STREAMER User Guide v3.3

STREAMER team

October 2023

Contents

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. 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.

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.

STREAMER is originally composed of 5 main modules (**M1** to **M5**). Since the release of STREAMER 3.0, an additional module, called **distributed learning module (dM)**, enables STREAMER to operate in distributed environments.

- 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:
  - Block: data are sent by blocks of N records and periodically every t seconds.
  - Timestamp: data are sent following its timestamp. Sending can be scaled, advanced or delayed in time.

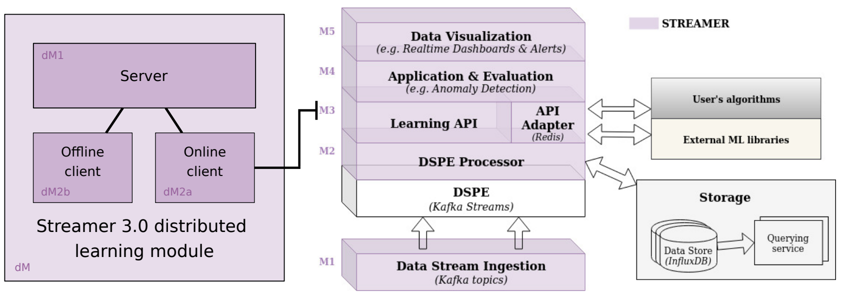


Figure 1: STREAMER architecture

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.

- 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.

- M3- **Learning Algorithms.** API with online and classic machine learning algorithms. Users have the flexibility of:

  - using the available algorithms
  - developing their owns in Java by extending the API interface (methods learn and run when applicable)
  - 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.

- M4- **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.

- M5- **Visualization & Monitoring.** 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/`).

- dM- **Distributed Learning Module**. It enables STREAMER to operate in distributed environments by providing a **server component (dM1)** which orchestrates a training process, and **client components** which are in charge of updating the model. The client components can be either online clients (**dM2a**) which are distributed layers built on top of the STREAMER instances, or offline clients (**dM2b**) which are lighter independent objects that do not use the rest of STREAMER modules. Submodule dM2a replaces module M3 when the distributed learning module (dM) is used.

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:

1. Kafka/Kafka Streams (cf. `https://kafka.apache.org/`): for building real-time data distributed pipelines (topics). Data are streamed through them.

2. Redis (cf. `https://redis.io/`): an in-memory data store.

3. InfluxDB (cf. `https://www.influxdata.com`): time series oriented database.

4. 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.

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.

3 Data Stream Ingestion -Producer-

The definition of the stream ingestion environment is setup up through its configuration file (sec. 9.1).

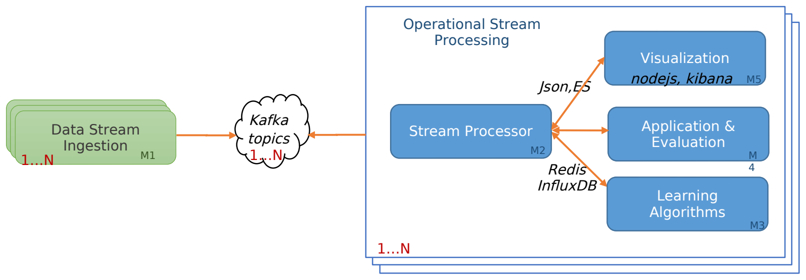


Figure 2: Modules of STREAMER

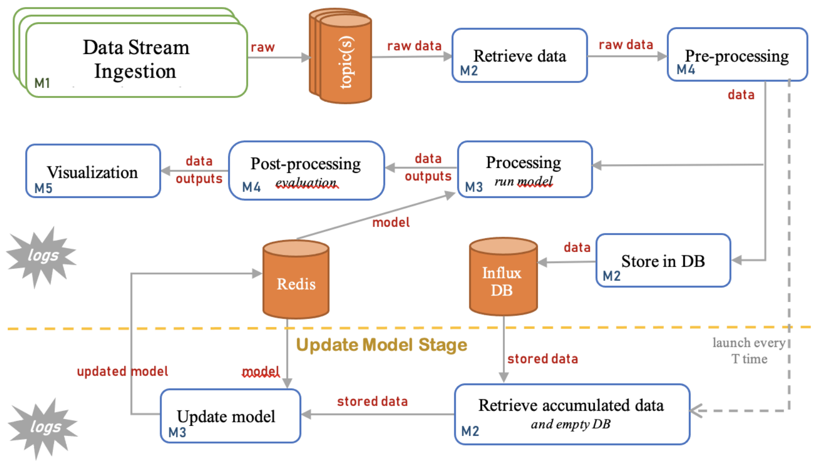


Figure 3: STREAMER Workflow

3.1 External Services

- Kafka (producer).
- Kafka Streams.

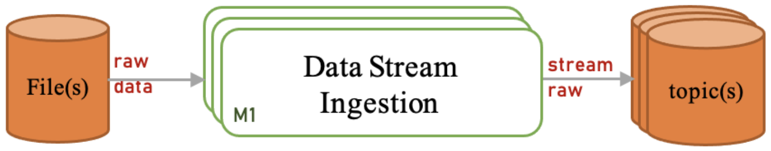
3.2 Description

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 serveral kinds of production:

1. Block: data are sent by blocks of N records and periodically every t seconds.

2. Block by group: same principle as Block producer, but adds a 'group' constraint for each line of the same block.

3. Timestamp: 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.

4 DSPE Processor -Launcher-

The definition of the stream ingestion environment is setup up through its configuration file (sec. 9.1).

4.1 External Services

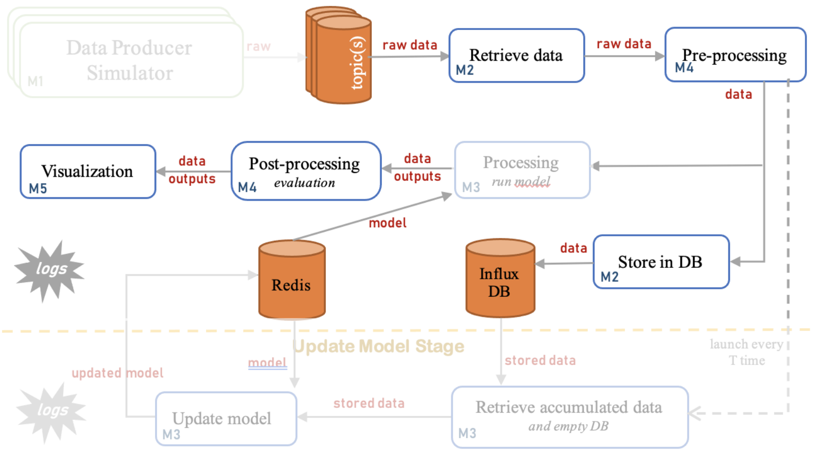
- Kafka (consumer).
- Kafka Streams (Low Level Processor)
- Redis https://redis.io/
- InfluxDB

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 and testing are both allowed but optional.



5 Learning API + Algs. Adapter

The definition of the stream ingestion environment is setup up through its configuration file (sec. 9.2).

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. Call to the neuralnet (implemented in R) algorithm contained in the library: (`https://www.rdocumentation.org/packages/neuralnet/versions/1.44.2/topics/neuralnet`).

- Clustering algorithm. It implements the optCluster (R) algorithm, which is an extension of cvalid. optCluster performs cluster analysis using statistical and biological validations for continuous and discrete data. When called with our data as a parameter, the algorithm provides us with a containing several pieces of information in response, including an ordered list of the methods each concatenated with the number of classes that goes with it (e.g. kmeans-3 means method kmeans with a number of classes equal to 3). Ref.: `https://www.rdocumentation.org/packages/optCluster/versions/1.3.0/topics/optCluster`

- 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.

Note that some algorithms need to specify their concrete hyperparameters. The way to do is using a namealg.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.

Furthermore, 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.

5.2.1 Online training mode

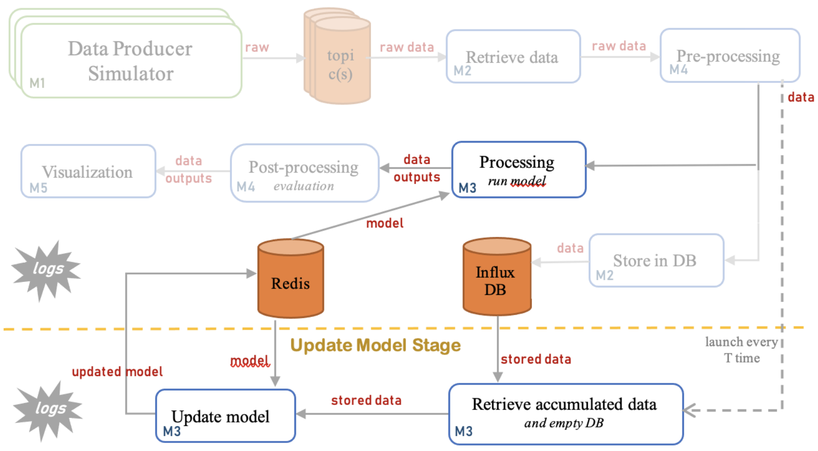
The Launcher retrieves data from the sources every specific period X of time (for instance 10 minutes). After each retrieving data (every 10 minutes), the Launcher calls a test algorithm so that a machine learning model can be applied on this new data (since the last call 10 minutes ago). As the framework 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:

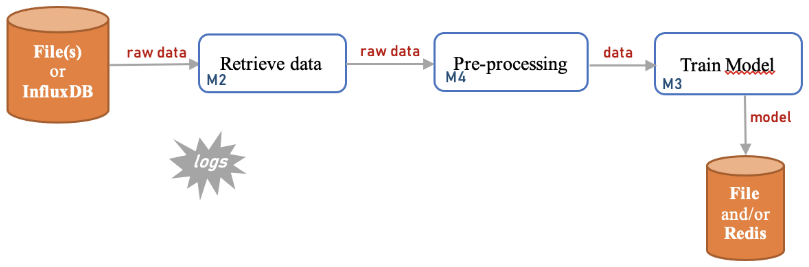
    (a) Learn over the entire data accumulated in InfluxDB and this can be achieved by setting the maxmum data for training to -1. training.maxdata = -1

    (b) Learn over the recent N records of InfluxDB. For example below N = 5000. training.maxdata = 5000



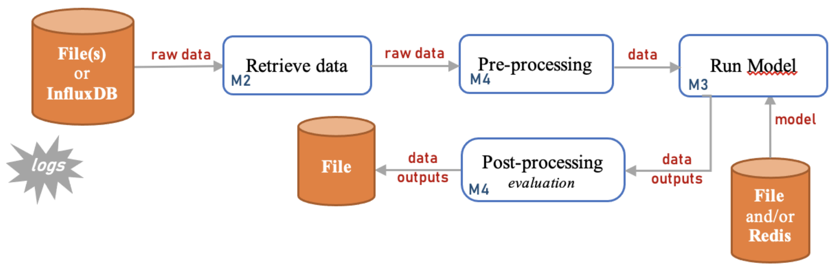
5.2.2 Offline mode (batch training and testing)

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:



5.3 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 :
AlgInvokr_Main.java arg1 arg2 arg3 arg4 [arg5]
Arguments are:

1. 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).

2. 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.

3. arg3. Perform training [true or false] (train).

4. arg4. Perform model running [true or false] (test).

5. 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.

6.1 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

In addition to classification applications, the framework consists of different metrics for evaluating the regression applications. These metrics include the following:

- Mean Absolute Error (MAE): is the sum of absolute differences between our target and predicted variables. This measures the absolute average distance between the real data and the predicted data.
- Mean Square Error (MSE): This measures the squared average distance between the real data and the predicted data.
- Root Mean Square Error (RMSE): This measure tells you how concentrated the data is around the line of best fit. It is the square root of MSE. It measures the standard deviation of residuals.
- R^2: also known as the coefficient of determination. It represents the proportion of the variance in the dependent variable which is explained by the linear regression model
- Prediction Interval Normalized Root Mean Square Width (PINRW)
- Prediction Interval Normalized Average Width (PINAW)
- Average Coverage Error (ACE)

The classes of those metrics are:

- MeanAbsoluteErrorMetric
- MeanSquareErrorMetric
- RootMeanSquareErrorMetric
- RSquaredMetric
- PredictionIntervalNormRootWidth
- PredictionIntervalNormAvWidth
- AverageCoverageError

User can specify the metrics to use through *algs.props* configuration file (sec. 9.2).

7 Visualization & Monitoring

7.1 External Services

- Redis `https://redis.io/`
- Elasticsearch and Kibana `https://www.elastic.co/kibana`

7.2 Monitoring & Model Update

The online monitoring module performs the following tasks:

1. Monitoring the evolution of the results provided by a model when performing inference and online learning through the use of certain metrics.
2. Automatically detecting when performance drops below expectations and raise an alert.
3. Updating the model
4. Showing graphically and in logs the whole monitoring process.

Users can define the monitoring context through the setup properties files *monitoring.props* (sec. 9.3 and consult at any time the evaluation metric values or monitoring messages in the log files which are updated in real time.

7.2.1 Detection and Alert raising

STREAMER enables monitoring [1...N] metrics with different priorities. It can detect when expected results are dropping to, therefore, raise an alarm and call the model update. For this, a **detection algorithm** was implemented based on two **deviation detection criteria**:

1. **CUSUM** detector: it sequentially calculates the positive and negative cumulative sum S across data. When the value of S exceeds a certain positive and negative threshold value, a change is detected.
2. **Threshold** detector: it follows the evolution of the metric values and notifies a deviation when one of them exceeds a specific *threshold*.

The detection algorithm is called each time a new bunch of data is evaluated. It observes each metric separately and raises an alert when a deviation is notified by the detection criteria for more than a certain number of iterations in a row (*robustness*). In the same way, the alert is removed when the detection criteria did not detect any deviation during consecutive *robustness* iterations. Thus, using the parameter *robustness* gives stability to the detection algorithm since it avoids constant changes.

When an **alert** is raised, the detection method launches a detection event with format $[id, metric, Start, Stop, Touch]$; where id is the identifier of the use case, $metric$ is the monitored metric, $Start$ is the first iteration the alert was raised, $Stop$ is the iteration in which the alert was stopped, and $Touch$ is the last iteration a deviation was identified.

Based on the previous specifications, we can observe the following constraints:

- Initial values: $Start = Stop = Touch$
- if $Start = Stop \neq Touch$ then alert is launched and ongoing

- if $(Touch + robustness) = Stop$ then alert is deactivated

For each iteration (a new bunch of data processed), if the alert is still activated, STREAMER calls the routine in charge of updating the model with the new data. Once the new model is generated, STREAMER removes the alert but does not stop the detection event. This way, if a deviation is still detected during the next iteration, the alert will be raised again.

7.2.2 Interface and Logs

Users can also follow the evolution of the monitoring (errors, warnings, evaluation metrics across time, monitoring detections, alerts, retraining phases, etc.) at any time by just checking the log files STREAMER generates. Placed in the log folder, the users can find:

- Logs reporting the execution of the algorithms as *displayLogTrain*, *displayLogTest*, *errorLog*, *infoLog* (also shows alerts and detections), *temp_[id]* (temporary log while algorithm is running), etc.
- *metricsLog* where all the evaluation metrics are stored in time.

7.3 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/`).

7.3.1 Kibana Interface

In order to facilitate the understanding of the monitoring system, STREAMER provides a dynamic graphical interface (figure 4) which shows in real time the metrics evaluations, the alerts, the historical charts, etc. But STREAMER also provides all evaluation metrics across the time, monitoring detection, alerts, retraining phases, inference User can also follow the evolution of STREAMER at any time by just checking the log files STREAMER generates. Section 7.2.2 provides a wide description of these functionalities.

To access, open a web browser and type `http://localhost:5601`. The dashboard will appear. (Settings can be modified in file *kibana.props* (sec. 9.5). 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.

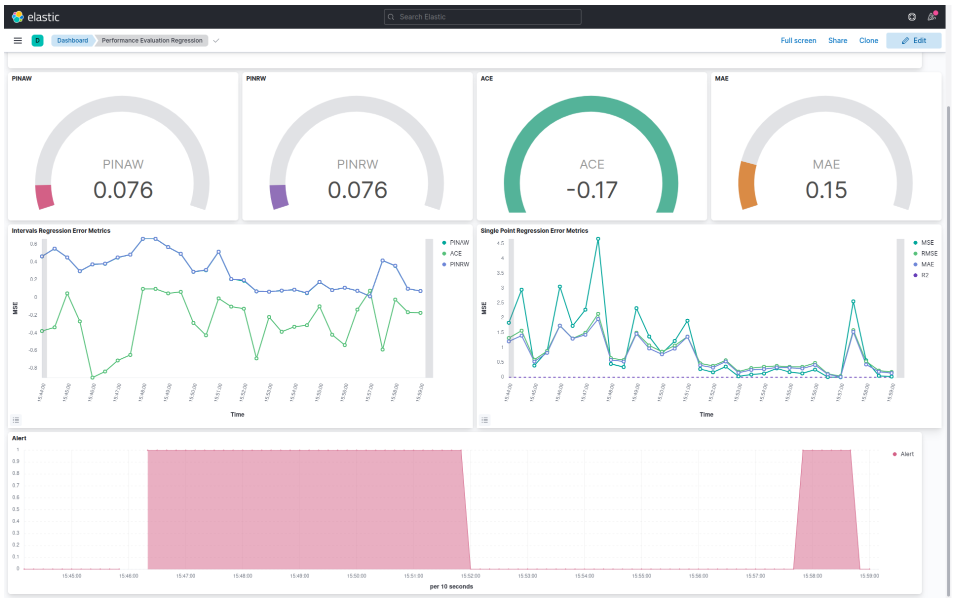


Figure 4: Example of the Monitoring interface

8 Distributed Learning Module

The setup of this module is done through federated.props (see sec. 9.4).

The distributed learning module enables STREAMER to operate in distributed environments. Instead of performing a traditional training process, a STREAMER instance can take part in a distributed learning phase, in which systems collaborate in a network with the aim of reaching a global model.

Federated Learning (FL) was selected as a distributed learning method. In this context, a **server** orchestrates a network in which **clients** can participate in a collaborative process using their own data source. First, each client, represented by the STREAMER instances, trains its model locally from the server model parameters. Then, clients send their own updated model parameters to the server which computes a global model by aggregating those client parameters. Finally, this process is repeated until the new global model is satisfactory. Since existing FL tools are not suitable for streaming data, we decided to implement our own module. Its development in JAVA was inspired by the Python FL framework "Flower" (`https://flower.readthedocs.io/en/latest/`). Moreover, communication between server and clients is managed by the framework gRPC (`https://grpc.io/`) which uses the protocol HTTP/2 for the transport.

The activation of this module implies that the server takes the lead in the training phase. This way, a new event-based training process will replace module M3 of STREAMER instances. Note that STREAMER can be used for training, inference, or both. If training is disabled, the STREAMER instance is a **passive participant**. It only retrieves the updated model and uses it for inference. When training is enabled, the STREAMER instance becomes an **active participant**. Whether learning is performed **online** (in data streams context) or **offline** (with batch data), the client participates in the distributed training process by following the instructions of the server. The only difference between offline and online training is that, in the first case, the process will be directly invoked without using the rest of STREAMER modules.

8.1 Architecture

The module is composed of two new components: the **server** which purpose is to orchestrate the global training process, and the **clients** which are responsible for updating the model. These last can be either online clients that are distributed layers built on top of the STREAMER instances, or offline clients that are lighter independent objects that do not use the rest of STREAMER modules.

The architecture overview and the interaction of these components are illustrated in Fig. 5.

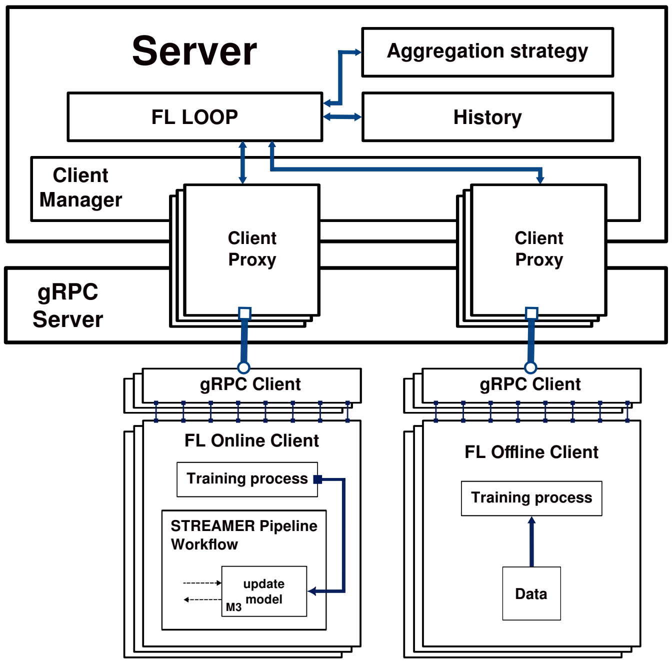


Figure 5: Distributed module architecture.

8.1.1 Server component

The server component launches two processes in parallel. The **server process**, which will orchestrate the FL loop and activate the difference phases (initialization, training, evaluation), and the **gRPC server process** which is a gRPC interface that manages the client message processing and the server message sending.

Several objects are involved during the server process:

- the **FL loop**, which determines the action performed in each round (model parameters initialization, fitting step, evaluation step, server evaluation step).
- the **ClientManager**, which controls the management of the client proxies. It creates, registers, and unregisters client proxies, samples clients, and manages the message waiting.
- the **aggregation strategy** class, which defines the FL strategy to follow. So far just the FedAvg algorithm has been implemented, but other strategies can be easily implemented.
- the **history** of the FL experience results, which are then saved in log files.

For further details, client proxies are defined at the server level as abstract clients with which the server can interact. It can send them requests, receive results from them, or disconnect them from the network. Each client proxy represents one single client that could be solicited by the server during the different phases.

8.1.2 Client component

On the other side, the client component (the online client one) is built on top of the STREAMER instance. A new process is opened in parallel to this instance in which a gRPC connection is made with the server. The training and evaluation steps will be now controlled by the server. This component contains a gRPC interface for the server that manages the server message processing as well as the client message sending.

The offline client only keeps all the gRPC processes without using the rest of STREAMER modules. However, in this case, the parameters training.source, in the 'alg.props' file, and, problem.type and containsHeader, in the 'streaming.props' file, are still needed to set up the record database.

8.2 Additional comments

Custom aggregation strategy: For the moment, just the FedAvg algorithm has been implemented as an aggregation strategy, but other strategies can be easily implemented. For this, you have to extend the AggregationStrategy class (in *src/main/java/cea/federated/server/aggregation*) and implement the 'aggregate_parameters()' and 'aggregate_losses()' methods.

FL steps & Python algorithm file: During the federated learning process, many actions (initialization, training, evaluation, inference) will be performed by the clients or the server. With the first version of the distributed learning module, only Python algorithms can be used to perform these actions. Moreover, to simplify the Python algorithm file, a new action process has been introduced to enable STREAMER triggering the requested action when it calls the associated Python algorithm file. All the Python algorithm files need to manage the process of the possible actions through a new argument 'state'. The Python file should implement and run the corresponding methods according to the following state values it receives:

- *init*: Initializes the model and pushes it to Redis.
- *train*: Fits the model and pushes the updated one to Redis.
- *evaluate*: Evaluates the model and pushes the metrics (the loss) to Redis.
- *inference*: Makes the inference and pushes the prediction to Redis.

9 Configuration Setup - properties files

In this section, configuration of STREAMER is setup through its properties files.

9.1 Producer & Launcher - *streaming.props*

This properties file serves to setup the data stream ingestion -producer- (sec. 3) and the DSPE processor reader -launcher- (sec. 4).

9.1.1 Data Stream Ingestion -producer-

There are two possible invocations:

1 producer process – 1 data source 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
  - In the case of many topics, the producer replicates the time records in all topics.
- Production Type ("TIMESTAMP" or "BLOCK" or "BLOCKBYGROUP")
  - producerType = BLOCK
- 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
- Specify if the added data file contains header
  - containsHeader = true

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. The names of folders must be provided as arguments to the main class.

1. 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.

2. Run the producer sending as arguments the name of all sources folders: cea.streamer.ProducerMain folderSource1 ... folderSourceN

3. The platform will create a new producer for each source with the properties specified in the correspondent folder.

9.1.2 DSPE Processor -launcher-

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

1. 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.

2. Run the launcher sending as arguments the name of all process folders: cea.streamer.LauncherMain folderProcess1 ... folderProcessN

3. The framework will create a new streaming process for each folder name with the properties specified in the correspondent folder.

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 consumers switching to a mode where it processes one buffer 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 record types in the framework:
  - problem.type = WaterPollution corresponds to WaterPollutionRecord
  - 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
- Launch the visualization, import the dashboard and visualize the performance of the model
  - Visualization = true
- Allow the user to choose between the following options:
  - online.train = true (The model is trained online), we can stop the training by setting the value of this property to false.
  - online.inference = true (activate the model's inference over the data), we can stop the inference by setting the value of this property to false.
  - The default value for both properties is true. That means the model will do both, do the training and perform the inference.

9.2 Learning API Connector - algs.props

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 the 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 = 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.
  - 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

9.3 Monitoring - monitoring.props

Users can also define their own monitoring context through the setup properties *monitoring.props* file and consult at any time the evaluation metric values or monitoring messages in the log files (see sec. 7.2.2) which are updated in real time. Make sure the configuration file is correctly defined and no compulsory parameter is missing. An example of setup is:

```
metrics = AverageCoverageError,PredictionIntervalNormAvWidth
detector = ThresholdDetector
#Parameters for Threshold
## Values: true or false. Alert is raised if value is greater [true] or
lower [false] than threshold
threshold.threshold.alert-if-greater = false
## Threshold is considered in its absolute value [true], otherwise [false]
threshold.threshold.abs = false

##Several thresholds accepted separated by comma, one per monitored metric
If only 1 value, it will apply to all metrics
threshold.threshold =  1300,0.18

##Several thresholds accepted separated by comma, one per monitored metric
If only 1 value, it will apply to all metrics
threshold.robustness = 2,5
```

It means that, for each metric, alerts will be raised after more than 2 occurrences (*threshold.robustness = 2*) of *AverageCoverageError* metric lower (*threshold.threshold.alert-if-greater = false*) than 1300 (*threshold.threshold = 1300, threshold.threshold.abs = false*). In the same way, the alert will stop after more than 2 batches with values greater than 1300. See sec. 7.2.1 for more information about detector algorithm.

9.4 Distributed Learning - federated.props

To configure the FL scenario, as to define the streaming context, its parameters can also be set up in the properties files (see sec. 9.4). For instance, users can configure the server address to launch an experiment on several machines, the number of rounds, the minimum number of clients to start a training step, and the maximum quantity of clients to sample during a training step.

Parameters are specific to clients and others are proper to the server, always filled in a 'federated.props' file, located in the same place as the other properties files.

9.4.1 Server

At the server level, it is necessary to fill in the parameters:

- **server.port**, the port of the server;

- **n.rounds**, the number of rounds in the FL loops;
- **aggregation.strategy**, the aggregation strategy to follow during the fitting and evaluation steps.

Some other parameters are optional. There are:

- **n.clients**, the number of clients to sample during the fitting and evaluation steps (default=-1, all the connected clients will be sampled);
- **min.n.clients**, the minimum number of clients required to run a sampling (default=1);
- **do.client.fit.step**, a boolean to active/deactivate the client fitting step (default=true);
- **do.client.eval.step**, a boolean to active/deactivate the client evaluation step (default=true);
- **do.server.eval.step**, a boolean to active/deactivate the server evaluation step (default=false with 'eval.python.file.path'=null);
- **eval.python.file.path**, the path of the python file used for the server evaluation step (default=null with 'do.server.eval.step'=false).

Note that 'alg.props' and 'streaming.props' are not required for the server as it only uses the distributed module of the STREAMER framework.

9.4.2 Client

At the client level, each client needs to have its own 'federated.props' file. It is necessary to fill in:

- the **distributed.mode**, a boolean that determines if the distributed mode is activated or not, only mandatory for online clients;
- the **server.address**, the address of the server needed to establish the connection;
- the **python.file.path**, the path of the ml python algorithm;
- the **data.type**, the way of storing the data for the ml python algorithm, only mandatory for offline clients (default=redis, option=redis/pickle).

Moreover, an offline client can be defined as 'passive' (only retrieves the updated model and uses it for inference) by setting the optional parameter **is.passive** on the 'true' value. An online client is automatically defined as 'passive' if the option 'online.train' of the 'streaming.props' file is set to 'false'.

Note: In the case of offline clients, a few parameters are still needed in the 'alg.props' and 'streaming.props' files. Only the parameters **training.source**, in the 'alg.props' file, and, **problem.type** and **containsHeader**, in the 'streaming.props' file, need to be filled.

9.5 Other setup

Users may also configure the different services using the different configuration files (placed in *setup* folder):

1. elasticsearch.props
2. kibana.props
3. redis.props
4. influxDB.props
5. codeConnectors.props
    - Specify the Python execution path (the command you use from the terminal to call Python)
    - Specify the R execution path (the command you use from the terminal to call Rscript)