfrOnaLinkRMCPStatusRequestClient()
```

Code 5. Groovy Script for getting Status data from ONA machines.

9.1.3 Electrolux

This section displays the NiFi dataflow used in the final prototype to ingest data from Electrolux induction hobs and Matlab(c). The NiFi dataflow can be overviewed in Figure 44.

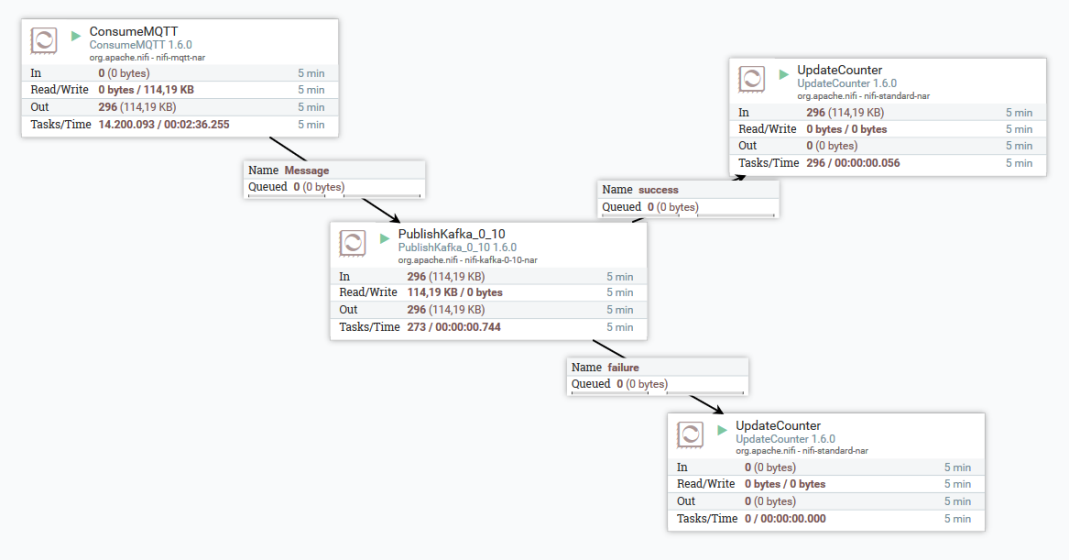


Figure 44 - NiFi dataflow for getting data from MQTT.

9.2 ESPER RULES

Event	Type	Rule
semRojo	Event	semRojoFlag1 <> 0 OR semRojoFlag2 <> 0 OR semRojoFlag3 <> 0 OR semRojoFlag4 <> 0
semAmbar	Event	semRojo = 0 AND (semAmbarFlag1 <> 0 OR semAmbarFlag2 <> 0 OR semAmbarFlag3 <> 0 OR semAmbarFlag4 <> 0)
semGris	Event	semRojo = 0 AND semAmbar = 0 AND (semGrisVerdeCond1 <> 1 OR semGrisCond2 <> 0 OR semGrisCond3 <> 0)
semVerde	Event	semRojo = 0 AND semAmbar = 0 AND semGris = 0 AND (semGrisVerdeCond1

		<> 0 semVerdeCond2 <> 0)
Thickness	Event	IF thickness is NULL THEN 20 ELSE thickness
MMR	Event	wireSpeed /thickness
spoolWireRemainingTime	Event	wireLength / wireSpeed
codState	Event	semRojo * 1 + semAmbar * 2 + semGris * 3 + semVerde *4
State	Event	IF codState = 1 THEN Idle IF codState = 2 THEN Alarm-Yellow IF codState = 3 THEN Warning-Red IF codState = 4 THEN Running-Green ELSE Error
instantConsumption	Event	IF codState = 4 THEN wireSpeed / wireDiameter
WconductivityInf	Warning	IF codState = 4 THEN(IF conductivity <= 11 THEN 1 ELSE 0) ELSE 0
WconductivitySup	Warning	IF codState = 4 THEN(IF spoolRemainingWirePercent >= 15 THEN 1 ELSE 0) ELSE 0
WspoolRemainingWirePercent	Warning	IF codState = 4 THEN(IF conductivity <= 10 THEN 1 ELSE 0) ELSE 0
Wtemperature_01Hour	Warning	(MAX(ambientTemperature) – MIN(ambientTemperature) INTERVAL 1 HOUR) > 2
Wtemperature_24Hour	Warning	(MAX(ambientTemperature) – MIN(ambientTemperature) INTERVAL 1 DAY) > 4
WmachineRunning	Warning	codState <> 4

9.3 PREDICTIVE ANALYTICS TEMPLATES

In this section, the *Source Code templates* used to (a) *Define and Train Predictive Models*, (b) *Develop Predictive Analytics Prediction REST Web Service Clients* and (c) *Access Predictive Analytics by Sending Messages to MQTT, NiFi and Kafka* are given.

Part of this material has been specified in SAFIRE D2.5 full prototype specifications deliverable but is included here again for clarity. The source code templates and the examples allow non-expert users to define and train medium complexity models.

9.3.1 Templates to Define and Train Predictive Models

9.3.1.1 Templates for Spark

Python source code below shows a simple spark template of a *Logistic Regression* (this function is part of several source code files used for ONA Electroerosion test case).

```
def trainModelLR(dataFrame, dataFrameFeatureColNames):

    # Assemble the input to produce the features column
    assembler = VectorAssembler(inputCols=dataFrameFeatureColNames, outputCol="features")

    # TODO
    # Select the machine learning algorithm and its parameters
    # In this case a Logistic Regression has been selected
    lr = LogisticRegression(maxIter  = 100, regParam = 0.01)

    # Chain in a pipeline the transformations
    # and machine learning algorithm
    pipeline = Pipeline(stages = [assembler, lr])

    # TODO
    # Create a Parameter Grid for Cross Validation
    # Assign a range to the hyper parameter for fine-tuning
    paramGrid = (ParamGridBuilder()
            .addGrid(lr.regParam, [0.01, 0.1, 0.3, 0.5])
            .addGrid(lr.maxIter, [10,25, 50, 100])
            .addGrid(lr.elasticNetParam, [0.0, 0.1, 0.2])
            .build())

    # Define cross validation model
    crossval = CrossValidator(estimator=pipeline,
                              estimatorParamMaps=paramGrid,
                              evaluator=BinaryClassificationEvaluator(),
                              numFolds=5)

    # Fit (train) the model
    model = crossval.fit(dataFrame)

    # Return the model bestModel
    return model
```

Python source code below shows a simple spark template of a *Decision Tree* algorithm (this function is part of several source code files). Following Spark's philosophy, it is very easy to interchange the algorithms to use to experiment with different alternatives. The code of the *logistic regression* and the *decision tree* is very similar.

```
def trainModelDT(dataFrame, dataFrameFeatureColNames):
```

```
    # Assemble the input to produce the features column
    assembler = VectorAssembler(inputCols=dataFrameFeatureColNames,  outputCol="features")

    # TODO
    # Select the machine learning algorithm and its parameters
    # In this case a Decision Tree Classifier has been selected
    dt = DecisionTreeClassifier()

    # Chain in a pipeline the transformations
    # and machine learning algorithm
    pipeline = Pipeline(stages = [assembler, dt])

    # TODO
    # Create a Parameter Grid for Cross Validation
    # Assign a range to the hyper parameter for fine-tuning
    paramGrid = (ParamGridBuilder()
            .addGrid(dt.maxDepth, [5, 10, 15, 20])
            .addGrid(dt.maxBins, [5, 10, 20, 40])
            .build())

    # Define cross validation model
    crossval = CrossValidator(estimator=pipeline,
                        estimatorParamMaps=paramGrid,
                        evaluator=BinaryClassificationEvaluator(),
                        numFolds=5)

    # Fit (train) the model
    model = crossval.fit(dataFrame)

    # Return the trained model
    return model
```

Python source code below shows a simple spark template of a *Random Forest Tree* algorithm (this function is part of several source code files).

```
def trainModelRF(dataFrame, dataFrameFeatureColNames):

    # Assemble the input to produce the features column
    assembler = VectorAssembler(inputCols=dataFrameFeatureColNames, outputCol="features")

    # TODO
    # Select the machine learning algorithm and its parameters
    # In this case a Random Forest Tree Classifier has been selected
    rf = RandomForestClassifier(labelCol="label",featuresCol="features")

    # Chain in a pipeline the transformations and machine learning algorithm
    pipeline = Pipeline(stages = [assembler, rf])

    # TODO
    # Create a Parameter Grid for Cross Validation
    # Assign a range to the hyper parameter for fine-tuning
    paramGrid = ParamGridBuilder()\
      .addGrid(rf.maxDepth, [2,4,10])\
      .addGrid(rf.numTrees, [10, 50, 100])\
      .build()

    # Define cross validation model
    crossval = CrossValidator(estimator=pipeline,
```

```
                    estimatorParamMaps=paramGrid,
                    evaluator=MulticlassClassificationEvaluator(),
                    numFolds=5)

    # Fit (train) the model
     model = crossval.fit(dataFrame)

     # Return the trained model
     return model
```

Python source code below shows a simple spark template of a *OneVsRest* algorithm (this function is part of several source code files).

```
def trainModelOneVsRest(dataFrame, dataFrameFeatureColNames):

    # Assemble the input to produce the features column
    assembler = VectorAssembler(inputCols=dataFrameFeatureColNames, outputCol="features")

    # TODO
     # Select the machine learning algorithm and its parameters
     # In this case a Random Forest Tree Classifier has been selected
    lr = LogisticRegression(maxIter = 100,
                                regParam = 0.01,
                                elasticNetParam = 0.8,
                                tol=1E-6, fitIntercept=True)
    ovr = OneVsRest(classifier=lr)

    # Chain in a pipeline the transformations and machine learning algorithm
    pipeline = Pipeline(stages = [assembler, ovr])

    # TODO
    # Create a Parameter Grid for Cross Validation
    # Assign a range to the hyper parameter for fine-tuning
    paramGrid = (ParamGridBuilder()
            .addGrid(lr.regParam, [0.01, 0.1, 0.3, 0.5])
            .addGrid(lr.maxIter, [10, 25, 100, 500])
            .addGrid(lr.elasticNetParam, [0.0, 0.1, 0.2, 0.8])
            .build())

    # Define cross validation model
    crossval = CrossValidator(estimator=pipeline,
                    estimatorParamMaps=paramGrid,
                    evaluator=MulticlassClassificationEvaluator(),
                    numFolds=5)



    # Fit (train) the model
     model = crossval.fit(dataFrame)

     # Return the trained model
     return model
```

9.3.1.2 *Templates for Keras*

Python source code below shows a simple template for Keras to define and fit a Neural Network (this function is part of several source code files used for Electrolux test case).

Define a Neural Network

```
def model(input_shape):
    '''
    Function creating the model's graph in Keras
    Argument:  input_shape -- shape of the model's input data (using Keras conventions)
    Returns:   model -- Keras model instance
    '''

 # Define input shape
 X_input = Input(shape = input_shape)

# Define first dense (fully connected)  layer with 100 neurons and 'relu' activation
X = Dense(100, input_dim=cnst.Tx, kernel_initializer='normal', activation='relu')(X_input)

# Define hidden dense (fully connected) layers
X = Dense(30, kernel_initializer='normal', activation='relu')(X)
X = Dense(10, kernel_initializer='normal', activation='relu')(X)

# Define output layer with 'sigmoid' activation for binary classification
X = Dense(1, kernel_initializer='normal', activation='sigmoid')(X)

# Finally build a return the model
model = Model(inputs = X_input, outputs = X)

return model
```

Compile and train the model

```
# Define the model invoking previous model
modelCurF08 = model(input_shape = [100])

# Compile the model for binary classification and adam training algorithm
modelCurF08.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Fit the model
modelCurF09.fit(X_trainCurF08, Y_trainCurF08, batch_size = 50, epochs = 500, shuffle=True,
class_weight = class_weightCurF08, validation_data=[X_devCurF08, Y_devCurF08])
```

9.3.2 Templates to Develop Predictive Analytics REST Web Service and Clients

This section (already included in D2.5 deliverable and reproduced here for clarity) shows the templates to develop Predictive Analytics REST Web Services and clients implemented in full prototype.

Service Specifications

This section specifies the input parameters of the service and the specification of the answer returned by service.

Service name

- ***String*** `SafirePrdAnalyticsPredictor`

Parameters in the Request

- ***String*** `ipAddress` – Identifies the ip address where the service is located.
- ***String*** `port` – Connection port to the service.
- ***long*** `clientId` – Identifies the client's request. Can be any number provided by the client. This identification will be included back with the answer.
- ***String*** `clientTopic` – Client topic is a string provided by the client. It is simply a complement to the client's request and might be the empty string. This topic will be included back with the answer and can help the client to indentify better the answer. An example of client's topic may be "*Boil_detection_25-Oct-2018_16-51-00*" that identifies a boiling experiment.

- ***String*** `modelName` – Upon request, the service will (a) invoke and load a previously trained predictive analytics model and (b) will call the model to predict values according to *dataFrameRowDataJSON* parameter (see below).

- ***String*** `backendName` – Indicates the backend that will process the invocation. Allowed values are: ***spark*** or ***keras***

- ***String*** `dataFrameColNamesJSON` – Contains the dataframe column names in JSON format, according to the following syntax:

  ```
  {"dataFrameColNames":["name1","name2",....]}
  ```

  Example (three columns case):

  ```
  {"dataFrameColNames":["id","text","label"]}
  ```

- ***String*** `dataFrameColTypesJSON` – Contains the dataframe column types in JSON format. Allowed types are *integer*, *double*, *string*, *arrayInteger*, *arrayDouble*. Syntax is as follows:

  ```
  {"dataFrameColTypes":["type1","type2", ...]}
  ```

  Example (three columns case):

  ```
  {"dataFrameColTypes":["integer","string","double"]}
  ```

- ***String*** `dataFrameRowDataJSON` – Contains the dataframe rows in JSON format. Each row must have the number of values specified in *dataFrameColNamesJSON* with its corresponding type specified in *dataFrameColTypesJSON*. Syntax is as follows:

```
{"dataFrameRowData":
      [[row1data1, row1data2, ...],
      [row2data1, row2data2, ...],
      [row3data1, row3data2, ...],
                  ...
      ]}
```

Example 1 (two rows with three columns of type *integer*, *string*, *double*):

```
{"dataFrameColNames":["id","text","label"]}
{"dataFrameColTypes":["integer","string","double"]}
{"dataFrameRowData":
      [[7,"this is an example ssd", 1.0],
       [8,"another text", 0.0]]}
```

Example 2 (two rows with one column of type *arrayDouble*):

```
{"dataFrameColNames":["currentValues"]}
{"dataFrameColTypes":["arrayDouble"]}
{"dataFrameRowData":
      [[[1.456, 2.3456, 3.2345, 1.3456]],
      [[2.3737, 4.2829, 1.2876, 8.7625]]}
```

Answer given by the service

The service will always return a JSON string containing the following fields:

- `long callCount` - Represents an automatic counter with the number of times the service has been requested since it was started (just informative purpose).
- `long clientId`- The client identification that was provided by the client in the request.
- `long clientTopic` - The client topic that was provided by the client in the request.
- `String modelName` – The predictive model that was provided by the client in the request.
- `String backendName` – The backend that was provided by the client in the request.
- `String dataFrameRowDataPredictionJSON` – In this parameter, the service returns in this parameter the predicted values for each row received in the request's param `dataFrameRowDataJSON`. The syntax of the JSON string (similar to that of `dataFrameRowDataJSON`) is the following:

```
{"dataFrameRowDataPrediction":
      [[Row1Prediction1, Row1Prediction2, ...],
       [Row2Prediction1, Row2Prediction2, ...],
       [Row3Prediction1, Row3Prediction2, ...],
                  ...
      ]}
```

Example: (predicted values for three rows, where each predicted value is a double):

```
{"dataFrameRowDataPrediction":
      [[0.9878],
       [0.45627],
       [0.87265]
      ]}
```

Note: The number of predicted values per row and their types is implicitly defined in the predictive model, but not defined in the request. Therefore, the client receiving the answer must know the expected number and types of fields.

- `String errorDescription` – The description of the error when the service execution fails (retCode <> 0).
- `int retCode` – Return code value is 0 when the service execution succeed, and non-zero otherwise.

Example of Request

The client sends a request as follows (Electrolux case example):

```
http://localhost:8080/SafirePrdAnalyticsPredictor?
clientId=1&
clientTopic=Boil_detection_26-10-2018_10-57-41&
modelName=electroluxNNTraineModelCurF08.h5&
backendName=keras&
dataFrameColNamesJSON=
      {"dataFrameColNames":["currentValues"]}&
dataFrameColTypesJSON=
      {"dataFrameColTypes":["arrayDouble"]}&
dataFrameRowDataJSON=
      {"dataFrameRowData":[[[1.54418102,  1.48782741, ... ]]]}
```

Example of Answer

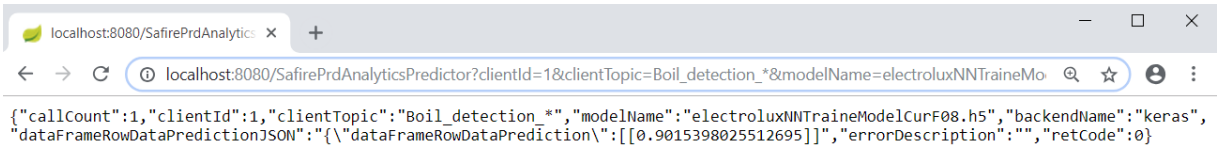
The service processes the request and answers with the following:

```
ClsSafirePrdAnalyticsPredictorWebServiceAnswer {
      callCount=1,
      clientId=1,
      clientTopic= Boil_detection_26-10-2018_10-57-41,
      modelName=electroluxNNTraineModelCurF08.h5,
      backendName=keras,
      prediction={"dataFrameRowDataPrediction":[[0.9015398025512695]]},
      errorDescription="",
      retCode=0}
```

In this particular case, the answer contains the prediction of the single sample passed as parameter being boiling (90,15%).

Invoking the Prediction Service from a Web Navigator

The Prediction Service is also callable from a Web navigator. **Error! Reference source ot found.**Figure below shows a call to the service executed from a web navigator (in this case Google Chrome) and the response given by the service.



Template for invocation from a Java Client Specifications

Source Code below represents templates for an easy development of REST Java clients. The template is composed by the two classes described below.

Class ClsSafirePrdAnalyticsPredictorWebServiceAnswer

This class represents a Java client. Only the section with *TODO* must be modified by the end-user using the template. The SAFIRE Predictive analytics REST Web Service client for Electrolux test case in the full prototype has been developed with this template.

```
@SpringBootApplication
public class ClsSafireWebServiceRestClientTemplate  {

	private static final Logger log =
		LoggerFactory.getLogger(ClsSafireWebServiceRestClientTemplate .class);

	public static void main(String args[]) {
		SpringApplication.run
			(ClsSafireWebServiceRestClientTemplate .class);
	}

	@Bean
	public RestTemplate restTemplate(RestTemplateBuilder builder) {
		return builder.build();
	}

	@Bean
	public CommandLineRunner run(RestTemplate restTemplate) throws Exception {

		return args -> {

			// Call service
			CallService(restTemplate);
		};
	}

	private void CallService(RestTemplate restTemplate) {

		// TODO
```

```
                // Define call parameters
                String port = "8080";
                long clientId = 1;
                String topic = "Boil_detection_29-oct-2018_12-08-00";
                String modelName = "electroluxNNTraineModelCurF08.h5";
                String backendName = "keras";

                // TODO
                // Generate the sample to predict
                // The key dataframe elements must be defined
                // The columns names, types and data values
                // Row Data is a list of data rows. The service will answer with
                // a prediction for each row.
                String dataFrameColNamesJSON =
                                "{\"dataFrameColNames\":[\"currentValues\"]}";
                String dataFrameColTypesJSON =
                                "{\"dataFrameColTypes\":[\"arrayDouble\"]}";
                String dataFrameRowDataJSON =
                                "{\"dataFrameRowData\":"
                                + "[[[1.456, 2.3456, 3.2345, 1.3456]],"
                                + "[[2.3737, 4.2829, 1.2876, 8.7625]]}";

                // Encode the data frame elements
                // This is necessary as they contain reserved chars for http requests
                dataFrameColNamesJSON =
                        UriUtils.encodeQueryParam(dataFrameColNamesJSON,"UTF-8");
                dataFrameColTypesJSON =
                        UriUtils.encodeQueryParam(dataFrameColTypesJSON,"UTF-8");
                dataFrameRowDataJSON =
                        UriUtils.encodeQueryParam(dataFrameRowDataJSON,"UTF-8");

                // Build the call to the prediction service
                String serviceCall =
                        "http://localhost:" + port + "/SafirePrdAnalyticsPredictor?" +
                        "clientId=" + String.valueOf(clientId) + "&" +
                        "clientTopic=" + topic + "&" +
                        "modelName=" + modelName + "&" +
                        "backendName=" + backendName + "&" +
                       "dataFrameColNamesJSON=" + dataFrameColNamesJSON  + "&" +
                       "dataFrameColTypesJSON=" + dataFrameColTypesJSON  + "&" +
                        "dataFrameRowDataJSON=" + dataFrameRowDataJSON;

                // Call the prediction service
                // In this case the call is synchronous so the caller will be
                // blocked here waiting for the answer.
                ClsSafirePrdAnalyticsPredictorWebServiceAnswer answer =
                        restTemplate.getForObject
                                (serviceCall,
                                ClsSafirePrdAnalyticsPredictorWebServiceAnswer.class);

                // TODO
                // Process the answer
                // In this template just print to log
                log.info(answer.toString());
        }
}
```

Class ClsSafirePrdAnalyticsPredictorWebServiceAnswer

This class represents the answer given by the prediction service and *does not need any modifications*, and can be used as-is:

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class ClsSafirePrdAnalyticsPredictorWebServiceAnswer {

   // Represents an automatic counter
   // with the number of times the
   // service has been requested
   private long callCount;

   // Id value passed by the caller
   // Will be returned back as it is
   private long clientId;

   // Topic value passed by the caller
   // Will be returned back as it is
   private String clientTopic;

   // Prediction Model name
   // requested by the caller
   private String modelName;

   // Prediction engine backend
   // requested by the caller
   // Allowed values are: spark, keras
   private String backendName;

   // List of Predicted Data Frame Rows values
   // Contains a JSON list with the Predictied Rows
   // produced by the model. It is responsible
   // of the caller to interprete the meaning of
   // the values
   private String dataFrameRowDataPredictionJSON;

   // Error description
   // when retCode != 0
   private String errorDescription;

   // 0-Success, <>0-Error
   private int retCode;



   public ClsSafirePrdAnalyticsPredictorWebServiceAnswer() {
   }

   public long getCallCount() {
      return callCount;
   }
   public void setCallCount(long callCount) {
      this.callCount = callCount;
   }

   public long getClientId() {
```

```
    return clientId;
}
public void setClientId(long clientId) {
    this.clientId = clientId;
}

public String getClientTopic() {
    return clientTopic;
}
public void setClientTopic(String clientTopic) {
    this.clientTopic = clientTopic;
}

public String getModelName() {
    return modelName;
}
public void setModelName(String modelName) {
    this.modelName = modelName;
}

public String getBackendName() {
    return backendName;
}
public void setBackendName(String backendName) {
    this.backendName = backendName;
}

public String getDataFrameRowDataPredictionJSON() {
    return dataFrameRowDataPredictionJSON;
}
public void setDataFrameRowDataPredictionJSON(String dataFrameRowDataPredictionJSON) {
    this.dataFrameRowDataPredictionJSON = dataFrameRowDataPredictionJSON;
}

public String getErrorDescription() {
    return errorDescription;
}
public void setErrorDescription(String errorDescription) {
    this.errorDescription = errorDescription;
}

public int getRetCode() {
    return retCode;
}
public void setRetCode(int retCode) {
    this.retCode = retCode;
}

@Override
public String toString() {
    return "ClsSafirePrdAnalyticsPredictorWebServiceAnswer {" +
            "callCount = " + Long.toString(callCount) +
            ", clientId = " + Long.toString(clientId) +
            ", clientTopic = " + clientTopic +
            ", modelName = " + modelName +
            ", backendName = " + backendName +
            ", prediction = " + dataFrameRowDataPredictionJSON +
            ", errorDescription = " + ((retCode != 0) ? errorDescription : "Ok") +
```

```
                ", retCode = " + Long.toString(retCode) +
                '}';
        }
    }
```

Speed Specifications

The SAFIRE project aims at real-time processing and therefore Web Service execution time must meet that requirement. However, real-time is a concept relative to the application and the requirements can be different for each application.

For example, in the case of Electrolux boiling detection, real-time means basically the order of one second. Execution speed depends obviously on the connection but also in the predictive model complexity.

As a general requirement, for medium size models and good quality connection, real-time requirement will be understood as execution time in the order of a few seconds.

9.3.3 JSON Message format to access Predictive Analytics Service via MQTT, NiFi, Kafka.

The Predictive Analytics service is also accessible by sending messages to SAFIRE cloud via MQTT, NiFi and KAFKA. A client can also send messages to SAFIRE cloud requesting predictions and waiting for an answers from the cloud. Next two subsections defined the format of the message to be sent and the answer received. Finally, third subsection defines the meaning of the fields.

9.3.3.1 JSON Request Messsage format to be sent to SAFIRE Cloud by the client

Example of message for Electrolux test case representing the case of a hob requesting prediction to check if the water is boiling. The hob sends one row of data with time, energy, coil temp and currents. JSON field names are in green and values in blue.

```
{
    "serviceName"          : "SafirePrdAnalyticsPredictor",
    "timestamp"            : "1517927276069",
    "clientId"             : "hob_2341",
    "clientTopic"          : "boil_detection",
    "modelName"            : "electroluxBoilDetectionNNTraineModelCur.h5",
    "backendName"          : "keras",
    "dataFrameColNames"    :
            ["Time [s]","Energy [kJ]","T_Coil [C]","Cur_F01 [A]","Cur_F02 [A]","Cur_F03
            [A]","Cur_F04 [A]","Cur_F05 [A]","Cur_F06 [A]","Cur_F07 [A]","Cur_F08 [A]","Cur_F09
            [A]","Cur_F10 [A]","Cur_F11 [A]","Cur_F12 [A]","Cur_F13 [A]"],
"dataFrameColTypes"    :
            ["integer","double","double","double","double","double","double","double","doubl
            e","double","double","double","double","double","double"],
```

```
    "dataFrameRowData"      :
            [["0.043058454","453.1684169","116.0166168","64.86247126","61.84273338","58.66685168","54.45028941","50.73850822","46.71570587","44.01422437","40.96934446","38.67995771","37.23618762","30.55048904","25.67852052","21.31361008"]]
}
```


9.3.3.2 JSON Answer Messsage format received by the client from SAFIRE Cloud

Example of message for Electrolux test case representing the answer of the predictive service to the request shown in previous section, in this case telling that the probability of water being boiling is 0.98.


```
{
    "serviceName"                   : "SafirePrdAnalyticsPredictor",
    "timestamp"                     : "1517927276069",
    "clientId"                      : "hob_2341",
    "clientTopic"                   : "boil_detection",
    "dataFrameRowDataPrediction"    : [["0.98"]],
    "errorDescription"              : ""
    "rectCode":                     "0"
}
```


9.3.3.3 Explanation of Fields

This section explains the meaning of fields in request and answer messages.

- "serviceName". Same value sent in Request is received in Answer. Always must be set to "*SafirePrdAnalyticsPredictor*" as this is the name of the service.
- "timestamp". Same value sent in Request is received in Answer. It is the value returned by function now() in the computer when the client requests the service.
- "clientId". Same value sent in Request is received in Answer. An identificator of the hob that is calling the service.
- "clientTopic". Same value sent in Request is received in Answer. The message will be sent to this topic.
- "modelName". Trained machine learning model to be invoked for prediction.
- "backendName". Machine learning module backend to be invoked, must be keywords "*spark*" or "*keras*".
- "dataFrameColNames". List ([]) of data field names provided by the request.

- "dataFrameColTypes". List ([]) of data field types provided by the request. Must be keywords "*integer*", "*double*", "*string*", "*arrayDouble*" or "*arrayInteger*".

- "dataFrameRowData". List of rows ([[]]) of data field values provided by the request. Prediction service will answer with a list of predictions, one prediction per row (a prediction may be composed by several values). In the example above there is only one row, by several rows can be provided.

- "dataFrameRowDataPrediction". List of rows ([[]]) of predicted values. There is one list of predicted values for each row. In the example above, as there is only one row, only one list of predicted value is received. In this case that list contains only one value that represents the probability of being boiling.

- "errorDescription". Description of the error returned by the service when no prediction is returned (retCode <> 0).

- "retCode". 0 if the service success and predicts, and <> 0 otherwise and no prediction was produced.