STREAMER:
Data Stream Framework

User Guide

SANDRA GARCIA RODRIGUEZ
MOHAMMAD ALSHAER

Table des matières

1. **Introduction** 3
2. **Architecture** 4
3. **Data Stream Ingestion** 7
4. **DSPE Processor** 9
5. **Learning API + Algs. Adapter** 12
6. **Application & Evaluation** 18
7. **Visualization** 18
8. **Learning API Connector** 22

1. Introduction

With the advance of technology, the last decade saw the rise of applications generating a large quantity of data in a continuous way. This comes with a proliferation of solutions for the collection and analysis of such *data streams* for various purposes. Learning from this unbounded and infinite arrival of data is crucial. Nevertheless, since storage devices are limited, classic big data approaches are not suitable for a full knowledge extraction since they require, as batch-processing oriented methods, the entire historical database. Thus, the challenge about how to overcome this limitation emerges. There is a need for powerful analysis tools that are able to process and learn from continuous data streams without storing them. Having efficient online models is key, and to achieve this, data scientists need to easily test their algorithms in streaming contexts.

Already existing tools do not always facilitate this task since they are complex, heavy in terms of computational cost, and not usually flexible enough to be customized for user's necessities. For instance, *Kafa/Kafka streams* handle balancing the processing load, maintaining the local state for tables and failure recovery[i]. Besides, it requires implementing the streaming pipeline from scratch (code, interactions with algorithms, etc.). *Spark Streaming* and *Sonora* provide a limited operator space to user-customized codes, whereas *Flink* can deal effectively with complex stream computations and batch-processing tasks, but it faces scaling limitations. *Samza* simplifies many stream processing operations and offers low latency performance. However, it does not provide accurate recovery of aggregated state (counts, average, etc.) in the event of failure since data might be delivered more than once. *SAMOA* offers the required tools for applying the machine learning algorithms in distributed streams, but it does not provide any graphical interface and does not support all operating systems[ii]. Overall, there exist various technologies operating over data streams that comply with some of the mentioned key features but do not meet them all together.

To deal with the aforementioned challenge we propose STREAMER, a cross-platform framework that helps data scientists to easily integrate machine learning algorithms in realistic streaming operational contexts. Testing algorithms is straightforward and can be completed in three steps:

(1) Define the data format to work with
(2) Implement or call the algorithm to be tested
(3) Set up the streaming context through the properties files.

Furthermore, STREAMER also satisfies all mentioned key features: scalability -ability to scale across a cluster-, fault-tolerance -guarantee the delivery despite any fault-, availability -data is accessible and operational at any moment-, and flexibility -user can adapt new algorithms and applications-.

STREAMER counts with a java documentation in HTML format, generated from the source code, with the description of all classes, attributes and methods placed in *javadoc* folder.

2. Architecture

STREAMER implements the whole chain of data stream collection, processing, knowledge extraction and visualization. It also offers several ways of simulating data streams ingestion from one or more sources (replacing real-world data producers). The framework counts with a wide option of settings to define the context. STREAMER is programmed in Java but learning algorithms are accepted in any programming language. Furthermore it can be used in any operative system. User is always guided by a log system that tracks the execution progress, errors, results, etc.

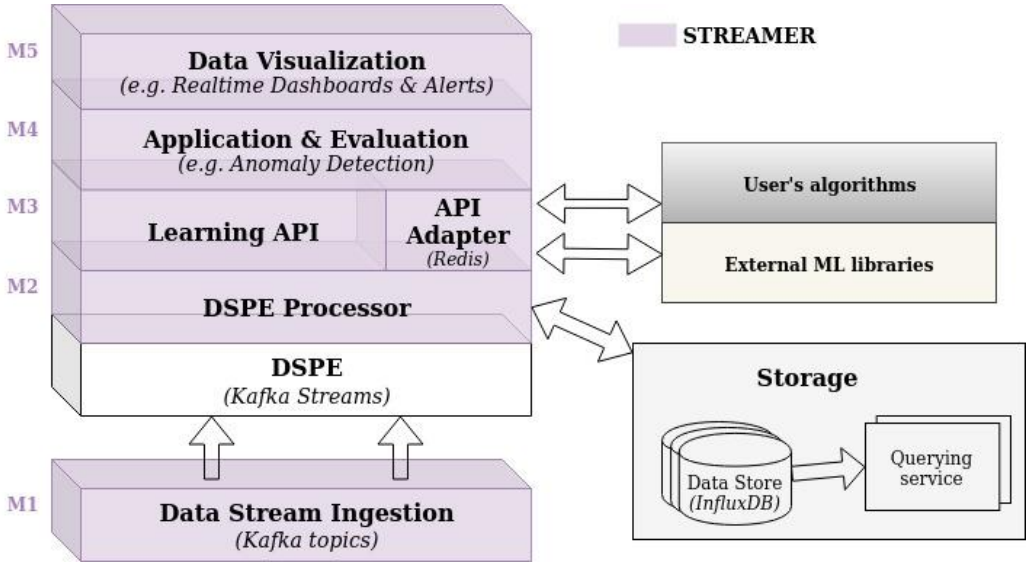


Figure 1 - STREAMER architecture

Figure 1 shows STREAMER architecture which contains the following modules:

1. M1- **Data Stream Ingestion**. It simulates a distributed data generation. Data are extracted from 1...N sources and sent to one or more *Kafka* channels (also called *topics*). These topics are then accessed by the *DSPE Processor* module to retrieve the data. It uses the abstract factory pattern to instantiate the appropriate production type:
    a. Offline: data are sent by blocks of *N* records and periodically every *t* seconds.
    b. Online: data are sent following its timestamp. Sending can be scaled, advanced or delayed in time.

    Each producer simulates one data source but it is possible to run several producers at the same time and send the data through one or more topics.

2. M2- **DSPE Processor**. Layer over the Data Stream Processing Engine *Kafka streams*. This DSPE adapter drives the streaming pipeline by communicating with the rest of the modules and invoking their functionalities. It is in charge of periodically retrieving the new data from the connected *topic(s)*; launching pre/post-processing, update-models and evaluation routines; processing the new collected data (run trained model); and sending information for *Data Visualization* in *JSON* format (cf. https://www.json.org/json-en.html). Incoming data between model updates are stored in *InfluxDB* (cf. https://www.influxdata.com). Such accumulated data are then used in the updating process and removed from the data base.

3. M3- **Learning Algorithms**. API with online and classic machine learning algorithms. Users have the flexibility of: (a) using the available algorithms, (b) developing their owns in Java by extending the API interface (methods *learn* and *run* when applicable), or c) incorporating some new ones through **Algs. Adapter**. Algorithms can be coded in any language since *Redis* (cf. https://redis.io) service is used for information exchange and model storage. M3 module can be invoked offline (batch training, to be used as a classical learning library) or online (learning update). Accumulated data between model updates (stored in *InfluxDB*) is then used for model updating.

4. M5- **Application & Evaluation**. Application domain specification, pre/post processing functions and evaluation metrics are implemented here. As other modules, M4 also allows users to easily integrate new functions or using the existing ones.

5. M4- **Visualization**. It provides a unique realtime and flexible dashboard for each running streaming pipeline. These dashboards can be either based on *elasticsearch* and *kibana* (cf. https://www.elastic.co/kibana) or on *Javascript* and *nodejs* (cf. https://nodejs.org/es/).

Modules are connected through some services provided by external tools. Note that STREAMER does not use the full functionality of the already existing tools but only integrates few specific services among the ones they provide. This way the framework remains light and with low complexity. Services act as interfaces that are strictly used for exchanging information between modules. This degree of decoupling enables to easily replace those components without affecting the rest of the framework. Figure 2 shows how modules communicate through the following services:

i. **Kafka/Kafka Streams** (cf. https://kafka.apache.org/): for building real-time data distributed pipelines (topics). Data are streamed through them.
ii. **Redis** (cf. https://redis.io/): an in-memory data store.
iii. **InfluxDB** (cf. https://www.influxdata.com): time series oriented database.
iv. **JSON** (cf. https://www.json.org/json-en.html): a lightweight data interchange format, easy for humans to read/write and easy for machines to parse/generate.

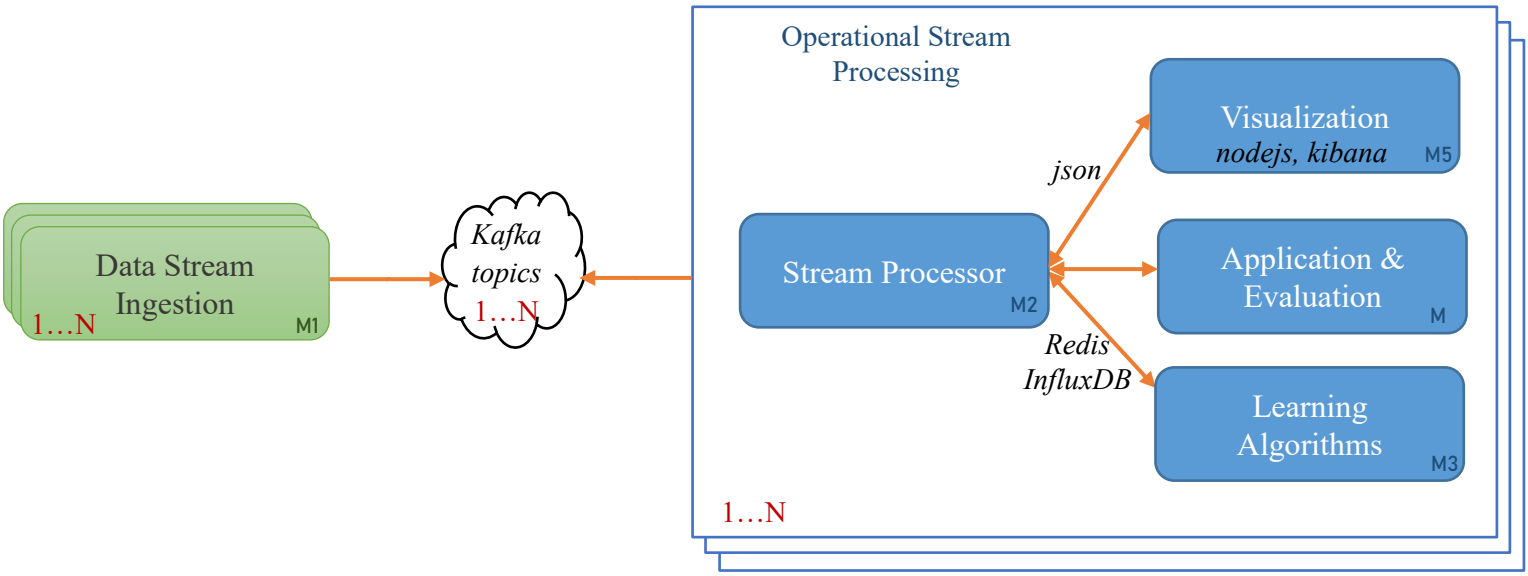


Figure 2 Modules of STREAMER

The aforementioned modules work in parallel and without strong dependencies. Their collaboration implements the whole streaming pipeline of *Figure 3 STREAMER Workflow*, however, users can decide to only use the functionalities they need. STREAMER also counts with a wide option of settings to define the context.

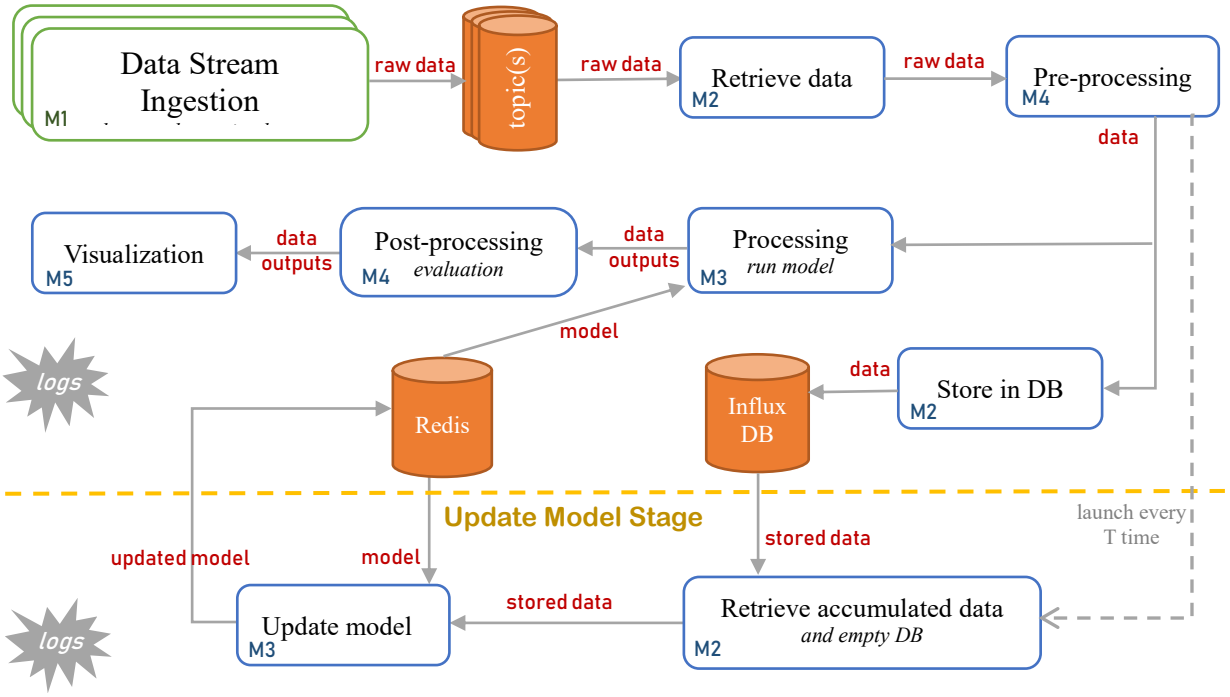


Figure 3 STREAMER Workflow

3. Data Stream Ingestion

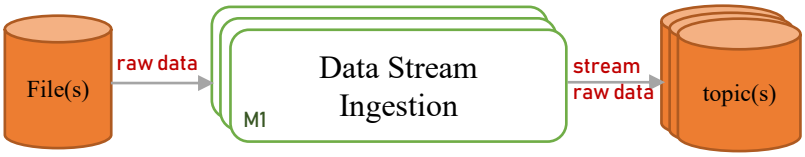
3.1. External Services

- Kafka (producer).
- Kafka Streams.

3.2. Description

Also called "Data Producer". It is in charge of simulating the data generation from 1...N sources. Data are extracted from the source and sent to one or more Kafka channels (also called *topics*). These topics are then accessed by the Stream Processor module to retrieve the data. Each producer simulates one data source but it is possible to run several producers at the same time and send the data through one or more topics. There are two kinds of production:

a. Offline: data are sent by blocks of *N* records and periodically every *t* seconds.
b. Online: data are sent following its timestamp. Sending can be scaled, advanced or delayed in time.



The data is extracted from the source and stored in one or more Kafka channels (also called *topics*). These topics are then accessed by the streaming module to retrieve the data.

This module allows running several producer processes at the same time (which means reading from different sources (files, etc.) and storing them in separate channels or the same one.

3.3. How to run it

There are two possible invocations:

3.3.1. **1 producer process – 1 data source**

a) Set producing properties in the file src/main/resources/streaming.props:

- Kafka server address and properties:
  bootstrap.servers=localhost:9092 # 10.0.238.20:6667
  acks=all
  retries=0
  batch.size=16384
  auto.commit.interval.ms=1000

linger.ms=0
key.serializer=org.apache.kafka.common.serialization.StringSerializer
value.serializer=org.apache.kafka.common.serialization.StringSerializer
block.on.buffer.full=true

- Maximum number of blocks to be sent :
  maxBlocks = 1000 #maximum number of blocks to send
- Time records per block
  recordsPerBlock = 6 #how many time records per block
- Time interval (in ms) between blocks
  producerTimeInterval = 10000
- Data source (from where it is read)
  datafile = ../data/ChloreScanSensors/data-test/capt1Test.csv
- Topic(s) name (from 1 to N) to write the source data
  mainTopic = topic1, … , topicN
- Production Type ("ONLINE" or "OFFLINE").
  producerType = OFFLINE
- Advance (positive) or delay (negative) time regarding the time stamp of the records (ms)
  epsilonne = -3000
- Time scale (ms). Note: it cannot be 0
  scale = 1800

In the case of many topics, the producer replicates the time records in all topics.

b) Run cea.streamer.ProducerMain

3.3.2. **Several producer processes – Several data sources.**

Each data source has a producer associated. This module also allows to have several independent processes running in parallel. To do so we just need to have different folders and configuration files on each. Name of folders must be provided as arguments to the main class.

a) For each data source/producer process, a new folder with file properties must be created. So for each src/main/resources/sourceName/streaming.props the same properties as in 3.3.1 must be set.
b) Run the producer sending as arguments the name of all sources folders: cea.streamer.ProducerMain folderSource1 … folderSourceN
c) The platform will create a new producer for each source with the properties specified in the correspondent folder.

***Note: Kafka properties will be taken from the streaming.props of the first argument folderSource1.

4. DSPE Processor

4.1. External Services

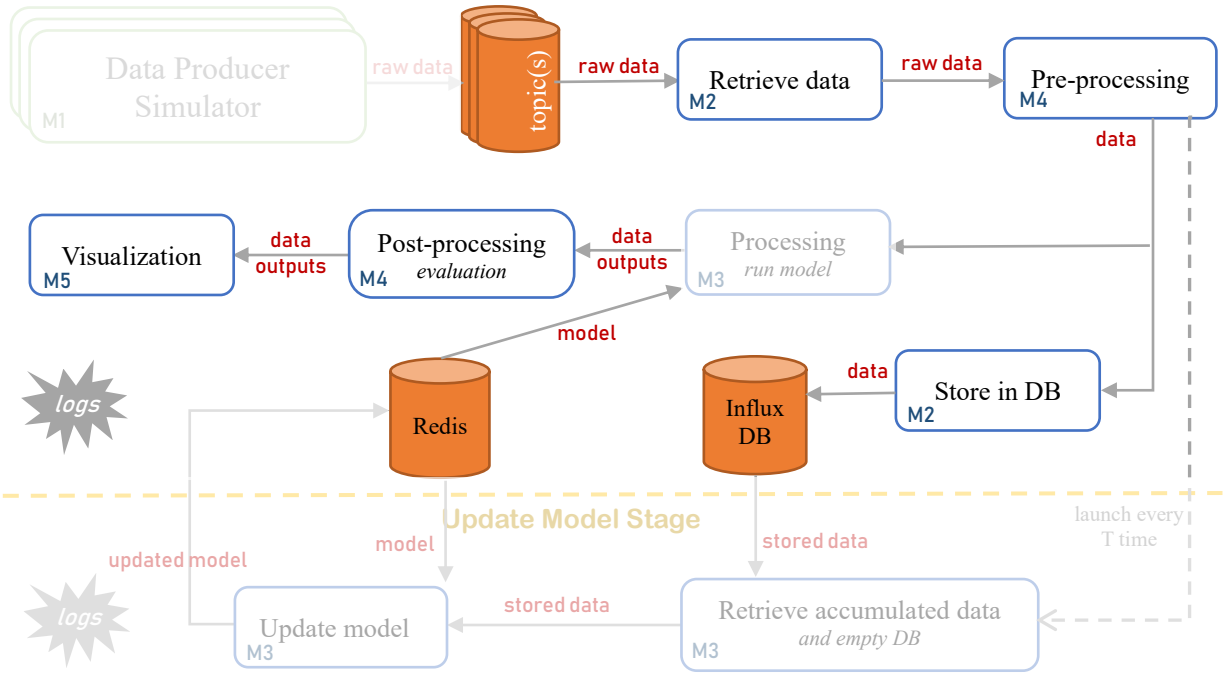
- Kafka (consumer).
- Kafka Streams (Low Level processor).
- Redis https://redis.io/
- InfluxDB.

4.2. Description

This is the main module of the framework. It is in charge of collecting the data from the input topic(s) and manage them in streaming. It drives the streaming pipeline by communicating with the rest of the modules through the services described above. It retrieves periodically the data from the connected topic(s), preprocesses them, launches the update learning model routine, processes the data using the trained model stored in *Redis*, generates the results, sends information for visualization, etc. Parameters, as interval times, are decided by the user.

It communicates with *ML Algorithms* module using *Redis* as the intermediate storage for models and information. Algorithms module is invoked online to:

- Optionally train a model by extracting all the data accumulated since the beginning of the execution. This trained model is stored in Redis.
- Use the trained model stored in Redis to test the new incoming stream of time records. Training online is also allowed but optional.



4.3. How to run it

There are two possible invocations:

4.3.1. **1 streaming process**

a) Set streaming properties in the file src/main/resources/streaming.props:

- Kafka server address and properties:

```
bootstrap.servers=localhost:9092
#bootstrap.servers=10.0.238.20:9092
#bootstrap.servers=10.0.238.20:6667
#bootstrap.servers=10.0.238.12:6667

####Launcher:
group.id=test
application.id=test
#enable.auto.commit=true
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer
# fast session timeout makes it more fun to play with failover
session.timeout.ms=10000
# These buffer sizes seem to be needed to avoid consumer switching to
# a mode where it processes one bufferful every 5 seconds with multiple
# timeouts along the way.  No idea why this happens.
fetch.min.bytes=50000
receive.buffer.bytes=262144
max.partition.fetch.bytes=2097152
```

- It indicates every how much time (in ms) the process collects the new data arrived in the main topic

```
readingTimeInterval = 10000
```

- Problem type. Be aware, this name must be the same as the time record class (ProblemNameRecord) in folder cea.streamer.core. There are now 2 available time records types in the framework:
    i. problem.type = WaterPollution → corresponds to WaterPollutionRecord
    ii. problem.type = LeakDetection → corresponds to LeakDetectionRecord

- If there is no preprocessing/postprocessing functionality specified, this line should not appear, otherwise the exact name of the preprocessor class must be indicated in this field

```
consumer.prepostprocessor = PreProcessorWaterPollution
```

- Output kafka channel (topic) where incoming streaming data is written

outputTopic = OutputPlatform

- Topic(s) name (from 1 to N) from where the data is collected

mainTopic = topic1, … , topicN

b) Set algorithm properties in the file src/main/resources/algs.props:

- Algorithm name to use for training and test purposes. The name of the algorithm must be same as the class that calls the algorithm in cea.streamer.algs. There are 4 algorithms implemented in the framework (see Section 3-ML Algorithms) for further details:

#algorithm = XGBoost
#algorithm = LeakDetector
#algorithm = Autoencoder
#algorithm = SVMWaterPollution
algorithm = SVMLeakDetector

- The training is called every *training.interval* time (real computer time). It is in in ms. For no online training set "-1"

#training.interval = -1
training.interval = 10000

- Maximum number of records to accept for training (if the limit is reached the rest of the records are not considered in the training).

training.maxdata = 10000000

- Data source only for offline training

#training.source = ../data/ChloreScanSensors/data-train/capt1Train.csv
training.source = ../data/data-leakdetection/data-train/sect1Train.csv

- Evaluation metrics to evaluate the machine learning algorithm used for online training. The user should provide the name of the class used for evaluation (existing or user-customized). In case of multiple classes, the user should provide the names of the classes separated by a comma.

# configurations for evaluation metrics
evaluation.metrics = AccuracyMultiClassClassificationMetric,SensitivityMultiClassClassificationMetric,PrecisionMultiClassClassificationMetric,F1scoreMultiClassClassificationMetric

- The exact number of classes used for multi-class classification. This configuration property is optional and it is related to the evaluation metrics (just in case of multi-class evaluation metrics were selected).

classes.number=23

c) Run cea.streamer.LauncherMain

4.3.2. **Many stream processes**

This module also allows to have several independent processes running in parallel. As with *Producer module*, each streaming process we want to run needs for separate properties setup. To do so we just need to have different folders and configuration files for each process. Name of folders must be provided as arguments to the main class.

a) For each process, a new folder with files properties must be created. So for each src/main/resources/processName/streaming.props and src/main/resources/processName/algs.props the same properties as in 4.1 must be set.
b) Run the launcher sending as arguments the name of all process folders: cea.streamer.LauncherMain folderProcess1 … folderProcessN
c) The framework will create a new streaming process for each folder name with the properties specified in the correspondent folder.

***Note: Kafka properties will be taken from the streaming.props of the first argument folderProcess1.

5. Learning API + Algs. Adapter

5.1. External Services

- Redis https://redis.io/
- InfluxDB.

5.2. Description

API with online and classic machine learning algorithms. Users can also easily add their own algorithms by just extending the API Interface methods *learn* and *run* (when applicable) to implement them or as a call to their own algorithm. Algorithms can be coded in any language since *Redis* service is used for information exchange and model storage. This module can be invoked offline (batch training, M3 used as a classical learning library) or online (learning update). It stores the trained models in *Redis*.

The framework provides several algorithms which can be used for multiple problem types:

- A One-class SVM model. This model is used for unsupervised learning such as binary clustering. In a binary clustering problem, the data used for the training is composed of only one class. The test algorithm tries then to identify whether the test data point is in the same

class as the training data or not. An example of problem using OCSVM is anomaly/outlier detection in time-series.

- An XGBoost model. XGBoost is a library (https://github.com/dmlc/xgboost) that provides scalable and very optimized gradient tree boosting models for supervised machine learning problems such as classification (binary and multi-class) and regression (see https://arxiv.org/pdf/1603.02754.pdf for more details).
  The training algorithm is an R script (using xgboost library) which consists of first a tuning parameters step. The tuning is a grid search with several proposed values of parameters. The algorithm constructs every possible model (by cross-validation) corresponding to every combination of parameters and retains the one who minimized an error function.

- An Autoencoder (neural network model). This is used for outlier/anomaly detection and dimensionality reduction. The NN tries to reconstruct the input with a compressed representation of it. Thus, the output is a rough reconstruction of the input. The outliers could then be identify by comparing the output and the input.
  The algorithm uses the H2O library (https://www.h2o.ai/). As well as for xgboost, the first step is a grid search with different values of parameters (in this case, the parameters are the number of layers, the number of neurons per layer and the activation function). The algorithm finds the best optimal combination of parameters by cross-validation.

- Neural Networks. *To complete*

- Clustering algorithm. *To complete*

- Anomaly detection. *To complete*

- Hoeffding Tree Online Classifier Algorithm. It is an online classification algorithm from scikit-multiflow library and coded in Python. It deals with incremental updates and adapts to concept drift.

5.2.1 Online (training) mode:

The Launcher retrieves data from the sources every specific period X of time (for instance 10 minutes). After each retrieving data (every 10 minutes), the Launcher calls a **test algorithm** so that a machine learning model can be applied on this new data (since the last call 10 minutes ago). As the framework is running, the data is accumulated in *InfluxDB*. Therefore a new model is built every period Y of time (generally, Y >> X) by calling a **training algorithm** with the data stored in *InfluxDB*. The idea is that the more the framework runs, the more effective the model (applying on the coming data) is.

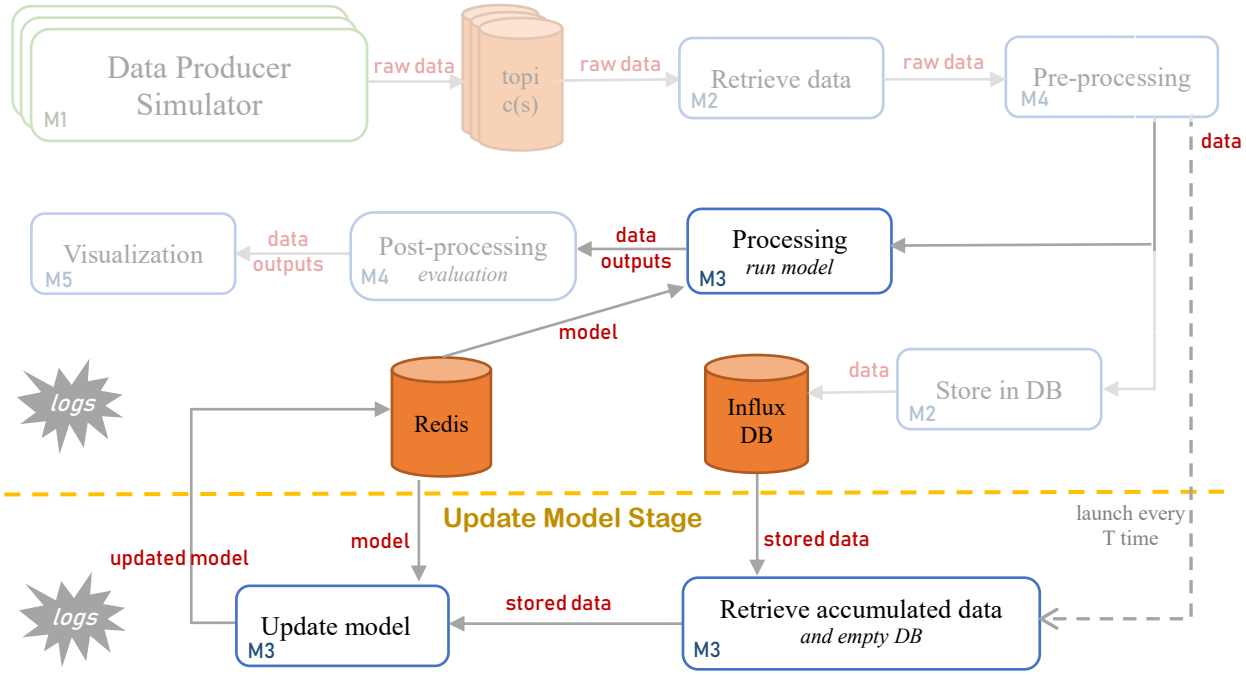
We support two types of running the algorithm in online mode:

1. Continuous learning: in which the algorithm does not need the entire data (or number of training records based on the configuration specified above for maximum data for training) in *InfluxDB* for training the algorithm. In such a case, the algorithm requires the previous model

and the new arriving data. This can be obtained by simply changing the attribute **up** in the constructor of the algorithm to **True**.

```
Public ConstructorNewAlgorithmClass(){
        modelUpdate = true;
}
```

2. Classical learning: in which the algorithm requires the previous data for training. For this type, we can have two ways of learning over the previous data:
    a. Learn over the entire data accumulated in *InfluxDB* and this can be achieved by setting the **maxmum data for training** to **-1**.
    training.maxdata = -1
    b. Learn over the recent N records of *InfluxDB*. For example below N = 5000.
    training.maxdata = 5000

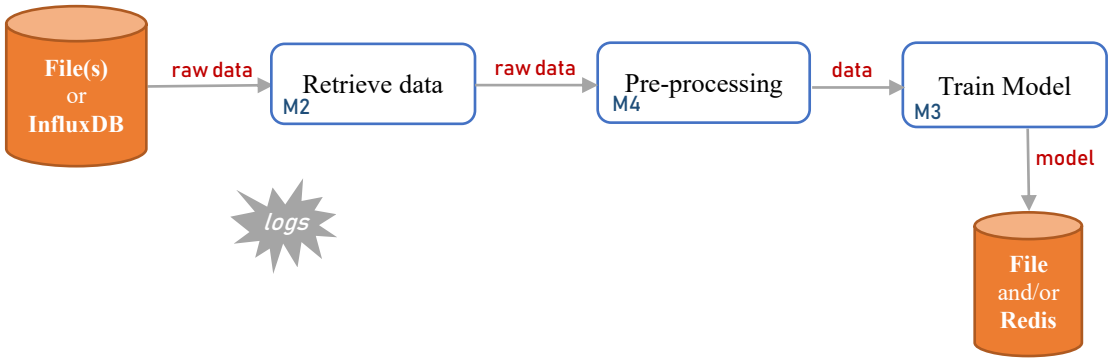


5.2.2 Offline mode (batch training and testing):

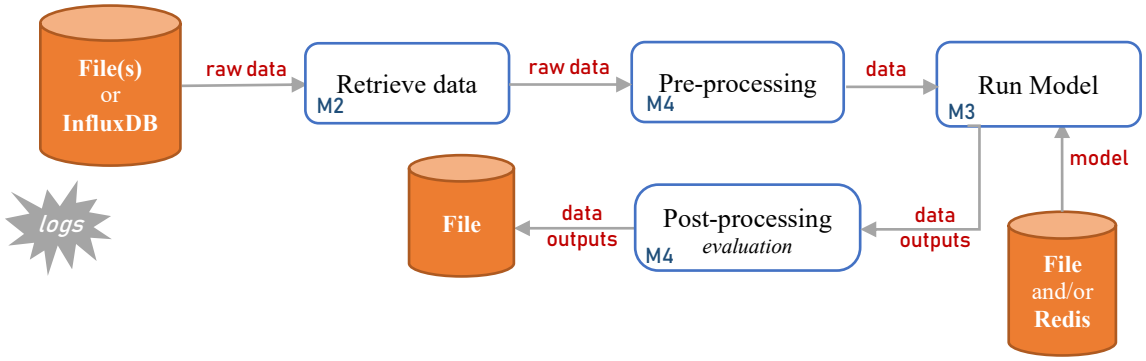
Besides the provided algorithms, the module allows the user to add his/her own algorithms. In fact, a new algorithm can be added by implementing the Java interface MLalgorithms (in src\main\java\cea\streamer\algs). The interface has two methods, learn() and run(), that are called periodically during the streaming according to the process above. The new algorithm can be written in any language. When the language is not Java, the connector service of *Redis* (util.connectors.RedisConnector) must be used to connect with the algorithm.

We use the main class streamer. AlgInvokerMain_offline to run the train and/or test of an algorithm. Their workflows are as follows:

- Train



- Test



5.2.3 Evaluation Metrics

This framework provide metrics for binary and multi-class classification applications. Evaluating binary classification algorithms is not the same as evaluating multi-class classification algorithms. Thus, the framework comes with the main metrics for evaluating both categories. These metrics include the following:

- Accuracy: is the number of correctly predicted data points out of all the data points.
- Sensitivity/Recall: is a measure of the proportion of actual positive cases that got predicted as positive (or true positive).
- Precision: quantifies the number of positive class predictions that actually belong to the positive class.
- F1-score/F-measure: it conveys the balance between the precision and the recall.

Concerning the classes used in the framework to be selected are as follows:

1. For binary classification (it is based upon calculating the true positive, false positive, true negative and false negative.
    a. AccuracyBinaryClassificationMetric
    b. SensitivityBinaryClassificationMetric
    c. PrecisionBinaryClassificationMetric
    d. F1scoreBinaryClassificationMetric
2. For multi-class classification (it is based upon calculating the confusion matrix for the different classes).
    a. AccuracyMultiClassClassificationMetric
    b. SensitivityMultiClassClassificationMetric
    c. PrecisionMultiClassClassificationMetric
    d. F1scoreMultiClassClassificationMetric

5.3. How to run it

5.3.1. **Changing the algorithm used**

See 4.3.1 b).

5.3.2. **Algorithm hyperparameters**

Some algorithms need to specify their concrete hyperparameters. The way to do is using a name_alg.props file where we will assign the different values to those specific hyperparameters. For instance, in the case of XGBoost algorithm, to choose between binary or multiclass classification for a XGBoost model, open the xgboost.pros and change the line "objective = binary:logistic" for binary or "objective = multi:softmax" for multiclass (you have to specify as well the number of class (num.class = ...)). The tuning step might be possibly long with a big amount of data. That's why the tuning for some parameters can be disabled. For that, change the boolean value in xgboost.properties of the parameters for which you want to disable the tuning.

5.3.3. **Offline Training and Testing Functionality**

The framework offers the option of training and/or testing the algorithms offline without the need of running the Launcher (streaming pipeline). We can train a model and store it in *Redis* and/or in disk, but we can also run any model from *Redis* or a file.
This offline functionality is very useful when you just want to use the algorithms of the learning API without implementing the streaming context. On the other hand, offline training can also be used, for instance, when we count with enough data to construct an initial ML model to then initially or permanently apply it to the data arriving (online training can be disabled as 4.3.1 b) shows) in the case of the streaming context implementation (see section 0).

To launch the offline mode, (algorithms test and/or train) run *AlgInvokerMain_offline.java* (in *streamer\src\test\java\cea\streamer*) as

```
AlgInvoker_Main.java arg1 arg2 arg3 arg4 [arg5]
```

Arguments are:

a) *arg1*. Data source type: From where data must be retrieved. Option: [*influx* or *file*]. It could be either "influx" to build the model using the data stored in InfluxDB or "file" to directly use the data from the file specify in algs.props (data.training).
b) *arg2*. Origin: [influx-data-base-name or origin]. *origin* indicates the folder where the properties files are located for this specific execution. If no arguments indicated, the system considers the properties files directly placed in *src/main/resources/setup* folder.
c) *arg3*. Perform training [*true* or *false*] (train).
d) *arg4*. Perform model running [*true* or *false*] (test).
e) [*arg5*]. [Optional] Path from where to store/retrieve the trained model.

The call to the main can be done manually as described above or through buttons in the graphical

interface (see 7.2).

6. Application & Evaluation

Furthermore, STREAMER counts with a set of functions that can be used in the development. They are placed in *src/main/java/cea/util* folder. Inside such folder we count with a set of:

- **Metrics** (folder */metrics/*) to be used in the evaluation of results. New ones can be easily added by the user.
- **Pre** and **post** data **processing** functions (folder */prepostprocessors/*). Some functions are already available. To add a new one it is necessary to create a new class that extends *PrePostProcessor.java* and implements the needed methods *preprocess()* and/or *postprocess()*. The name of the class to use must be specified in *streaming.props* as section 4.3.1 indicates.
- **Services connectors** (folder */connectors/*). Different functions are available to connect the services used in STREAMER (*InfluxDB*, *Redis*, *Kafka*, *R*, *Python*, etc.).
- Other utilities that can be used (read from file, store, prints, etc.).

Their description of all classes and their methods can be found in the *javadoc*.

7. Visualization

7.1. External Services

- Redis https://redis.io/
- NodeJS https://nodejs.org/
- D3.js https://d3js.org/
- Elasticsearch and Kibana https://www.elastic.co/kibana
- Socket.IO https://socket.io/
- json

7.2. Description

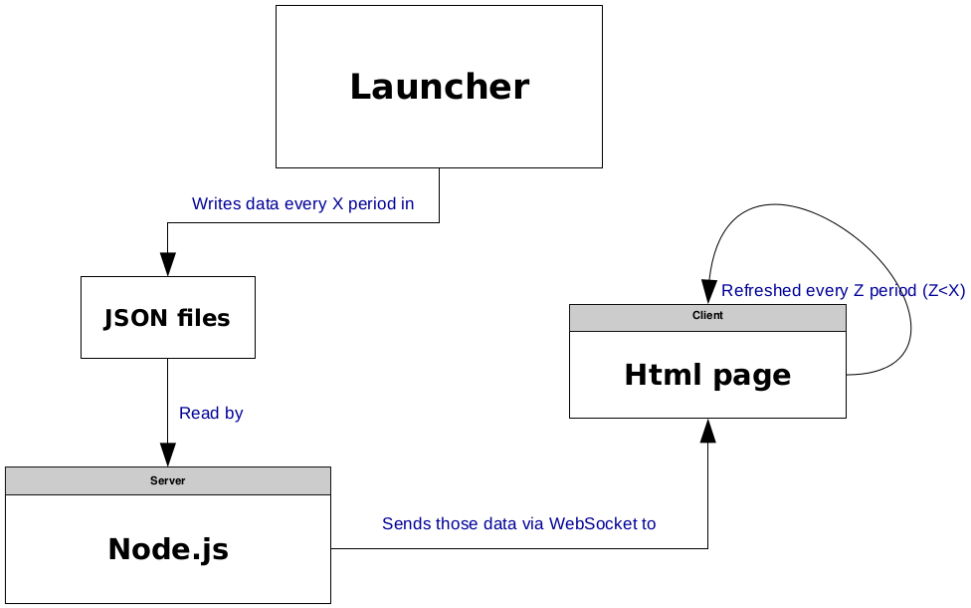
It provides a unique realtime and flexible dashboard for each running streaming pipeline. These dashboards can be either based on *elasticsearch* and *kibana* (cf. https://www.elastic.co/kibana) or on *Javascript* and *nodejs* (cf. https://nodejs.org/es/).

7.2.1. Nodejs Interface

The graphical interface is a Node.js server. It provides as many html pages as sources are (one web page corresponding to the analysis of one source in the framework). For displaying charts and plots, the html code uses the graphical JavaScript library **D3.js**. The server-side uses **Socket.IO** to perform real-time events via WebSocket.

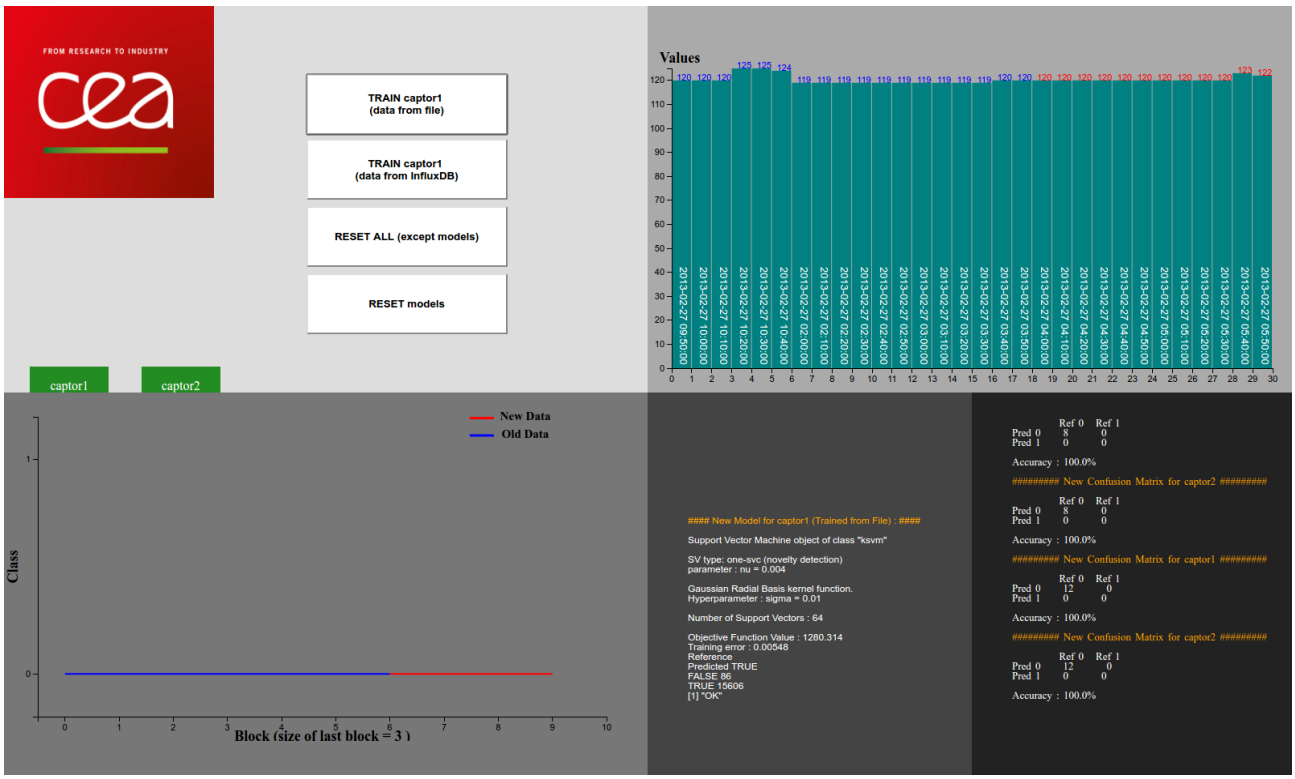
The figure below gives a complete description of the functioning of the module. As we can see, the streaming module (Launcher application) writes in JSON files the data (extracted from the topic) and

the results of the test algorithm every X period. Each time new JSON files are overwritten, the server reads them and extracts the time series and the test results. These two information are then sent to the html page (via WebSocket) which constructs data charts using D3.js. The page is refreshed automatically every period Z so that the plots can be displayed (Z < X). Therefore, every period Z, a new plot is displayed. It thus simulates a real-time data streaming process.



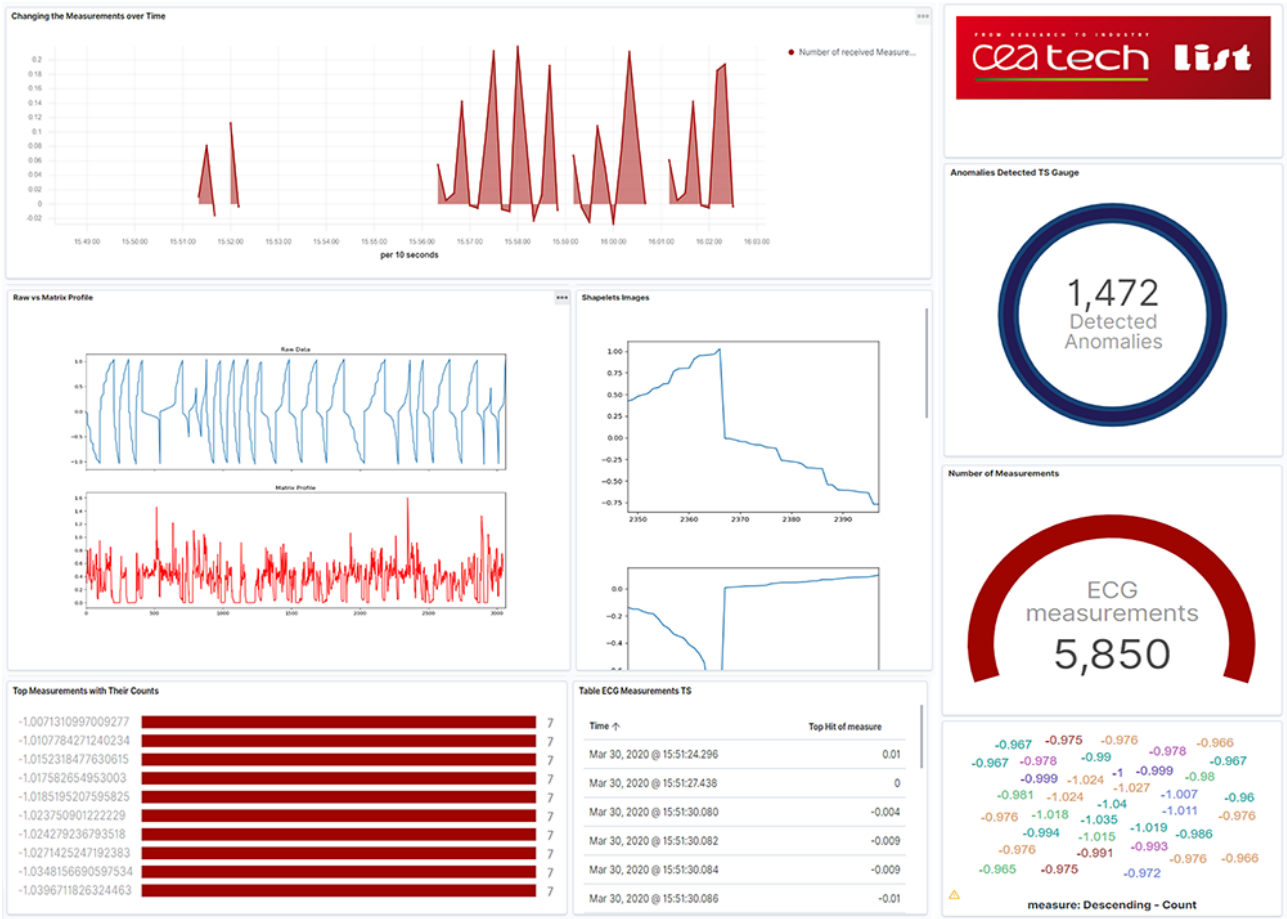
As we can see in the following figure, the web page for a source is divided in 4 parts:

- Up Left: The two training buttons (from InfluxDB and from file), the URL links to the other sources and the CEA logo. The click on one of the training buttons builds a new model for the source corresponding to the html page. It is trained from either a data file (training.source in algs.props) or from the data stored in InfluxDB.
- Up Right: The chart that displays the data extracted from the topic periodically.
- Down Left: The chart that displays the results of the test algorithm. For example, it could be classes in case of a classification project.
- Down Right: The part that displays the logs. Generally, the logs contain the summary of the model that is currently applying on the data.



7.2.2. Kibana Interface

So far only adapted to ECG anomaly detection application.



7.3. How to run it

7.3.1. Nodejs Interface

7.3.1.1. **1 streaming process**

The streaming application running or not, you can run the main application of the interface module (*app.js*) with the following command (make sure to be in *streamer\src\main\resources\graphics*):

```
node app.js
```

An HTML page indexdefault.html is then created. You can go to http://localhost:8080/default to display it.

7.3.1.2. **Many stream processes**

For multiple sources, you have to specify the names of the folder in arguments when calling *app.js*. For example, for two sources with folder names *origin1* and *origin2*, run the command:

```
node app.js origin1 origin2
```

Then open to http://localhost:8080/origin1 and http://localhost:8080/origin2. You can display as many pages as *originX* you specified as arguments since the interface works in parallel. Note: the arguments [*origin1 … originN*] must be the same ones introduced when running *LauncherMain* class.

7.3.1.3. **Graphical configuration**

The graphical interface gives the possibility to configure a certain number of displaying parameters according to your project. These parameters are: the maximum amount of displayed data (Up Right), the maximum amount of displayed results (Down Left), 2 booleans that indicate if the timestamps/values of the data are displayed (Up Right) and the X/Y axes name (Down Left). The default value of the parameters are respectively: 30, 10, true/true, "Class"/"Block". If you want to change any of these parameters, open the index.html file (in src/main/resources/graphics/index/) and replace the lines corresponding to the parameters in the configuration section (the lines starting by "// configuration").

7.3.2. Kibana Interface

Open a web browser and type http://localhost:5601. The dashboard will appear. (Settings can be modified in file dsplatform/data/streamops_data/kibana_settings). Kibana provides a way to build your own enriched interface depending on the application you are building. Visualizations are application-dependent. Thus, we facilitate the proper connection and the services in order to provide all the necessary data in time for visualization. Building your own customized dashboard is why we selected Kibana in the first place as an alternative visualization tool to act as an insightful graphical interface.

8. Learning API Connector

STREAMER counts with a connector that allows using the learning API as a library from any external codes. Located in *src/main/java/cea/util/connectors/AlgsExternalConnector.java*, this connector implements the calls to the methods *learnModel()* and *runModel()* to the API algorithms. Setups and algorithms hyperparameters do not no longer need to be specified in properties files but passed them as arguments to the functions. Available functions are:

```
	/**
	 * Constructor of the learning API connector
	 * @param id Identificator of the problem (must be unique)
	 * @param algorithmName Algorithm to use
	 * @param hyperParams List of the hyperparametres values for the algorithm, null if there is none
	 * @param prePostproc Name of class that contains the pre and post processing methods to apply to the data, null if no processing method is needed
	 */
	public AlgsExternalConnector(String id, String algorithmName, String[] hyperParams, String prePostproc)

	/**
	 * Calls the training of the algorithm
	 * @param file Input data to read from file
	 * @param modelStoredPath Path where trained model will be stored
	 * @return Outputs (if applicable as it is the case of clustering)
	 */
	public Vector<String> learnModel(String file, String modelStoredPath)

	/**
	 * Calls the training of the algorithm
	 * @param files List of files from where inputs for the algorithm are read
	 * @param modelStoredPath Path where trained model will be stored
	 * @return Outputs (if applicable as it is the case of clustering)
	 */
	public Vector<String> learnModel(Vector<String> files, String modelStoredPath)

	/**
	 * Calls the algorithm to run the model
	 * @param recordsDB Input records
	 * @param modelPath Path of file where trained model is stored.
	 *                  NULL if model is stored in redis not in disk
	 * @return Vector<String> or results
	 */
	private Vector<String> runModel(Vector<TimeRecord> recordsDB, String modelPath)

	/**
	 * Runs the model
	 * @param inputs for the model
	 * @param modelPath Path of file where trained model is stored.
	 *                  NULL if model is stored in redis not in disk
	 * @return Vector<String> or results, for no inputs null outputs
	 */
```

```
public Vector<String> runModel(String[][] inputs, String modelPath)

/**
 * Runs a model
 * @param fileInputs file in from where we read the inputs
 * @param modelPath Path of file where trained model is stored.
 *                                  NULL if model is stored in redis not in disk
 * @return Vector<String> or results
 */
public Vector<String> runModel(String fileInputs, String modelPath)

/**
 * List hyperparams need for the algorithm
 */
public void getAlgHyperparams()
```

From the external code, the user must instantiate the connector class and then call its methods freely as the example shows:

```
AlgsExternalConnector("id0","AutoEncoder",null,"PrePostProcessorNormalizer");
ec.setRecordName("TestRecord");
ec.learnModel("fileTrain.csv", modelPath);
Vector<String> results = ec.runModel("fileTest.csv" modelPath);
```

Check java documentation placed in *javadoc* folder for more details.

[i] https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple
[ii] samoa.incubator.apache.org/documentation/SAMOA-for-MOA-users.html