2018 IEEE International Conference on Big Data

# A Generic and Scalable Pipeline for Large-Scale Analytics of Continuous Aircraft Engine Data

Florent Forest
Safran Aircraft Engines
77550 Moissy-Cramayel, France
florent.forest@safrangroup.com

Jérôme Lacaille
Safran Aircraft Engines
77550 Moissy-Cramayel, France
jerome.lacaille@safrangroup.com

Mustapha Lebbah
LIPN, Université Paris 13
93430 Villetaneuse, France
lebbah@lipn.univ-paris13.fr

Hanene Azzag
LIPN, Université Paris 13
93430 Villetaneuse, France
hanene.azzag@lipn.univ-paris13.fr

***Abstract*—A major application of data analytics for aircraft engine manufacturers is engine health monitoring, which consists in improving availability and operation of engines by leveraging operational data and past events. Traditional tools can no longer handle the increasing volume and velocity of data collected on modern aircraft. We propose a generic and scalable pipeline for large-scale analytics of operational data from a recent type of aircraft engine, oriented towards health monitoring applications. Based on Hadoop and Spark, our approach enables domain experts to scale their algorithms and extract features from tens of thousands of flights stored on a cluster. All computations are performed using the Spark framework, however custom functions and algorithms can be integrated without knowledge of distributed programming. Unsupervised learning algorithms are integrated for clustering and dimensionality reduction of the flight features, in order to allow efficient visualization and interpretation through a dedicated web application. The use case guiding our work is a methodology for engine fleet monitoring with a self-organizing map. Finally, this pipeline is meant to be end-to-end, fully customizable and ready for use in an industrial setting.**

***Big data; aviation; aircraft engine; health monitoring; hadoop; spark; scalable; generic***

## I. Introduction

Nowadays, aviation industry and aircraft operation generate growing amounts of data that can be leveraged for various applications [1]–[3]. For engine manufacturers, an important one is engine health monitoring (EHM). The general aim is to improve availability and operation of engines [4], [5]. It consists in monitoring the state of an engine or a fleet of engines by using operational data and past events. The first objective is to avoid abnormal events as in-flight shutdowns, aborted take-offs and delays and cancellation. The second objective is optimizing maintenance operations to improve safety while reducing costs for manufacturers and airline companies. Data-driven maintenance enables to detect faults and prevent failures before they happen, extending the life span of systems and reducing costs [6]. From the operations point of view, data analytics are already used for fuel consumption and route optimization [7]. Aircraft manufacturers provide customized data-driven services to airlines [8]. Engine manufacturers also provide such services, allowing client airlines to manage their operational data and use data analysis in the objective of improving safety, maintenance and reducing fuel consumption [9]. With the growth of air traffic, the volumes of data to be processed are growing exponentially and can no longer be handled in traditional ways: as an example, recent aircraft are equipped with tens of thousands of sensors and generate several terabytes of data per day [10].

In response to the need for processing and analyzing the petabytes of information generated by billions of users at the internet era, companies developed solutions to process large volumes of data spread across clusters of machines, relying on a simple but efficient and cost-effective paradigm called map-reduce [11]. Map-reduce is at the core of Apache Hadoop, the most used open-source platform for distributed processing of large datasets [12]. Choosing the adequate big data infrastructure and software tools for storage, analysis and visualization is a major challenge also for actors in more traditional industries as aerospace [13], [14]. The Hadoop platform was chosen by a large number of companies, but its utilization is not straightforward for non-trained engineers. In traditional industry, the domain knowledge is held by engineers who often have sufficient programming skills to implement algorithms in some programming language (e.g. Python) and run them on a small to moderately large, local, dataset (e.g. a CSV file), but are not acquainted with the big data tools and programming frameworks that are needed to efficiently query and process large, distributed datasets stored on a cluster.

Our objective is to design and implement a generic and scalable processing pipeline to power health monitoring applications based on continuous operational aircraft engine data. The goal is not to present new algorithms, nor to focus on health monitoring, but rather to present a completely operational pipeline based on open-source software tools from the Hadoop ecosystem and ready to use in an industrial setting. Our main contributions are:

- Processing and analyzing real, large-scale, industrial data sets coming from thousands of operating aircraft engines
- Combining recent programming techniques, open-source technologies and a big data infrastructure, which is not yet common in traditional industries such as the aerospace industry and engine manufacturing industry in particular
- Focusing on genericity and allowing domain engineers (who are not software engineers or data scientists) to deploy their domain-specific computations and algorithms in an agile way
- Demonstrating how scalable unsupervised learning algorithms can be integrated and used for visualization in engine health monitoring applications

The next section presents previous related work dealing with health monitoring of industrial systems using data analysis, and large-scale analytics in aerospace industry, two topics that are combined in our work. Then, the third section presents our data and their main properties, acquisition process, and storage, in order to set the context of our work. Section IV details each component of the pipeline along with their requirements and motivations. Our guiding thread throughout the development of the pipeline will be the engine health monitoring (EHM) use case using a self-organizing map (SOM) described in [15]–[18], which we will refer to as the SOM-EHM use case. However, the range of possible applications is much wider.

## II. Related Work

Data analysis and machine learning have already been used extensively for health monitoring applications across all industries. In particular, we will present methods based on clustering and visualization with the SOM algorithm in the first paragraph. However, these approaches do not address the issue of large-scale processing of large volumes of data. In recent years, much effort has been put into designing and building big data architectures by aerospace industry actors, and some examples are presented in the second paragraph.

### A. *Health Monitoring using SOM*

Kohonen's self-organizing map (SOM) [19], [20], the most well-known self-organizing clustering algorithm, is a useful tool to survey and visualize high-dimensional datasets [21]. It maps the data space onto a two-dimensional grid preserving the topology of the original data space. The approaches presented by [15]–[18] use this algorithm to visualize the state of a fleet of aircraft engines. In [17], their data contains 20 variables (15 context variables and 5 engine variables) measured on a fleet of 91 engines during approximately one year. Their methodology consists in four modules: environmental condition normalization, changes detection, SOM, and a search module based on edit distance to find similar engine trajectories (a trajectory is the name given to the sequence of self-organizing map units corresponding to successive flights of an engine). After normalization w.r.t. the context variables and removal of abrupt changes and slow trends, the residuals of the engine variables are used to fit a SOM model. It is shown that such a model is useful to monitor the state of an engine, its evolution flight after flight (trajectory), and detect potential faults and deterioration of engine parts.

We aim at reproducing this methodology with our analytics pipeline, with some major differences. First, the previous approach processed a small dataset using Matlab, while our data is large-scale and stored on a cluster, and is processed using distributed software. Second, their variables are scalar values for each flight (*snapshot* data), unlike ours which are time series and require preprocessing and feature extraction. Finally, we focus on genericity, customization and ease-of-use, in order to solve not just the SOM-EHM use case but a wide range of applications. Note that we have not implemented the change detection algorithms and the search module at the time of this writing.

In [22], the authors used SOM to monitor vehicle cooling systems and detect anomalies by using a distance metric between two SOM models.

### B. *Big Data Architectures in Aerospace Industry*

Reference [13] present a big data architecture for storing and analyzing operational and repair data at Honeywell. Their operational data come from multiple sources and supports, for example snapshot and summary data from the ACMS (Aircraft Condition Monitoring Systems). This typically includes parameters as speed, position, altitude and exhaust temperature at specific flight phases. Repair data consists in shop visit reports. They use a Hadoop cluster and store data in HDFS. For analytics, they use R along with packages to interact with Hadoop, and also provide analytics as a service through either an R Shiny web application, or an OpenCPU RESTful service for interacting directly with R. Unlike their approach, we will use the Spark framework for distributed computing.

Reference [23] from Boeing use a big data architecture to analyze near real-time ASDI (Aircraft Situation Display to Industry) data. This data is a feed of XML messages, which have to be translated into a relational database. Their infrastructure relies on IBM software for message brokering, database management (DB2), processing and business intelligence. They present their use case and some optimizations for near real-time processing. Their data is quite different from our continuous operational data, and we use a Hadoop ecosystem.

## III. Continuous Operational Engine Data

### A. *Presentation*

Our work focuses on continuous operational engine data (CEOD) that are recorded on recent aircraft. CEOD are a data stream composed of several hundreds of parameters measured during whole flights. Due to the large number of parameters recorded at high frequencies, these data contain much more information compared to other types of aircraft datasets, for instance *snapshot* data, which correspond to a set of parameters recorded during a short time interval (between 3 and 30 seconds) at key flight phases (e.g. during take-off, climb or cruise) and sent back to the ground near real-time. This leads to large volumes that can no longer be processed in traditional ways. For now, due to bandwidth limitations, continuous data are offloaded post-flight. In future, this data may be streamed in real-time, allowing for live predictive analytics.

Concretely, each flight (identified by its *flight_id*) consists in a set of flight parameters (*param*) measured at given timestamps (*time*). The data used in our experiments concern a fleet of engines of the same type, identified by their engine serial numbers (*esn*). As a consequence, CEOD are a set of univariate time series. Because the flight recorders dynamically adapt their sampling frequency, time series can have very different frequencies, and they can vary throughout the flight. See Fig. 1 for the schema specification and Tab. I for a small data sample.

```
flight_id: string
esn: string
param: string
time: timestamp
value: float
```

Fig. 1. Schema of CEOD

In this format, observations are stored row-wise, i.e. one row per flight, per parameter and per time step at which a value was recorded for this parameter. The main advantages of this format to represent CEOD time series are:

- The format is fixed and robust to evolutions in the data: in particular, parameter names are expected to change (e.g. with new versions of engines or flight recorders)

- It handles variable frequencies: some parameters have a high frequency, whereas others only have few measurements throughout the flight

A summary of dataset properties is indicated in Tab. II and illustrates the fact that we already are in a high volume context, keeping in mind that we do not have access yet to the data produced by all engines operated daily. The volumes are expected to grow enormously in the coming years. Health monitoring must provide insights on the incoming data quickly (and in a not so distant future, in real-time), thus we also have a high velocity. Altogether, the processing of CEOD for health monitoring requires the use of "Big Data" methods and tools.

Tab. I CEOD SAMPLE

| flight_id | esn | param | time | value |
|---|---|---|---|---|
| 542021_0 | 542021 | IFV_TAMB | 07-14-18 11:03:42.005 | 26.495 |
| 542021_0 | 542021 | IFV_TAMB | 07-14-18 11:03:45.005 | 26.993 |
| 542021_2 | 542021 | IFV_TAMB | 07-15-18 22:13:42.000 | 26.245 |

Tab. II DATASET PROPERTIES

| Property | Approximate value |
|---|---|
| Number of flights | 10 000 |
| Number of parameters | 500 |
| Recording frequency | < 1Hz up to 60 Hz |
| Hive table lines | 200 billion |
| Storage volume (compressed) | 700 GB |

## B. Acquisition Process

We quickly present the current acquisition process of CEOD. Our work begins once the data is stored on the cluster, at the end of following steps:

1. Manually download raw data from aircraft flight recorder. Raw data contain concatenated recordings of several flights, depending on the time since last download.
2. Decode raw data into a structured file format using a proprietary software.
3. Cut the files into separated flights and ingest the data into the cluster.

## C. Storage

CEOD are stored and processed on a Hadoop cluster, in order to benefit from the scalability and fault-tolerance of the Hadoop ecosystem. As we have large volumes of structured data, it is stored on the Hadoop Distributed File System (HDFS) using the Hive data warehouse [24] in ORC format (Optimized Record Columnar File) [25], [26].

## IV. ANALYTICS PIPELINE

This section details the components of our processing pipeline. An overview of the different steps is represented as a diagram in Fig. 2.

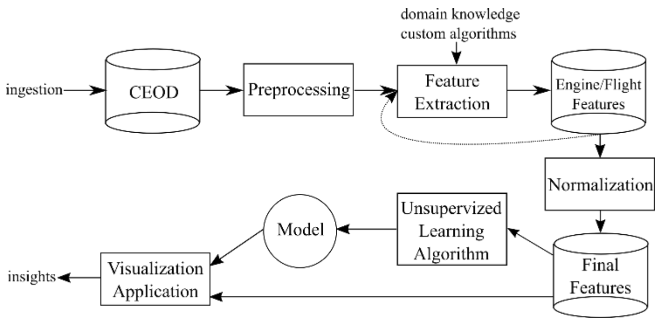

Fig. 2. Diagram representing the analytics pipeline

Our technological stack is briefly exposed in paragraph A. Paragraph B explains the preprocessing of CEOD. Then, paragraph C presents our methodology for computing features in a generic way. The features are then preprocessed to obtain a column-wise format adapted to analysis and machine learning (paragraph D). A normalization step, described in paragraph E, is required, before feeding the variables into an algorithm (paragraph F). Preprocessed data and features can be persisted to disk into Hive tables to be re-used, and each of these steps can be executed independently. Finally in paragraph G, we present the application developed to visualize model results.

## A. Technological Stack

Data is stored on a Hadoop cluster using Hive. For computations, we use the Apache Spark distributed processing engine [27]. In particular, we use Spark SQL and the Dataset/Dataframe API. The Spark jobs are written either in Scala or in Python (using the pyspark wrapper). For linear algebra, numerical analysis and machine learning, we used Spark ML, as well as Python and Scala libraries (numpy, scipy, pandas, scikit-learn, breeze, smile). Finally, for visualization, we developed an HTML/JavaScript application with the D3.js library, as well as a minimalistic Python server with Flask [28] to interact with Hive.

## B. CEOD Preprocessing

The computations carried on during the feature extraction step will need to process the whole time series of a flight parameter, so it is not necessary to have one row per time step in our Spark Dataframe: the smallest entity to be processed in parallel is a parameter during an entire flight. Thus, we aggregate the CEOD on the time dimension into vectors of timestamps and values, using Spark SQL's *collect_list* aggregation. This drastically reduces the number of rows (from approximately 200 billion to 5 million rows) and accelerates subsequent processing. It only needs to be executed once on new incoming data. See Fig. 3 for the resulting schema.

```
flight_id: string            time: array<timestamp>
esn: string                  value: array<float>
param: string
```

Fig. 3. Schema of preprocessed CEOD

### C. Generic Feature Extraction

The first step in engine health monitoring is to choose and compute a set of features that represent the health state of an engine or engine sub-system (for example fuel, oil or control system) at a given flight. Such features can be as simple as the value of a parameter at a specific instant of the flight but they may be more complex features that were engineered by domain experts.

Computing these features is an essential step of our processing pipeline, as they will be the input for the subsequent algorithms and define what aspects of the engine health will be monitored. The main requirements of the feature extraction step are:

1. **Scalability.** Features are extracted in a distributed, data-parallel way, allowing to process a huge number of flights and engines in parallel.
2. **Genericity**. It is possible to use generic functions and already existing algorithms to compute features.
3. **Ease-of-use.** Engineers should be able to implement and use their own feature extraction algorithms without any knowledge of Hadoop cluster architecture and distributed computing with Spark.

Requirements 2 and 3 come from the observation that scaling and deploying algorithms created by domain experts to production is not straightforward. Since every engineer cannot be trained to these technologies (which is not their core profession and would require months if not years of experience), it is necessary to go through a long and costly industrial deployment step, often requiring to rewrite the algorithm from scratch using different programming techniques. Here, we take an alternative path: users only have to write their core algorithm as a generic function using a compatible programming language (in our case, Python or Scala), plug it into the pipeline, and it will be executed in parallel by the Spark engine, transparently for the user. The user's custom function has to follow a standard input and output schema. The user also specifies what entities are processed in parallel: for example, one might want to run a function on all flights in parallel in order to compute features for each flight. The same can be done on different levels (e.g. all flights for a given engine, a sliding window of flights, etc.), defining the parallelism. This allows an agile work flow where engineers can quickly deploy their computations.

Our feature extraction API supports three types of functions:

1. **Native Spark code**, for users familiar with Spark and seeking optimal performance. Some commonly used feature extraction functions are already implemented and ready to use. For instance, for simple operations like aggregations, it is not efficient to write a dedicated custom function.
2. **Custom functions**, written in local (i.e. non-distributed) Python or Scala code, allowing to flexibly define any kind of algorithm. Common numerical, data analysis and machine learning libraries can be used. Parallelism is achieved by using Spark's *group by* and *map* or *apply* operations.
3. **Python modules**. Legacy algorithms are already packaged as Python modules, so an API allows to import them and act as an interface to use them the same way as custom functions.

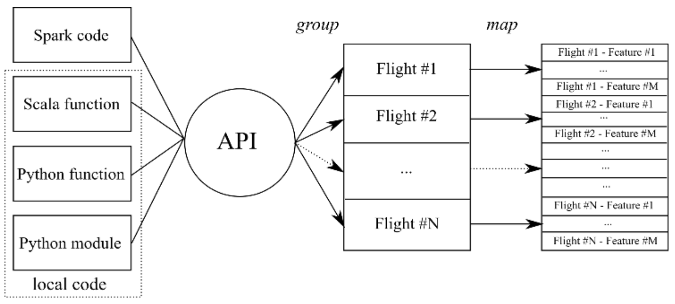

Fig. 4. Diagram of the API for distributed computation of flight features

As Spark practitioners know, these custom functions are considered as black boxes and not optimized by Spark's execution engine, but it is the only solution allowing to express any processing logic (which may rely on external libraries) and without knowledge of distributed programming. However, when a custom algorithm is in its final version and ought to be used regularly in an industrial workflow, it can be rewritten in native code and optimized. Custom functions are applied by grouping on some key or sliding window, and using a *map*-like operation. Note that we use the most efficient Spark API and associated *group*/*map* methods whenever possible: Dataset API for Scala functions, Dataframe API with pandas vectorized UDFs or RDD API for Python functions. Fig. 4 illustrates our generic feature extraction API for the flight-parallel case.

For the SOM-EHM use case, we used 5 simple flight features consisting in the values of 5 variables at a specific instant of the take-off phase. They were calculated by grouping the data per flight and applying a custom Scala function with the *flatMap* operator. The variables are presented in Tab. III.

Tab. III FEATURE VARIABLES USED IN OUR EXPERIMENT

| Name | Description | Type |
|---|---|---|
| temp | Ambient air temperature | Context |
| N1 | Fan speed | Context |
| N2 | Core speed | Engine |
| fuelflow | Fuel flow | Engine |
| EGT | Exhaust gas temperature | Engine |

Features are stored using a format almost identical to the CEOD, only renaming the *param* column to *feature* and adding two columns for metadata: *computed_on*, a timestamp indicating when the feature was computed, and *author*, indicating the ID of the user (see Fig. 5). This additional information is useful to manage different versions of the features computed by different users. The *time* column is kept even if we no longer store time series, because engineers find it useful to associate a timestamp to the feature (e.g. for a maximum value, we also keep the timestamp at which the maximum was reached during the flight).

```
flight_id: string          value: float
esn: string                computed_on: timestamp
feature: string            author: string
time: timestamp
```

Fig. 5. Schema of flight features

As already mentioned, this fixed-schema, row-wise format is practical due to its robustness and flexibility, allowing to easily append new features to an existing table. As a consequence, it allows to calculate features in several passes, by executing the feature extraction step on the output of the previous pass: this enables, for example, to compute more complex features from more basic ones (hence the loopback at the feature extraction step in Fig. 2).

## D. *Feature Preprocessing*

While the previously used row-wise format is flexible and robust, it is not adapted to analysis and fitting machine learning models, where the norm is to represent features as a matrix with the columns containing the features and the rows containing the individuals. Thus, we select the final features and apply a pivot operation on the table, producing the desired output (see Tab. IV).

Tab. IV FINAL FLIGHT FEATURES SAMPLE

| **flight_id** | **esn** | **N2_takeoff** | **fuelflow_takeoff** | **…** |
|---|---|---|---|---|
| 542021_0 | 542021 | 27.059 | 4.957 | … |
| 542021_1 | 542021 | 25.551 | 4.975 | … |
| 542022_0 | 542022 | 96.759 | 5437.73 | … |

## E. *Context and Environment Normalization*

In order to compare flights together, it is necessary to remove the effects of the context and environment of the flight. For example, is has no sense to compare engine state variables if the aircraft has been operated on different routes, in different regions with all different ambient air temperatures, altitudes, speeds, utilizations of the engine, etc. Thus, the features computed at step B are divided into two categories:

- Engine state variables, which will be the inputs of the clustering and visualization algorithms
- Context or environment variables

The types of the variables used in our experiment are indicated in the last column of Tab. III.

The simplest approach to normalization is using a linear regression model and calculating the residuals of the state variables w.r.t. the context variables, for instance using the LASSO [29] method as is presented in [16], [17]. This removes linear dependencies. The model can be expressed as follows, in its simplest form:

$$\begin{aligned} Y_r^{(i)} &= \mu_r + \beta_r^1 X_1^{(i)} + \cdots + \beta_r^q X_q^{(i)} + \epsilon_r^{(i)} \\ &= \mu_r + \beta_r^T X^{(i)} + \epsilon_r^{(i)} \end{aligned}$$

for each engine state variable *r* and each flight *i*, where $\mu_r$ is the intercept and $\beta_r$ are the regression coefficients for each state variable w.r.t. the context variables 1…q. In practice, additional terms must be taken into account, as for instance the engine effect representing individual variations between different engines.

To fit the LASSO model, we used Spark's ML machine learning library. The residuals of the linear model are then used

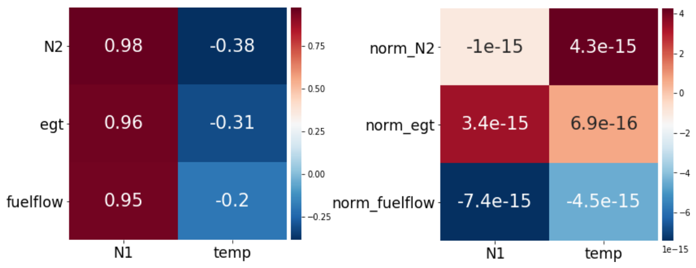

Fig. 6. Pearson correlations between context and engine variables (left) Raw engine variables. (right) Normalized engine variables.

as normalized variables, and will be the input features of the clustering and visualization models in the next step of the pipeline. Fig. 6 shows that the linear model has effectively removed the strong linear correlations between context and engine variables.

Another normalization approach is taken in [30], using a MPPCA model (mixture of probabilistic PCA). This is a typical example of complex mathematical manipulations that cannot be done with available Spark ML functions, but can be implemented as a custom *map* function that calls local numerical analysis libraries (in the same way as for feature extraction). It was not integrated into our experiment.

## F. *Clustering and Visualization Algorithm*

The next step of our pipeline is visualizing the previously calculated features. As the number of features describing an engine (or any other system) is usually larger than two, this requires a dimensionality reduction step, using unsupervised learning algorithms. In particular, we focus on a class of models called *self-organizing models* that achieve simultaneous clustering and visualization of high-dimensional datasets. The most famous and used algorithm of this class is Kohonen's self-organizing map (SOM) [19], [20]. In order to test our analytics pipeline on the SOM-EHM use case, SOM is the first model we integrated.

We used a Spark implementation of the SOM algorithm [31], part of the C4E project[1] [32]. Here again, genericity is a strong requirement, thus we also integrated the k-means algorithm, to demonstrate that any centroid-based clustering model could be plugged into the pipeline. For example, SOM could be replaced by a GMM (Gaussian Mixture Model) or another self-organizing clustering model (e.g. probabilistic maps like GTM [33], PRSOM [34], maps for mixed continuous/categorical data [35], etc.), as long as it is scalable w.r.t. the size of the data, to meet our scalability requirement. Supervised algorithms could also be used, but their output will not be adapted to the visualization tool presented in paragraph *F*. Fig. 7 shows the resulting SOM, representing the values of our three normalized engine variables. Interpreting such a map is the role of domain experts and not our aim here, moreover, the hyper-parameters of the normalization (regularization) and of the SOM model have not been tuned at all. Still, we can mention a strong correlation between the *N2* and *egt* variables.

The resulting models are serialized to simple JSON files (see Fig. 8) containing metadata (including the name of the Hive table containing the training set) and the prototype vectors along with

[1] https://github.com/Clustering4Ever/Clustering4Ever

their cardinalities (number of training points belonging to each cluster).

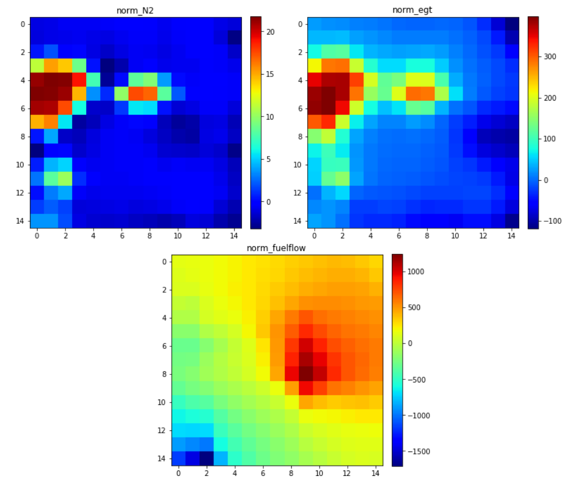

Fig. 7. Normalized engine variables represented on the resulting SOM

```
{
  "model": "SOM",
  "params": {"height": 15, "width": 15,
             "topology": "rectangular"}
  "date": "2018-07-12T13:52:46.0",
  "train_data": "db.table_train",
  "cardinalities": [74, 127, 119, …],
  "prototypes": [[-0.96, 25.37, 40.82], …]
}
```

Fig. 8. Example of SOM Model Metadata

## G. *Visualization*

The last step of our analytics pipeline is a web interface developed specifically for displaying the results of centroid-based clustering algorithms such as k-means and SOM. The application is developed in HTML/JavaScript using the D3.js library. It can be easily accessed via a browser, which is adapted for widespread use in a company.

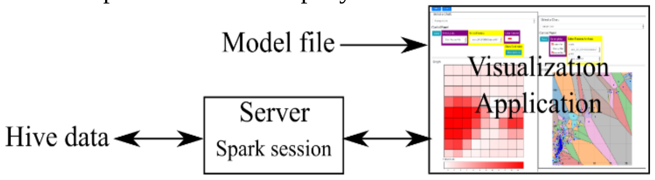

Fig. 9. Architecture of the Visualization Application

Its main functionalities are:

- Providing a summary of the cluster centroids (also called prototype vectors): feature values, cardinalities, basic statistical properties of each cluster.
- Visualizing the self-organizing map, by displaying the previous quantities on the topology-preserving grid, as well as engine trajectories.
- Projecting and visualizing a sample of the training set along with the model prototypes on a scatter plot, density plot or a Voronoï tessellation using PCA or t-SNE [36] to reduce the dimensions to 2D

The user interface takes as input the model file output at the previous step. The model selection process is still manual but will be improved in future work. Along with the application, a Python server interacts with Hive. When the users visualize the training data using PCA or t-SNE, the application calls the server that in turn queries a data sample from the training set. The name of this table is part of the model metadata. The whole architecture is summarized in Fig. 9 and a screenshot is provided in Fig. 10.

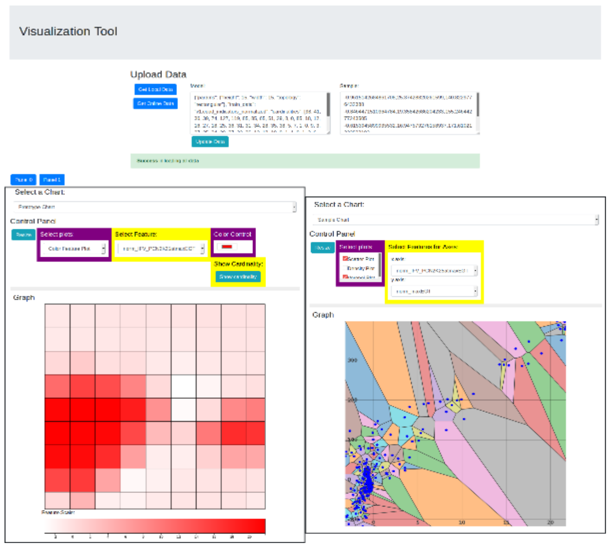

Fig. 10. Screenshot of the visualization application showing feature values on a SOM and a Voronoï tessellation

## H. *Creator and Consumer Utilizations*

This analytics pipeline can be used by two target classes of users: (1) *Creators*: engineers who will adapt and create their own processing. They select the data (fleets, engines, flights, variables) and features relevant to them (i.e. relevant to the part of the engine they want to study), implement and deploy their own algorithms if needed, execute the processing and collect the resulting features for their own utilization. They may also select and tune the learning algorithm, and visualize the results to obtain insights. In a nutshell, they may intervene at any step of the pipeline, so genericity is crucial; (2) *Consumers:* users who will not modify the pipeline, but rather consume already computed results for a specific health monitoring task, and interpret them to inform decision-making. They use the visualization application to observe results that are calculated automatically by a "frozen" version of the pipeline, where the features and algorithm are fixed and tuned for a specific use case. When new data from recent flights are ingested, the pipeline must run automatically in order to show up-to-date results, using for example a workflow manager to schedule the jobs. For these users, it is crucial that the web application is always available, shows relevant and up-to-date results and that the user interface is adapted to their use case. Specific versions of the visualization application might be needed to satisfy all needs.

## V. Conclusion and Future Work

We have designed and built an end-to-end pipeline for large-scale analytics of continuous operational aircraft engine data collected on a modern type of aircraft engine, and stored on a Hadoop cluster. Using this pipeline, we implemented a minimal but fully operational version of a health monitoring application. Each step of the processing scales to large datasets thanks to map-reduce data-parallel processing. Moreover, each step is kept generic and customizable in order to adapt to a wide range of use cases. The pipeline integrates an unsupervised learning algorithm (in the present use case, a self-organizing map) and a visualization web application to analyze the results. We also enable domain engineers with no big data skills to deploy their algorithms at scale, using a simple API for custom functions.

The next step is to automate the pipeline using workflow management software, for instance Oozie [37], Airflow [38] or Azkaban [39], that allow to author workflows that schedule jobs sequentially, in parallel or based on conditions and triggers (e.g. availability of new data). Another important future development is data and model versioning. As the database is constantly growing with incoming flights, it is essential to keep track of what data was used to fit a model. Thus, the application should handle multiple models and compare them in order to analyze their evolution. Generally, efficient data and model versioning capability should be integrated into the pipeline.

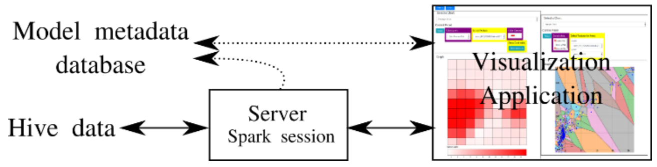

Fig. 11. Architecture of the Visualization Application with Metadata Database

Finally, model metadata should be stored in a database management system (DBMS). The server will interact with this database to fetch model metadata, as illustrated in Fig. 11.

## References

[1] R. Akerkar, "Analytics on big aviation data: Turning data into insights," *Int. J. Comput. Sci. Appl.*, vol. 11, no. 3, pp. 116–127, 2014.
[2] M. T. Akpinar and M. E. Karabacak, "Data mining applications in civil aviation sector: State-of-art review," *CEUR Workshop Proc.*, vol. 1852, pp. 18–25, 2017.
[3] C.-G. Oh, "Application of Big Data Systems To Aviation and Aerospace Fields ; Pertinent Human Factors Considerati ....," in *International Symposium on Aviation Psychology*, 2017, no. May.
[4] S. Blanchard, M. Cottrell, and J. Lacaille, "Health monitoring des moteurs d'avions," in *Les entretients de Toulouse*, 2009.
[5] G. Bastard, J. Lacaille, J. Coupard, and Y. Stouky, "Engine Health Management in Safran Aircraft Engines," in *PHM Society*, 2016.
[6] Y. O. Lee, J. Jo, and J. Hwang, "Application of Deep Neural Network and Generative Adversarial Network to Industrial Maintenance : A Case Study of Induction Motor Fault Detection," *2017 IEEE Int. Conf.*, vol. 7, no. 2, pp. 3248–3253, 2017.
[7] E. Kasturi, S. Prasanna Devi, S. Vinu Kiran, and S. Manivannan, "Airline Route Profitability Analysis and Optimization Using BIG DATA Analytics on Aviation Data Sets under Heuristic Techniques," *Procedia Comput. Sci.*, vol. 87, pp. 86–92, 2016.
[8] Airbus, "Airbus' open aviation data platform Skywise continues to gain market traction," 2018. [Online]. Available: http://www.airbus.com/newsroom/press-releases/en/2018/02/airbus--open-aviation-data-platform-skywise-continues-to-gain-ma.html. [Accessed: 11-Jun-2018].
[9] Safran, "Cassiopée," 2018. [Online]. Available: https://www.cassiopee.aero/. [Accessed: 27-Jul-2018].
[10] B. Marr, "That's Data Science: Airbus Puts 10,000 Sensors in Every Single Wing!," 2015. [Online]. Available: https://www.datasciencecentral.com/profiles/blogs/that-s-data-science-airbus-puts-10-000-sensors-in-every-single. [Accessed: 10-Jun-2018].
[11] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," 2008.
[12] Apache Hadoop, "Hadoop Project," 2011. [Online]. Available: http://hadoop.apache.org/. [Accessed: 02-Aug-2018].
[13] A. Murugan, D. Mylaraswamy, B. Xu, and P. Dietrich, "Big Data Infrastructure for Aviation Data Analytics," *2014 IEEE Int. Conf. Cloud Comput. Emerg. Mark.*, pp. 1–6, 2014.
[14] S. Li, Y. Yang, L. Yang, H. Su, G. Zhang, and J. Wang, "Civil Aircraft Big Data Platform," *2017 IEEE 11th Int. Conf. Semant. Comput.*, pp. 328–333, 2017.
[15] M. Cottrell *et al.*, "Fault prediction in aircraft engines using Self-Organizing Maps," *WSOM*, 2009.
[16] E. Côme, M. Cottrell, M. Verleysen, and J. Lacaille, "Aircraft engine health monitoring using Self-Organizing Maps," *Ind. Conf. Data Min.*, 2010.
[17] E. Côme, M. Cottrell, M. Verleysen, and J. Lacaille, "Aircraft engine fleet monitoring using Self-Organizing Maps and Edit Distance," *Mach. Learn.*, pp. 298–307, 2011.
[18] J. Lacaille and E. Côme, "Visual mining and statistics for a turbofan engine fleet," in *IEEE Aerospace Conference Proceedings*, 2011.
[19] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biol. Cybern.*, vol. 43, no. 1, pp. 59–69, 1982.
[20] T. Kohonen, "The Self-Organizing Map," in *Proceedings of the IEEE*, 1990, vol. 78, no. 9, pp. 1464–1480.
[21] J. Vesanto, "SOM-based data visualization methods," *Intell. Data Anal.*, vol. 3, no. 2, pp. 111–126, 1999.
[22] M. Svensson, S. Byttner, and T. Rögnvaldsson, "Self-organizing maps for automatic fault detection in a vehicle cooling system," *2008 4th Int. IEEE Conf. Intell. Syst. IS 2008*, vol. 2, no. October, pp. 248–2412, 2008.
[23] S. Ayhan, J. Pesce, P. Comitz, D. Sweet, S. Bliesner, and G. Gerberick, "Predictive analytics with aviation big data," *Integr. Commun. Navig. Surveill. Conf. ICNS*, no. January 2016, 2013.
[24] Apache Hive, "Hive Project," 2010. [Online]. Available: http://hive.apache.org/. [Accessed: 13-Jul-2018].
[25] Apache ORC, "ORC Project," 2015. [Online]. Available: https://orc.apache.org/. [Accessed: 31-Jul-2018].
[26] Y. He *et al.*, "RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems," in *International Conference on Data Engineering*, 2011, pp. 1199–1208.
[27] Apache Spark, "Spark Project," 2014. [Online]. Available: https://spark.apache.org/. [Accessed: 11-Jul-2018].
[28] Flask, "Flask Project," 2010. [Online]. Available: http://flask.pocoo.org/.
[29] R. Tibshirani, "Regression Shrinkage and Selection via the Lasso," *J. R. Stat. Soc. Ser. B*, no. 58, pp. 267–288, 1996.
[30] J. Lacaille, A. Bellas, C. Bouveyron, and M. Cottrell, "Online Normalization Algorithm for Engine Turbofan Monitoring," pp. 1–8.
[31] T. Sarazin, H. Azzag, and M. Lebbah, "SOM clustering using spark-MapReduce," *Proc. Int. Parallel Distrib. Process. Symp. IPDPS*, pp. 1727–1734, 2014.
[32] Paris 13 University - LIPN UMR CNRS 7030, "C4E." 2018.
[33] M. Svensen, C. M. Bishop, and C. K. I. Williams, "GTM: The Generative Topographic Mapping," *Neural Comput.*, no. April, 1997.
[34] F. Anouar, F. Badran, and S. Thiria, "Probabilistic self-organizing map and radial basis function networks," *Neurocomputing*, vol. 20, no. 1–3, pp. 83–96, 1998.
[35] M. Lebbah and A. Chazottes, "Mixed Topological Map," *Neural Networks*, no. April, pp. 27–29, 2005.
[36] L. J. P. Van Der Maaten and G. E. Hinton, "Visualizing high-dimensional data using t-sne," *J. Mach. Learn. Res.*, vol. 9, pp. 2579–2605, 2008.
[37] Apache Oozie, "Oozie Project," 2011. [Online]. Available: http://oozie.apache.org/. [Accessed: 06-Aug-2018].
[38] M. Beauchemin, "Airflow: a workflow management platform," *Airbnb Engineering & Data Science Blog*, 2015. [Online]. Available: https://medium.com/airbnb-engineering/airflow-a-workflow-management-platform-46318b977fd8. [Accessed: 12-Jun-2018].
[39] Azkaban, "Azkaban Project." [Online]. Available: https://azkaban.github.io/. [Accessed: 06-Aug-2018].