Combining Machine Learning and Semantics for Anomaly Detection

Badre Belabbess[1,2](✉), Musab Bairat[1], Jeremy Lhez[2], and Olivier Curé[2]

[1] ATOS, 95870 Bezons, France
{badre.belabbess,musab.bairat}@atos.net
[2] LIGM (UMR 8049), CNRS, ENPC, ESIEE, UPEM, 77454 Marne-la-Vallée, France
{jeremy.lhez,olivier.cure}@u-pem.fr

Abstract. The emergence of the Internet of Things and stream processing forces large scale organizations to consider anomaly detection as a key component of their business. Using machine learning to solve such complex use cases is generally a cumbersome, costly, time-consuming and error-prone process. It involves many tasks from data cleansing, to dimension reduction, algorithm selection and fine tuning. It also requires the involvement of various experts such as statisticians, programmers and testers. With RAMSSES, we remove the burden of this pipeline and demonstrate that these tasks can be automated. Our system leverages on a Lambda architecture based on Apache Spark to analyze historical data, perform cleansing and deal with the curse of dimensionality. Then, it identifies the most interesting attributes and uses a continuous semantic query generator executed over streams. The sampled data are processed by self-selected machine learning methods to detect anomalies, an iterative process using end user annotations improves significantly the accuracy of the system. After a description of RAMSSES's main components, the performance and relevancy of the system are demonstrated via a thorough evaluation over real-world and synthetic datasets.

Keywords: Real time · Streaming · Machine learning
Knowledge extraction · Anomaly detection · Smart water management

1 Introduction

In a global context of smart cities and highly efficient business systems for large companies, identifying potentially harmful singularities that could lead to environmental or economic risks opened a new world of possibilities. Therefore, the growing research domain of anomaly detection [1] reached multiple fields from medicine (*e.g.*, to identify malignant tumors in MRI images) to finance (*e.g.*, to discover cases of credit card transaction frauds) or information technology (*e.g.*, to detect hacking situations in computer networks). Machine learning proposes a powerful set of approaches that can help solving such use cases in an efficient way. However, it represents a heavy process with strict rules that assumes an extensive time-consuming list of tasks such as data analysis, data cleansing, dimension reduction, sampling, proper algorithms selection, hyper-parameters fine-tuning and so on. It also involves several experts that will work collectively to find the right approaches. Therefore, any attempt to solve a use case with machine learning methods rapidly becomes an important team effort that requires great resources and top-notch skills. RAMSSES was specifically designed to remove the burden of machine learning heavy process and ease the deployment of an efficient solution for use cases related to detecting singularities related to Internet of Things (IoT).

Within the scope of the Waves project[1], RAMSSES aims at identifying irregularities in large potable water networks overseen by a worldwide company. The actual volume of lost water in the world has peaked to 32 billions m^3/year corresponding to a loss of US$ 14 billion/year with 90% of them being invisible due to the underground nature of the network. Our project partner's French water network counts around 100.000 Km of pipelines outfitted with more than 3.000 sensors and delivers potable water to roughly 12 million customers. The premise of this paper is to show that one can draw new knowledge and find hidden patterns by mixing quantitative and semantic approaches leading to a powerful system for supporting strategic business decisions. In summary, the main contributions of the RAMSSES platform are: **(i)** An automatic method to realize data pre-processing such as dimension reduction handled by a continuous query generator that leverages on semantic web technologies; **(ii)** An automatic approach to select attributes in a multidimensional large dump by analyzing the data profile; **(iii)** A complex set of rules to automatically select the proper machine learning algorithm by carefully analyzing the profile of the data ingested such as variables dependency or distribution type; and **(iv)** An extensive evaluation on both real and synthetic datasets to assess the overall performance of the platform.

2 Architecture

RAMSSESS was designed to process both massive dynamic and static data using a fault-tolerant distributed architecture that can easily scale. Figure 1 gives a general overview of the system's architecture. The main goal is to face real-time heavy throughputs of data as well as to compute intensive models for machine learning purposes. To meet the needs of a robust, fault-tolerant system, able to serve a wide range of workloads and use cases, and in which a low latency of reads and writes is required, we adopted a Lambda architecture [2]. This type of big data architecture solves the problem of heavily calculating functions on real-time data by decomposing the problem into three layers: *a batch layer* that manages an immutable append-only master dataset and pre-computes query functions called batch views; *a serving layer* which indexes the batch views for ad hoc low latency querying; and *a speed layer* that uses fast and incremental algorithms over recent data only.

[1] https://www.waves-rsp.org/.

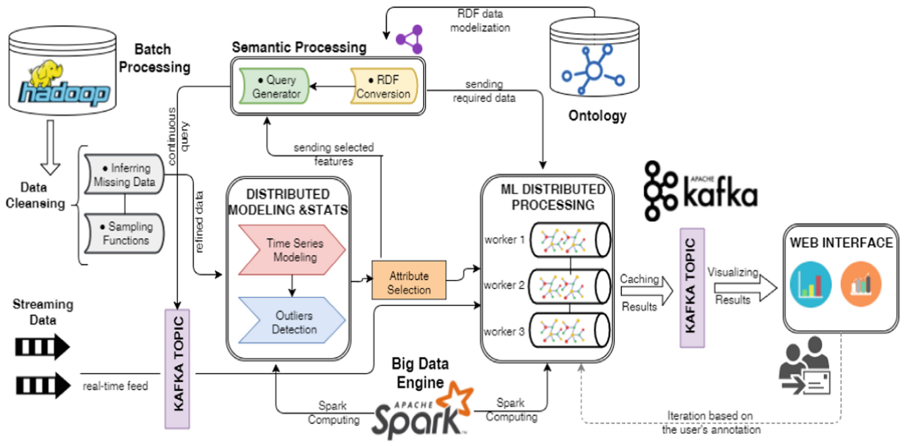

Fig. 1. RAMSSES architecture

A general end to end scenario for RAMSSES starts with storing historical time-series data of a use case for pre-analysis purpose. Caching heavy loads of data needs a robust distributed file system for storing and retrieving files in record time. Due to its massive capacity and reliability, Hadoop Distributed File System (HDFS) [3] is a storage system well suited to Big Data requirements. In combination with YARN [4], this system increases the data management capabilities of the Hadoop cluster and therefore allows petabytes of data to be processed efficiently. RAMSSES leverages on a distributed Spark cluster to take care of the batch processing part during the first stage. However, in most cases, raw data contains dirty records that need to be cleansed to increase the accuracy of the main processing modules. Two steps occur here, the system will infer missing data based on interpolation and expectation-maximization techniques. Moreover, dimension reduction techniques such as graph-based kernel PCA [5] will be attempted on the dataset to ease in decreasing the complexity of the dataset. Due to space constraints, the data cleansing stage will not be detailed in this paper.

Next, a distributed modelization unit will apply several time-series models on the cleansed/sampled data to put it in the right structure. This unit aims at finding specific outliers based on mathematical and statistical techniques that will leverage on the generated time-series structure. The outliers found will be used in ranking the features so that the system applies machine learning models only on the required attributes, thus significantly reducing the processing time. This method enforces a dynamic data allocation by optimizing the size in each packet of transferred data between HDFS and the Apache Spark engine. The extraction of the minimum required data for machine learning processing is handled by the semantic unit. After converting the reduced data into RDF, based on an ontology carefully designed to fit the use case, a query generator will select the minimum size graph using top ranked features based on a SPARQL query. This sub-graph will be sent to the main processing unit that leverages on Spark to run multiple anomaly detection algorithms in a distributed manner.

In order to select the right algorithm that fits the streams signature RAMSSES uses a complex set of rules based on statistics such as variables dependency or data distribution. After selecting the proper algorithm, we apply it on the output generated from running the previously mentioned SPARQL query on the RDF streams, resulting in finding anomalies in the streams. Finally, results found will be written on a messaging system, namely Apache Kafka, that will queue the messages in an ordered fashion to be used by a visualization tool.

3 Outlier Detection and Attribute Selection

Data processed by RAMSSES is associated with timestamps, making it a time-series set of values. To identify outliers accurately, we propose a simple but yet powerful pipeline based on a combination of time-series modelization and outliers detection models selected from established research papers [6,7]. RAMSSES builds a specific model used in computing an expected value at time T, then a number of errors E are computed by comparing the expected value with the actual value at time T. The system will automatically determine thresholds on E and outputs the most probable anomalies (Fig. 2).

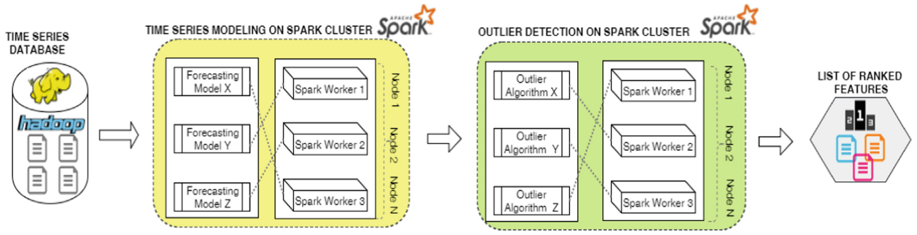

Fig. 2. Outlier detection process

First, our system needs to model received data into a proper time series shape based on a set of models such as the *Simple Exponential Smoothing Model* or the *Multiple Linear Regression Model*. Then, RAMSSES should detect outliers automatically using a set of algorithms selected amongst various libraries such as Yahoo's EGADS framework [8] such as the *Adaptive Kernel Density Change Point Detector* or the *K-Sigma Model*. The complete list of models used for time series modelization and outlier detection is available on github. The main criteria used to select these algorithms are:

1. *Genericity:* Since we aim at solving different use cases, the applied models should be able to adapt to various data shapes. Therefore, all selected models have been able to generate a proper time series and detect outliers against models that have been tested against 8 use cases (see datasets in Evaluation

Section). All the other models outputting errors or simply unable to produce clean results were discarded.

2. *Accuracy:* The results computed by all the models for each evaluation dataset were assessed using well know statistical functions [9] such as cross-amplitude function (CAF), cross-correlation function (CCF) and coherency function (COF). We discarded the models that showed abnormal or biased results for the 3 functions.
3. *Complexity:* Our system targets low latency and high speed performance by distributing the operations in a cluster of machines. Hence, for each model applied to every dataset in the evaluation, we assessed the average running time and selected the ones that produced the fastest results under 10 s (a constraint set with the partner for his dashboards that can be changed as a parameter).

Algorithm 1 Feature Ranking

```
                 Input: Tabular dataset D
      Output: HashMap ranked features
 1: Initialize HashMap out.
 2: Let $COL$ be a column from the dataset D
 3: for all $COL$ do
 4:     let $anomsMap$ be a HashMap to hold anomalies index and count
 5:     let $TSM$ be one of the time series building models.
 6:     for all $TSM$ do
 7:         build $COL$ time series using $TSM$
 8:         let $AD$ be one of the outliers detection models.
 9:         for all $AD$ do
10:             $List < anomalies >$list= AD.run($TSM$) { run AD against $TSM$}
11:             let $anomaly$ be anomaly in $list$
12:             for all $anomaly$ do
13:                 anomsMap.put(anomaly.getIndex(),
                    anomsMap.get(anomaly.getIndex())+1)
14:             end for
15:         end for
16:     end for
17:     $maxCount$ = anomsMap.getCommonAnomaliesMaxCount()
18:     $out.put$(COL.getName, maxCount)
19: end for
20: order $out$ ASCENDING based on count value
21: Export $out$ to CSV
```

Algorithm 1 illustrates the overall process of feature selection. After building the new time-series structure based on the models available in our system, all algorithms from the outlier detection module are run in order to obtain a list of ranked features, *i.e.*, the columns with the highest proportion of outliers. Therefore, we generate a new abstract outlier object including its index, value, and time-stamp. Finally, we count the number of common outliers obtained from running all algorithm combinations on every time-series model. The result of this process is a number of outliers for each of the features. The features are ranked depending on these count values. Results are then aggregated in a file and forwarded to the semantic query generator component that will use it as a base in the WHERE clause of the SPARQL query.

4 Query Generator

The query generation process creates relevant, precise SPARQL queries without any end-user intervention. These queries are executed continuously on the speed layer of our lambda architecture to detect anomalies automatically. The inputs of this generator are: **(i)** A text file containing ranked features. **(ii)** data streams which correspond to RDF graphs. It will be on such elements that the query will be executed. **(iii)** The static knowledge base of the project, *i.e.*, domain knowledge. Since streams are generally very compact, redundant knowledge (*e.g.*, geographical coordinates) are stored in a static way. **(iv)** The terminology (Tbox) used by the static and dynamic graphs (*i.e.*, concept and property hierarchies used in the domain vocabulary).

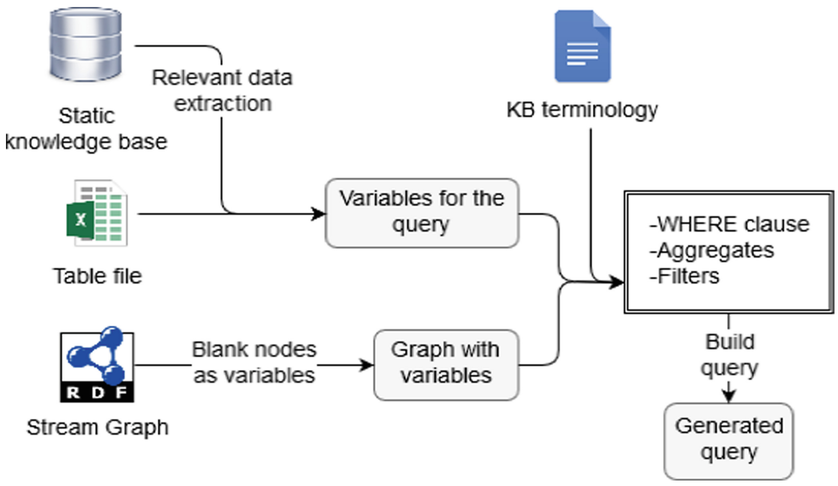

Fig. 3. Continuous query generation process

Figure 3 presents an overview of the query generation system. The main part of the query, the most difficult to build, is the WHERE clause. We initialize it using the dynamic graph coming from the stream, and we replace all blank nodes by variables. It is an easy operations, which gives us a valid clause, but way too imprecise; we need to specify it much more. Our next step is to identify what are the variables of the query SELECT clause. For this purpose, we parse the ranked features file and identify the information that can be associated to our graph URI. The matching looks for both properties and objects: the former represents relevant links between concepts, the latter gives the variables to be fetched by the SELECT part of the query. For the objects identified, we replace the literal in the graph by a variable, and add it to the SELECT clause (it is always a literal, since it is stored in a column of the input file).

At this stage we know there is a subgraph of our actual WHERE clause, composed of the interlinked triples containing the URI of the elements matched from the ranked features file, which is relevant. However, we still have to decide which ones of the remaining triples are irrelevant. For each one linked to the subgraph, we identify three possibilities: **(i)** the triple's object is a leaf and has a data type property, **(ii)** the triple's object is a leaf and has an object property and **(iii)** the triple's object is not a leaf (it leads to a graph, and has an object property).

a. Input Stream (extract)

```
_:1 type ssn:Sensor
_:1 measures _:2
_:2 hasFlow _:3
_:2 hasUnit qudt.cubicMeterPerHour
_:3 hasValue "4.4"
_:3 timestamp "07:15:00"
```

b. Generated Query

```
SELECT ?value ?timestamp
WHERE {
?x1 hasFlow ?x2
?x2 hasValue ?value
?x2 timestamp ?timestamp
}
```

Fig. 4. Query generation based on a stream

In the first two cases, we have to identify if the triple's object is static or dynamic, by querying the static database. If the object is stored in the static base, we have a redundant information, and it is not required in the query. If not, it is relevant for the query. When the triple leads to another graph, we need to know if it is relevant for the query, or independent. Using the Tbox of the project, we can verify if the initial properties (at the root of the subgraph) are *disjoint* or *inverse*. In such cases, the whole subgraph can be discarded since it is not relevant for the query. In case the Tbox is incomplete (*i.e.*, missing concepts, properties or individuals), such method can be problematic. Also, if the properties are not disjoint, we have to deal with the whole subgraph, and check every triple which is a costly operation. To specify the query even more, it is possible to use mathematical methods to identify eventual schemes in the ranked features file. A simple example is presented in Fig. 4, but the query can be more complex.

5 Machine Learning

5.1 Clustering Algorithms and Configuration

RAMSSES aims at identifying unusual behaviors through monitoring sensor data streams. Abnormal events refer to events which deviate from those considered normal based on historical patterns. In this unsupervised context, i.e. data with no predefined categories, RAMSSES has to detect anomalies which have not been previously encountered. The process is based on estimating a model of typical behaviors from past observations and comparing current observations against this model. The system is meant to provide a list of efficient and generic algorithms that can adapt to various use cases.

K-means: This algorithm is used to find groups in data based on a variable k. It iteratively assigns data points to one of the k groups based on the features provided, and data points are clustered based on the features similarities. The result is a list of centroids which can be used to categorize new data and label it. Each centroid of a cluster is a collection of feature values which define the resulting groups. In anomaly detection use cases, the model is used to cluster new data based on euclidean distance between new data point and centroids of the model clusters. If this distance is further than a certain threshold for all of the available clusters, we consider this value to be an anomaly.

Bisecting K-means: It is a combination of k-means clustering and hierarchical clustering. It splits one cluster into two sub-clusters at each step by using K-means, instead of partitioning the data into clusters in each iteration, until k clusters are obtained. This algorithm is based on K-means but has the main advantage of being more efficient when k is large. Moreover, Bisecting K-means produce clusters of similar sizes, while K-means is known to produce clusters of widely different sizes. To detect an anomaly, the distance of incoming data points from centroids is calculated, and a data point is considered an anomaly if this distance was further than a certain threshold.

Gaussian Mixture Model (GMM): It is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think of mixture models as generalizing K-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussian. In anomaly detection use cases, this algorithm is used to find the probability of an incoming data point to belong to one of the model clusters, where the model was built using clean data with no anomalies or labeled anomaly data. If the resulting probability is less than 0.05% for all of the clusters of the model, the data point is considered an anomaly, since there is not enough confidence that this data point will be following one of the clusters.

5.2 Selecting Algorithms and Automation

We rely on historical batch data to select one of the aforementioned algorithms and apply it to the streaming data for anomalies retrieval. To achieve this result, multiple statistical data analysis are needed to choose automatically the proper algorithm without human intervention. The process is organized into 4 main steps:

Data Standardization: This concept arise when continuous independent variables are measured at different scales, which means that these variables do not contribute equally to the analysis. The idea is to rescale the original variable to have an equal range and/or variance. It is important to standardize variables before running machine learning algorithms because cluster analysis techniques depend on the concept of measuring the distance between the different observations we're trying to cluster. There are different techniques for standardizing data [10], our system uses Z-Score method since it is one of the most popular methods among data scientists. In this case, we rescale an original variable to have a mean of zero and standard deviation of one.

Distribution Type: Data distribution is a very important factor to decide which clustering algorithm should be used. For example, there are some clustering algorithms that works very well with a normal distribution (*e.g.*, Gaussian Mixture Model) where other performs very well for multinomial distributed data (*e.g.*, Latent Dirichlet allocation). To decide which algorithm to use, a distribution test is done on the historical dataset to check if the data is following a normal distribution or not. There are many methods to perform this test [11], the one chosen was the D'Agostino's K-squared test for its popular goodness-of-fit measure. It is based on transformations of the sample kurtosis and skewness, and has power only against the alternatives that the distribution is skewed and/or kurtic.

Variables Dependency: Finding the correlation is another important step when deciding which clustering algorithm should be used on certain data. Some models work well with dependent variables where other performs better with independent variables. Several methods can be used to determine variables dependency in a dataset, if it is a categorical variable, then Chi-Square test can be used, if it's a quantitative value, then the Simple Pearson Correlation can be used. As this work is focusing on quantitative anomaly detection, we'll only focus on Pearson Correlation method. Pearson correlation is a linear measure between two variables X and Y. It results in a value between $+1$ and -1, where 1 is a total positive linear correlation, 0 is no linear correlation, and -1 is a total negative linear correlation.

Dataset Complexity: A very useful statistic in deciding which clustering algorithm to choose is having a sense of the complexity of the data that includes data size and features space count. Some algorithms are really time-consuming due to the complex mathematics used and don't perform well with very large datasets. To select the proper algorithm, we are following a straight forward approach. Taking into consideration that Gaussian Mixture model will fail for high-dimensionality data, if the dataset has high dimensions (*i.e.*, larger than 50 dimensions) and larger than 50,000 records in terms of size, K-means based algorithms will be suggested.

6 Evaluation

6.1 Methodology and Datasets

All of the datasets used in the evaluation are real datasets except one that is generated. All of them are labeled and can be found in well-known benchmarks [12] except the auto-generated synthetic dataset and Yahoo production traffic dataset that can be found in specific research papers [13]. In order to evaluate our system, a cleaning process was performed by removing records that are labeled as anomalies, the resulting dataset was used to estimate the number of clusters. Then the model is built using the cleaned data with the following parameters: k as being the estimated number of clusters, *1000* as being the maximum number of iterations, and *1e−9* the distance threshold above which clusters centroids converge. Other parameters are default parameters of the Spark Machine Learning Library (Table 1).

Table 1. Evaluation datasets details [14]

Name	Description	Size	Dimensions	Anomalies
ALOI	Color image collection	50,000	27	3.02%
Shuttle	Space shuttle radiator positions	46,464	9	1.89%
Speech	Recorded English language	3,686	400	1.65%
Pen	45 writers handwritten digits	809	16	11.1%
KDD	Simulated IP attack traffic	620,089	38	0.17%
Thyroid	UCI medical ML repo	6,916	21	3.61%
YahooS5	Yahoo services metrics	1,441	1	7.49%
Synthetic	Artificially generated	4,000	2	3%

Finally, the original dataset that includes anomalies were run against the model, and the number of detected anomalies are recorded. This approach for evaluation was applied on a local machine and on a cluster for the sake of evaluating if the system is scalable and performs faster when it's distributed.

6.2 Clustering Evaluation

In this evaluation, all datasets are real world measures from different sources, each one is labeled to indicate clustering quality. Before discussing the results, we will explain the key metrics. We assume the standards semantics for True Positive, False Negative, True Negative and False Positive, resp. TP, FN, TN and FP. We evaluated RAMSSES against all the datasets. For each one, the system suggested an algorithm to use, but for the sake of critical comparison of the results, we included the results when using other algorithms to see if the suggested algorithm was the best choice or not. Table 2 lists the results obtained, within each group. The bold Entries represents results with suggested algorithm by the system.

As we can see in Table 2, some of the datasets had excellent results in terms or both accuracy and precision by using the algorithm suggested by RAMSSES. *Shuttle, Yahoo S5, and Pen* datasets recorded very good results for the clustering task and most of the real anomalies were detected. We can note as well that in datasets where the suggested algorithm is *Gaussian Mixture*, such as *Thyroid dataset*, the results were reasonably good when we consider both accuracy and precision, while in fact when using other algorithms, *i.e.*, K-Means, we have excellent accuracy result but very poor precision. Therefore, we can conclude that for many datasets, the system achieved an optimum for both accuracy and precision whenever an algorithm was suggested. The table highlights also the

Table 2. Clustering Results

Dataset	Est K	Algorithm	Accuracy	Precision	Error	Recall	F-Measure
ALOI	10	**K-Means**	**0.961**	**0.043**	**0.039**	**0.117**	**0.063**
	10	**Bi.K-Means**	**0.961**	**0.032**	**0.039**	**0.093**	**0.047**
	10	Gaussian Mixture	0.969	0.003	0.030	0.285	0.008
YahooS5	15	**K-Means**	**0.990**	**0.990**	**0.009**	**0.892**	**0.939**
	15	**Bi.K-Means**	**0.965**	**0.660**	**0.034**	**0.847**	**0.742**
	15	Gaussian Mixture	0.212	0.990	0.782	0.087	0.160
Thyroid	5	**Gaussian Mixture**	**0.605**	**0.632**	**0.394**	**0.056**	**0.104**
	5	K-Means	0.954	0.008	0.045	0.029	0.012
	5	Bi.K-Means	0.955	0.004	0.044	0.016	0.006
Shuttle	5	**K-Means**	**0.989**	**0.980**	**0.011**	**0.654**	**0.784**
	5	**Bi.K-Means**	**0.989**	**0.981**	**0.010**	**0.655**	**0.786**
	5	Gaussian Mixture	0.981	0.019	0.782	0.5	0.013
Speech	3	**K-Means**	**0.973**	**0.016**	**0.026**	**0.027**	**0.020**
	3	**Bi.K-Means**	**0.974**	**0.016**	**0.026**	**0.027**	**0.020**
	NA	NA	NA	NA	NA	NA	NA
Synthetic	3	**Gaussian Mixture**	**0.504**	**0.650**	**0.496**	**0.038**	**0.073**
	3	K-Means	0.982	0.725	0.018	0.696	0.710
	3	Bi.K-Means	0.975	0.5	0.024	0.612	0.550
Pen	5	**K-Means**	**0.954**	**0.667**	**0.045**	**0.895**	**0.764**
	5	**Bi.K-Means**	**0.961**	**0.733**	**0.038**	**0.904**	**0.810**
	5	Gaussian Mixture	0.111	0.99	0.888	0.111	0.200
KDD	10	**K-Means**	**0.989**	**0.476**	**0.011**	**0.075**	**0.129**
	10	**Bi.K-Means**	**0.989**	**0.476**	**0.011**	**0.075**	**0.129**
	10	Gaussian Mixture	0.957	0.99	0.043	0.038	0.074

fact that RAMSSES adjusted remarkably well with the *Synthetic datasets*, even though with less clustering quality. Finally, for *ALOI and Speech datasets*, we had very poor results in terms of precision for all used algorithms. Spark failed when trying to run the *Speech* dataset, explaining the absence of results on the Gaussian Mixture algorithm.

6.3 Performance Evaluation

In this section, we will show and discuss the running time for using Spark MLLib clustering algorithms. We first run on a local machine with maximum parallelism. Then we move to an Amazon Web Services (AWS) cluster to observe if RAMSSES can truly scale and perform better in a real distributed mode.

Local Evaluation: All of the clustering tasks using the three clustering algorithms were first run on a local machine whose specifications are: **(a)** MacBook Pro running macOS Sierra; **(b)** Intel Core i7 CPU 2,5 GHz with 4 cores; **(c)** 16 GB DDR3 RAM, **(d)** L2 Cache (per Core): 256 KB, L3 Cache: 6 MB; **(e)** 500 GB SSD; **(f)** Spark 2.0.0 with Java 8. Figure 5 shows the running time for different datasets using different algorithms. Please note that for the *KDD* dataset, when running *Gaussian Mixture* algorithm, the running time was *13146.025* s, which is approximately *3.65* h. So it was excluded from the graph because it will make other values unreadable due to graph scale.

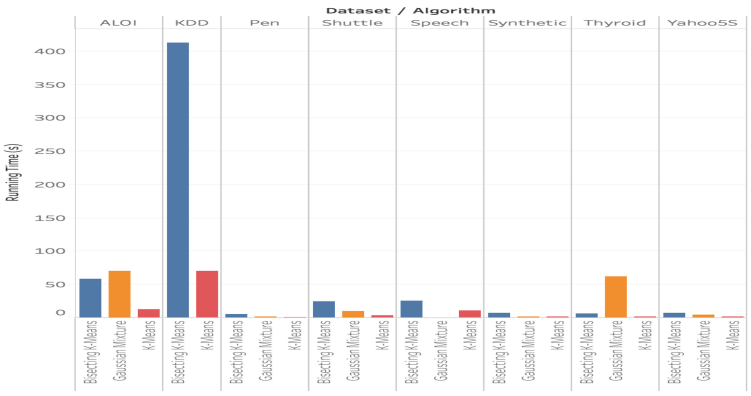

Fig. 5. Running time for datasets using different algorithms

Generally speaking, *Bisecting K-means* takes more time than *K-means*. Nonetheless, when the number of dimensions of the dataset gets higher, *Gaussian Mixture* gets a lower running time performance, thus becoming the slowest algorithm among the three for large datasets.

Cluster Evaluation: In this part, we will evaluate if RAMSSES truly performs better in terms of running time due to it's distributed nature. We used *KDD* dataset as it's the largest dataset we have, and we can notice running time difference when distributing the clustering process. We will compare it with local performance using the previously mentioned MacBook Pro with full utilization of its 4 cores. Amazon AWS service was used to create a cluster with one master node and four workers (machines), all of them of type EMR m3.xlarge, with the following specifications: **(a)** Intel Xeon E5-2670 2.6 GHz Processors, 4 cores;

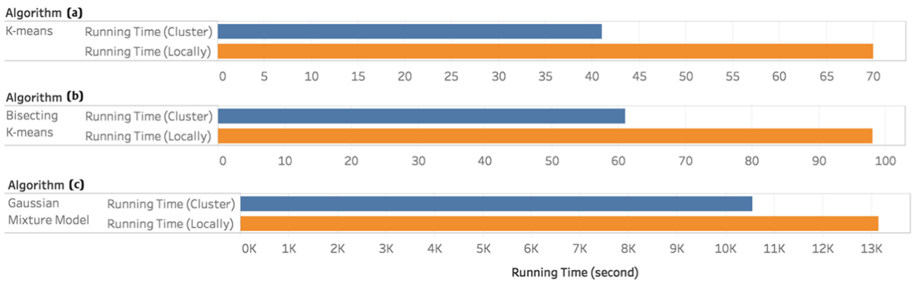

Fig. 6. Local vs cluster running time for ML algorithm

(b) 15 GB of RAM; **(c)** 2×40 GB SSD; **(d)** Spark 2.0.0 with Java 8. As shown by Fig. 6, we had 41.4%, 37.75% and 19.76% performance enhancements for K-Means, Bisecting K-Means and Gaussian Mixture algorithms respectively. These tests demonstrate that RAMSSES is able to scale and perform better when distributed.

6.4 Continuous Query Generator Evaluation

We based our evaluation on two parameters: the relevance of the continuous queries, and their impact on the processed streams. The tests have been realized on sensor measures archived by our partner; due to their proprietary nature, the dataset are not made public. They represent a year of measures over 3 different geographical zones (2 French ones and one international). For each zone, we have records for 15 different measure types, stored in RDF files and organized by year of emission. In total, one million of each type of measure is archived per year, for a size on disk of 3.4Go.

The validation of the query generator has been performed using panels of 5 domain experts from the partner company. Intuitively, we presented them a set of about ten queries over the 3 zones, concerning different types of measures, and asked them if the queries would retrieve information that could help in detecting anomalies. For the simplest queries, the output was judged extremely satisfying (95% of the queries were judged as relevant). With queries implying more parameters, some imprecisions may occur, but after discussion we found out that adjusting the relevant parameters in the input file solves most of the problems. The general method consists in changing the labels of the columns, so they can be identified as ontological concepts more easily. The main issue remains with more complex queries, where some specifications such as aggregates and additional parameters need to be added to the query.

We consider now a real-world Waves stream example. The inputs coming from the stream (for example in Fig. 4a) are composed of RDF triples coming from each sensor. We also have a file listing the relevant elements that need to be conserved to build the query. We use this file to identify the triples that are essential for the query building, generating a WHERE clause with a subgraph including all of them (Fig. 4b). This list of elements can be built using tabular files in our use case. We can associate the label of the columns to some of the properties in our ontology (in red in the figure), and associate the values in the columns to objects (in blue). These values will represent the variables in our query, and the labels will only be triples to specify the WHERE clause. In our example, we have three triples extracted that are linked, and thus we have reduced the graph by half its original size. In this example it only represents three triples, but with the number of sensors, the different measures types, and the frequency of the measures, it can make a significant improvement. Other use cases can get different gains depending on the stream triples and the list of relevant elements.

Table 3. Query generator performance improvement

Zones	Measure	Triples	Size
Geographical zone 1 (France)	75%	50%	29%
Geographical zone 2 (France)	66%	52%	32%
Geographical zone 3 (Asia)	72%	46%	27%

Table 3 presents the gains for each zone with queries generated on the archives: the first column gives the percentage of removed measure types, the 'Triples' column provides the number of triples from the stream retained for the query processing. Finally, the last column represents the gain in size of retrieved streams from our query (*e.g.*, in zone 1, 29% of the 1 million triples streamed were processed by our query). The validation of the generated query has been performed with the help of domain experts for our use case. We generated and executed our query using the data they gave us, and presented our results to their appreciation. The results are presented in Table 4, for different queries.

Table 4. Domain experts evaluation

Evaluator	Queries						
	1	2	3	4	5	6	7
1	✓	✓	✓	✓	✓	×	✓
2	✓	✓	✓	✓	×	✓	×
3	✓	✓	✓	×	✓	×	×
4	✓	✓	✓	✓	✓	×	×
5	✓	✓	✓	✓	✓	✓	×

The three first ones are variations of the example presented previously, adapted to different kinds of measures, and were very satisfying. The two next one were more specific queries, which is also very satisfying: one expert complained about some missing parameter, but it could be added to the result by adapting the input data to the tabular file. The last queries were the most tricky, and aimed at obtaining a very specific result using some values from a specific cell (thus needing some filtering). About half of the evaluators judged the precision satisfying, the other half evaluated it insufficient. As we mentioned, the main difficulty for our query generation system is to build aggregates using the input data, and this is our main focus for now.

7 Conclusion

In this paper, we presented RAMSSES, a scalable system enabling the detection of anomalies on real time streams using a mix of machine learning techniques and a semantic web approach. Leveraging on the statistical characteristics of historical data, the system relies on a complex set of rules to select the best algorithm that can be applied to find data singularities. The features identified as potential support for anomaly detection then need to be retrieved from data streams. The queries that are retrieving these information are automatically generated in the form of SPARQL queries since streams are modeled as RDF graphs. The answer sets of these are then processed by machine learning methods to detect anomalies in close to real time.

To the best of our knowledge, RAMSSES is the first system aiming to automatize the complete process of dimension reduction, feature selection, query generation and anomaly detection. A task that is generally performed manually by end-users needing some expertise in both the domain of application and machine learning. Computing on a distributed environment to handle massive data, RAMSSES proposes the algorithm that could yield the best results, therefore we also evaluated its performance and scalability both locally and on a real cluster. RAMSSES is being used in production at one of the Waves's partner. Some of our future works concern the integration of third party libraries to support clustering algorithms that are not yet supported by Spark, support for textual data (currently, RAMSSES only deals with numerical data), tackling expressive ontologies (up to the OWLQL and OWLRL profiles) in our query generator component.

References

1. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: a survey. ACM Comput. Surv. **41**, 15:1–15:58 (2009)
2. Nathan Marz, J.W.: Big Data Principles and best practices of scalable realtime data systems. Manning, 1st edn. (2015)
3. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST 2010, Washington, DC, USA, pp. 1–10. IEEE Computer Society (2010)
4. Vavilapalli, V.K.: Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC 2013, New York, NY, USA, pp. 5:1–5:16. ACM (2013)
5. Boutsidis, C., Mahoney, M.W., Drineas, P.: Unsupervised feature selection for principal components analysis. In: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2008, pp. 61–69. ACM (2008)
6. S. C. from the ITISE Conference, Time Series Analysis and Forecasting. Springer (2016)
7. Aggarwal, C.C., Reddy, C.K.: Data Clustering: Algorithms and Applications. Chapman & Hall/CRC, 1st edn. (2013)

8. Laptev, N., Amizadeh, S., Flint, I.: Generic and scalable framework for automated time-series anomaly detection. In: Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1939–1947. ACM (2015)
9. Box, G.E.P., Jenkins, G.: Time Series Analysis. Forecasting and Control. Holden-Day, Incorporated (1990)
10. Trebuna, J.P., Fil'o, M.: The importance of normalization and standardization in the process of clustering. In: Proceedings of IEEE 12th International Symposium on Applied Machine Intelligence and Informatics, SAMI (2014)
11. Yazici, B., Asma, S.: A comparison of various tests of normality, vol. 77, pp. 175–183, February 2007
12. Goldstein, M.: Unsupervised anomaly detection benchmark (2015)
13. YAHOO, S5 - a labeled anomaly detection dataset (2015)