Making sense of data streams: Complex Event Processing for Controls Applications

August 2014

Author:
Kacper B. Sokol

Supervisor(s):
Filippo Tilaro
Axel Voitier

CERN openlab Summer Student Report 2014

Project Specification

CERN is currently investigating the usage of data analysis technologies to study the behaviour of the industrial control systems. An activity related to these analysis is using *Complex Event Processing* (CEP) tools to classify a real abnormal behaviour from one generated by a human intervention on the system.
In this study the Complex Event Processing classification is run over signals produced by variety of sensors and simulators. The selected tools is `Esper`. Presented here work consists of installing chosen tool, and developing the classification system with it to address given above merit.

Abstract

This project aims at building a tool to process live streams of data produced by various sensors and artificial generators. To this end, a `Java` code is written, which uses `Esper` *Complex Event Processing* package to: receive data feeds, apply user defined rules and filters, and pass the resulting information to a clustering framework.
The last step employs *Affinity Propagation* based clustering algorithm, which choice is motivated by its dynamic adaptation to number of clusters in the data. This is key feature in data streaming scenario as the number of clusters can evolve with time. Furthermore, [Zhang et al., 2013] have documented overall good performance of Affinity Propagation in cases of live data analysis.
Finally, presented here approach is compared and contrasted against static clustering algorithms applied to data gathered from streams incoming over one run of the program, followed by in-depth results analysis.

Keywords: Esper, Complex Event Processing, Affinity Propagation, data streams

This report was written in LaTeX.

Table of Contents

1 Introduction

Physicists at CERN are mainly concerned with event reconstruction. This implies collecting data first and then processing them. It is not possible with sensors data like pressure, or temperature. In case of abnormal behaviour, e.g. overheating, the action need to be taken instantly.
This study presents an approach to handle live data streams and analyse them "on the fly" to differentiate between noisy feed and real threats, hence produce human readable data inferences.

In machine learning, unsupervised clustering aims at discovering cluster structure underpinning the data ([Flach, 2012] extensively describes main concepts of machine learning). There are many state-of-the-art algorithms designed to solve this problem, nevertheless, majority of them deals with static data, forcing user to define many parameters prior to classification. The example of these might be a predefined number of clusters. Also, many of these algorithms need data with static distribution, and what is more static dataset, i.e. not changing with time.
The major study addressing these issues is presented by [Zhang et al., 2013], who proposed clustering algorithm based on affinity propagation scheme that solves all of the above concerns: it adapts to patterns evolving in data by tracking current number of clusters, therefore it handles non-stationary data distribution; furthermore it can be easily adjusted to data streams.

I aim at extending presented by [Zhang et al., 2013] approach with `Esper` framework (`Esper` engine is briefly introduced by [Marinescu, 2006]). My concept facilitate modular feature extraction: user can change signal characteristics of interest while application is running; and it gives all advantages of time windowing provided by `Esper`.
I construct feature extraction system based on `EPL` statements and clustering solution capable of handling live data stream.

1.1 What is *Complex Event Processing*

Complex Event Processing (CEP) is a software family facilitating complex analysis of high throughput live data feeds.
The concept behind it is similar to database queries, but instead of interacting with static data pool the extraction is done on live streams. It gives possibility of applying filters, functions, and statistical analysis to chosen part of signal by *querying* it. [Etzion and Niblett, 2010] presents comprehensive introduction to the topic.

1.2 What is `Esper`

`Esper` is a flagship software of Complex Event Processing tools family. It is capable of handling and analysing multiple independent incoming data streams. The main goal of analysis is to trigger user defined actions, e.g. if the temperature feed is above some level for given amount of time an additional cooling system can be deployed.

`Esper`'s main advantage is *time windowing*, which allows to focus on a specific time period. The windowing can be done in multiple modes:

Time incoming data from last t milliseconds/seconds/minutes/etc. are processed—sliding window.

Time batch data are processed in t milliseconds/seconds/minutes/etc. batches i.e. algorithm collects data for t units of time, then process them, and repeats this cycle—fixed length interval windows e.g. $\{[0,t);[t,2t);[2t,3t),...\}$ where 0 point is the start of our analysis.

Length n most recent events are processed—the order of arrivals is the only quantity of interest.

Moreover, `Esper` is capable of joining multiple, independent, asynchronous (events form one stream can be lively processed regardless of others) incoming signals for processing purposes. With proprietary `EPL` querying language the analysis is as simple as writing number of human readable statements.
One approach is to hard-code the EPL queries in the application but it is more convenient to provide them as a module (external plain text file) what gives flexibility of changing the queries without stopping the application.

Finally, `Esper` is available as a `Java` and `.NET` framework hence it is multi-platform and easy to incorporate into any application. It can work as both: local and server programme, where later solution facilitates on-the-fly changes like rules injection via AIP.

1.3 Why to use it

The major advantage of `Esper` is ability to process millions of events per second with low computational complexity and no time consuming read/write disk operations.
The data processing is describe by developers as "**4-D**" concept:

***D*etect** events of interest.

***D*erive** events complying with specified rules.

***D*ecide** what to do based on gathered evidences—data.

***D*o** the action bonded with occurring event.

1.3.1 Complex Event Processing at CERN

I adopt the "**4-D**" flow to my application as follows:

- *detect* incoming signals from the sensors (generators);
- *derive* the specified features from the signal;
- *decide* to send the features to clustering algorithm; and
- *do* the classification.

1.4 Applications

`Esper` is mainly used in high throughput data analysis services, where processing must be flexible and adaptable to constantly changing environment. The following examples show different aspects of *Complex Event Processing*.

1.4.1 Nuclear Power Plant

While managing nuclear power station there are plenty of components whose malfunctioning may lead to a disaster, for example:

- to high core temperature,
- failure of cooling pumps, or
- abnormal seismic movements.

Only specific combination of these factors states should raise an alarm. For instance low flow of cooling liquid and constantly raising core temperature.
`Esper` is a great fit for this scenario. It allows to view the sensors readings in different time windows hence discover long and short term patterns. For instance, if the temperature raises slowly the trend will not be visible in last hour frame, but it will appear in week or month window. Discovering this long term property may prevent a disaster.

1.4.2 Stock market exchange

In this example we consider signals as all the real time price processes injected to `Esper`. We begin by filtering them with EPL statements to retrieve stocks of interest. Then, we use our financial knowledge to create inferences between them and perform complex analysis of price processes. We can also use weather forecast and news feed to gather some background. Combining all these rules can trigger market actions like: shares quantity, and buy/sell order; therefore increasing bidding speed and automatizing trading. Other approach could be to extract market statistics which can be used by financial advisers.

2 Model of processing

In this section I describe a model of processing used in the application, and presented in *Figure 1*.

We divide the model into two main branches: *off-line resources* and *live data processing*.
The first one consists of

- `EPL` file (see *Listing 1*), that defines the features to be extracted form the incoming signal (rules);
- `CSV` file which serves as a data repository—signals are saved here for future analysis; and finally
- data reconstruction script written in `Python`: `visualize.py`, which uses `CSV` data repository to plot signal and all extracted features against time as well as features against each other.

Live data processing module consists of the following components:

Signals layer includes signal generators and signal feeds interfaces.

`Esper` layer collects signals, and applies to them rules taken form `EPL` file. It also saves signals, extracted features, and time stamps to `CSV` repository. Finally, it performs defined actions, in this particular application it supplies them to clustering algorithm.

Events layer transfers data (events) to clustering module.

Affinity Propagation Clustering layer receives events and performs clustering on live data stream. Once a datum point is assigned to cluster, this information is appended to `CSV` repository.

2.1 Feeds and generators

A number of signal generators is implemented to test the framework:

- *sine* wave with normally distributed noise,
- *cosine* wave with normally distributed noise,
- *normal* distribution generator,
- *uniform* distribution generator,
- *multivariate normal* distribution generator.

The time between generated samples is modelled according to *Poisson* distribution—it simulates pseudo-random signal occurrence.

The package also contains module which collects temperature readings from the electronic thermometer placed at CERN Prévessin site, and Yahoo! weather forecast feed for the same location. Both of them can be used as signal providers for the application.

In *Esper* the easiest (and the one that I use) way to represent a signal is *POJO*—Plain Old Java Object. Such object contains some properties—constants—like `currentValue` and `timeStamp`. Every event arriving from source is or can be converted to a POJO. Such container holding signal properties is delivered to the processing unit of `Esper` via *listeners*.

2.2 `Esper` engine

The `Esper` engine runs in the background gathering signals from all indicated by user (see §2.3) sources. It filters all these incoming data with `EPL` rules and delivers desired ones to corresponding listener (see §2.4).
The `Esper` framework is capable of running multiple servers simultaneously. It also allows to make changes to model of processing without stopping the server via provided API. This feature significantly increases usability of `Esper` as it prevents any data lose and allows fine-tuning.

2.3 Rules: feature extraction

In presented model *rules* decide what measures of data feed the user is interested in. They allow to get the value of the signal and apply built-in or **user-defined** filters on-the-fly.
`Esper` allows to either *hard-code* the rules in a source code of a program, or read them in from external `EPL` file, alternatively they can be sent to the engine via API.
The `EPL` language is similar to database queries, but instead of fixed pool it uses part of stream selected by time-window.

In my application I use rules as *signal feature extractors*. In particular I am interested in:

mean signal average in given time period $\bar{v}$,

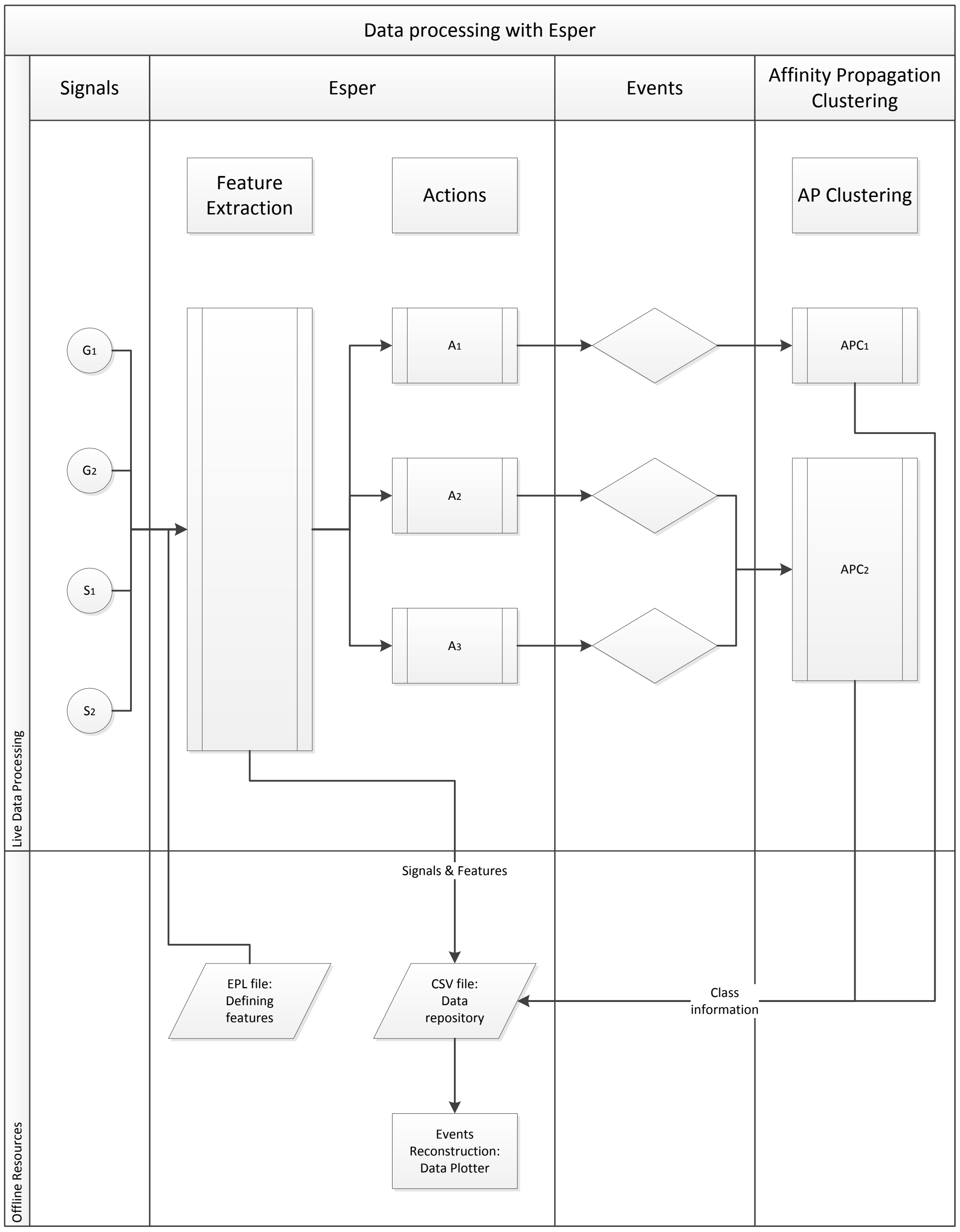


Figure 1: Data processing model.

standard deviation standard deviation in given time period v_σ,

k**-lag** a difference between current value of the signal and n$^{\text{th}}$ previous value i.e. $v_t - v_{t-n}$,

threshold a user-defined function which thresholds the signal based on a single value $\mathscr{V}$,

current value a current value of the signal v_t,

`FN` a number of defined features,

`TS` an assigns time-stamp t of received signal.

The example of rules extracting above features are placed in `EPL` file shown in *Listing* 1. The `EPL` file consists of (in preserved order):

- module name,
- dependencies that need to be imported,
- alias definitions i.e. short names for signals,
- rule name,
- rule description,
- rule body.

Generic rule starts with a `select` keyword. Then, features are extracted by applying functions to current signal value. For simplicity, each extracted feature is assigned an alias with `as` keyword. To clarify, in example given below variables `current` and `timer` are members of `randomGenerators.Sine` class—they are signal properties. Finally, `from` keyword is used to indicate which signal is used and `win:time(60 sec)` is placed to indicate time window of interest.
For convenience each file can contain multiple rules.

```
module
  SpringfieldNuclearPowerPlant.engine.ExternalFeatureExtractor;

import
  featureExtractors.FeatureExtractor;

create schema SinTick  as randomGenerators.Sine;

@Name('Basic——Statistics ')
@Description('Extract basic signal statistics to features')
select
avg(current) as F1,
stddev(current) as F2,
featureExtractors.FeatureExtractor.posNeg
  ( (current – prev(1, current)) ) as F3,
featureExtractors.FeatureExtractor.posNeg
  ( (current – prev(2, current)) ) as F4,
current as F5,
featureExtractors.FeatureExtractor.threshold
  (current) as F6,
6 as FN,
timer as TS
from SinTick.win:time(60 sec);
```

Listing 1: EPL file example.

2.4 Listeners for change

A listener is a thread running in the background and acting upon arrival of new event—signal tick—according to some attached rule.
`Esper` gives user a flexibility to define multiple rules and listeners, and pair them in non-restrictive manner e.g. :

- multiple listeners are attached to a single rule,
- one listener is attached to multiple rules,
- one listener is bonded with one rule.

I use this module to pass features extracted from signals to clustering framework (see §2.5 for reference).

2.5 Clustering with Affinity Propagation

2.5.1 Live stream adaptation of clustering algorithms

[Zhang et al., 2013] presented a general framework for adapting an unsupervised clustering algorithm to data streams. Their approach can be used with any clustering algorithm that does not need a fixed, predefined number of clusters as a model parameter. This restriction motivates their use of *Affinity Propagation* unsupervised clustering which is presented in more details in *§*2.5.2.

The overview of their approach is presented in *Figure* 2.

The process begins by feeding the framework with stream of extracted features. The algorithm uses initial batch of data (user defined, fixed number of initialization points) to create first model. Once the model is initialized, all incoming points are tested for fitting the model: if they do, they are appended to current model, and if they do not they are stored in *reservoir*. Fitness test checks whether given point is an outlier with regard to current model.
There are two possibles events that can trigger model rebuild: full reservoir—its fixed size is defined by user, or positive outcome of *change point detection* test on incoming stream—it detects "significant" and "sudden" change on the input stream. If any of these triggers fire, the model is rebuilt with data from reservoir.

The detailed description of outlined approach is presented in *Algorithm* 1.

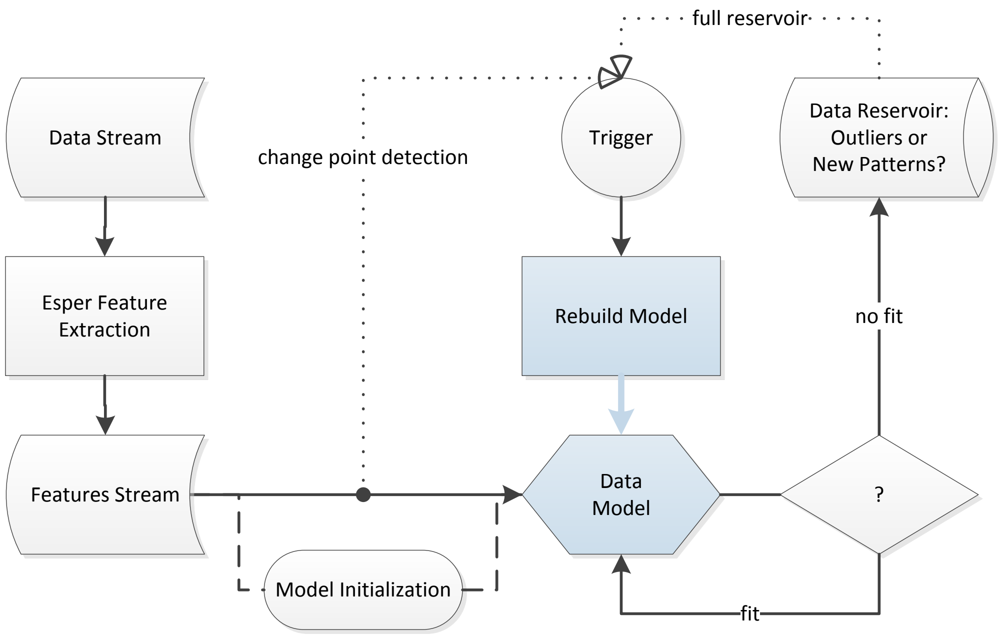


Figure 2: Stream clustering model.

```
Data: Data stream ...$x_t, x_{t+1}, x_{t+2}$, ...;number of initialization points $T$; reservoir size $r$; fit
      threshold $\epsilon$.
/* Initialization                                                    */
m ←APModel($x_1, ..., x_T$);
$Reservoir$ ←{};
/* Receiving data                                                    */
for $t > T$ do
    Compute $e_i$ = nearest exemplar to $x_t$ ;
    if $d(x_t, e_i) < \epsilon$ then
        Update model $m$ ;
    else
        $Reservoir$ ←$x_t$;
    end
    Rebuild Trigger ←( $|Reservoir| > r$ || Change Point Detection on input stream );
    if Rebuild Trigger then
        Rebuild model $m$;
        $Reservoir$ ←{};
    end
end
```

Algorithm 1: Streaming Affinity Propagation Clustering Algorithm.

2.5.2 Affinity Propagation

The predominant part of my application is *Affinity Propagation* module—an unsupervised clustering algorithm adapted to data streams as described above (§2.5.1).

Imagine two dimensional clustering task. It can be described as finding *clouds* of points which are similar. If we choose an Euclidean distance as our similarity measure the task is to locate areas on $\mathbb{R}^2$ plain where points lie close together.
Each cluster can be represented by a *medoid* or a *centroid*. The first is a central datum point of a "cloud" that *belongs* to the data set; and the later is a point on a plane which describes "centre of gravity" of the "cloud".

Affinity Propagation is a state-of-the-art clustering solution based on concept of message passing between data points. The algorithm uses notion of distance or similarity between any two points to find centroids of clusters. To this end, it maintains two matrices: R—*responsibility matrix* and A—*availability matrix*. Entries r_{ij} of the first one quantify how good does a datum point x_j act as an exemplar for x_i. Matrix A with elements a_{ij} describes how well does x_i fit to the model while being assigned to exemplar x_j.
Both matrices are initialised to 0 and iteratively updated by passing messages between each other. Once they are ready, one can retrieve from them the most probable centroids and all points assigned to them. For more details please refer to [Frey and Dueck, 2007].

2.5.3 Making sense of signals

At this stage of processing we gain valuable information about the signal: its current status. Imagine a noisy sine wave with possible flat state, which can be described with three parameters: *value low/medium/high*, *threshold negative/0/positive*, and *direction decreasing/steady/increasing*. Such classification yields 27 distinctive states which we want to predict. The main difficulty in presented scenario which prevents from simple fixed-value thresholding is lack of knowledge what value of the signal is low and what is high as it can evolve with time. Also in some cases it might be tricky to predict the direction of the signal: distinguishing between noise and trend.
Another, simplistic inference that we may want to do is: *behaviour normal/abnormal*. In this case we can either combine 27 named above states into 2 groups or we can perform separate classification.

Being able to build a model which can distinguish different state of signal based on complex features gives the user significant advantage over raw stream analysis. With such information user can write simple rule to trigger different actions based on current signal status.

2.6 Data visualization

To visualize gathered data, I wrote a separate `Python` script. It uses `CSV` repository generated by the main program to show the signal and its defined features as functions of time. It is also capable of plotting in "re-play" mode, where timestamps are used to model pause between plotting two adjacent points.
The script can also be used to scatter a plot of selected features against each other.

2.7 Static clustering

To analyse the quality of my experimental results (final stage of live clustering) I compare them with results obtained via static clustering performed on data gathered in `CSV` repository. For this purpose I use k-means, expectation-maximization, and affinity propagation algorithms. First two are implemented in `Weka` machine learning tool-kit and the last one is a part of *APCluster* package for `R` language.

3 Results

This section presents result gathered over a short run of signal analysis with feed generated as *noisy sine wave*. *Figure* 3 shows the waveform.

We begin signal processing by feature extraction. The sample of *average*, *standard deviation*, and *5-Lag* features extracted in *60 seconds time-sliding-window* are presented in *Figure* 4.

Once the features are extracted, they are sent to affinity propagation clustering algorithm, which runs in the background. Sample results of clustering can be seen in *Figure* 5, where separate cluster have distinctive colours.
The main idea behind signals clustering in system controls is to either detect abnormal behaviour or recognize the current signal status.

The first one can be understood as points appearing outside of main cluster (see *Figures* 5a, 5b , and 5c). In both figures, clusters were pruned to produce eye-friendly plots. In case of presented here sine

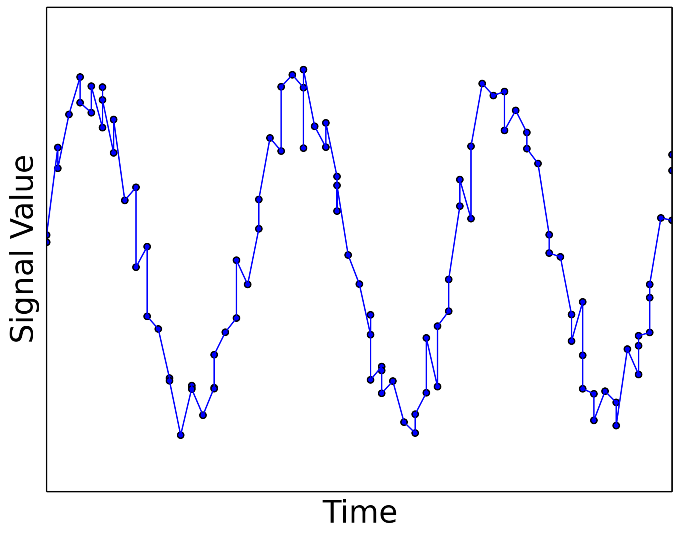
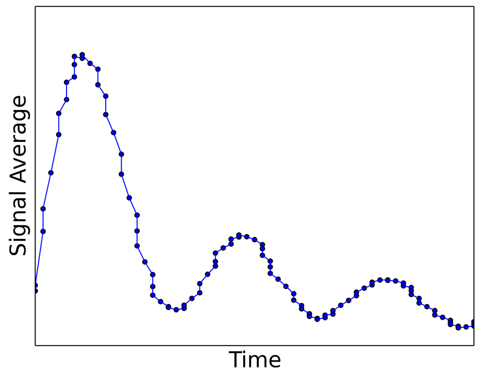
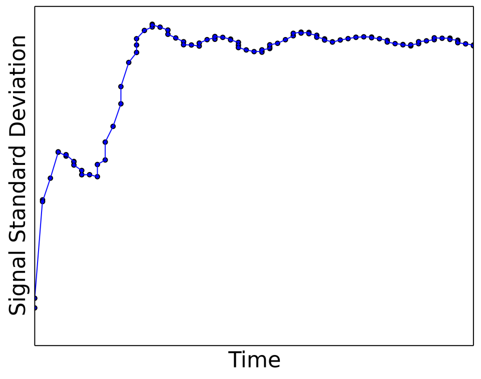


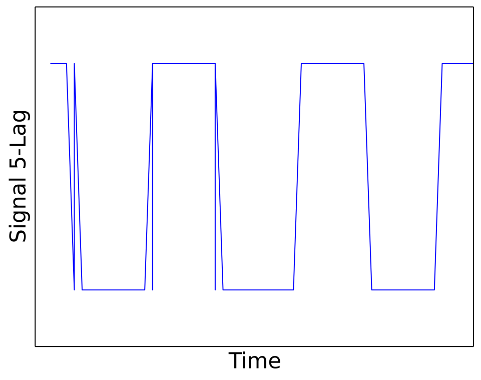
Figure 3: Generated signal sample—noisy sine wave.



(a) Signal average in time.



(b) Signal standard deviation in time.



(c) Signal 5-Lag in time.

Figure 4: Features extracted form signal.

wave the abnormal behaviour can be caused by sudden change of *period* or stretch in y-axis dimension—in graphs visible as detour from the main concentration of points and caused by the 60-seconds time-window not being filled yet with events at the very beginning of analysis.
In all of these graphs we consider *red* as "healthy" cluster, and *blue* as "suspicious" state. If new-coming points are assigned to the first group the system works smoothly, but once stream is being shifted towards blue zone the alarm is raised.

The current signal status recognition (see *Figure* 5d) is based on classification with respect to 3 distinctive measures based on extracted features:

- increasing—decreasing: k-lag being positive or negative,
- positive—negative: the sign of current signal value,
- high—low: inference made by clustering algorithm based on absolute value of current signal value.

Figure 5d shows result of clustering, where each colour corresponds to combination of listed above signal states e.g. *green* is *decreasing-negative-low*.

In both mentioned above approaches to noisy sine wave analysis implemented algorithm performs well and gives enough precision to be useful in real life applications.

3.1 Comparison with static clustering

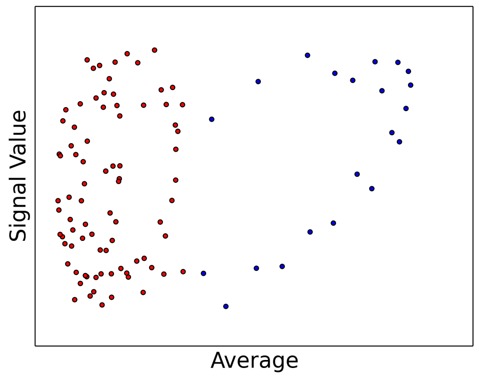
To benchmark above results I use static clustering on data repository created by one run of my program. Static clustering algorithms are: *k-means*, *expectation maximization* (EM), and *affinity propagation*; first two implemented in `WEKA` machine learning package, and the last being extension for `R` language.
The choice of the first one is motivated by its simplicity and popularity, nevertheless it needs fixing the number of clusters therefore its application in my case is not realistic. Two later ones do not need aforementioned parameter hence are more suitable for my approach.

Figures 6, 7, and 8 present 3 static approaches to good/bad state signal classification. Once pruned (similar clusters are joined) these outcomes become very similar to results in live clustering case. This indicates good performance of proposed in this paper approach.

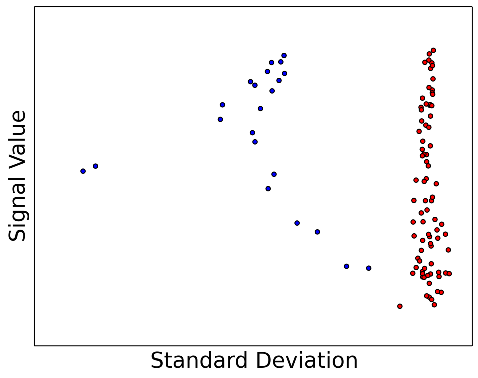
Graph 9 presents signal classification with respect to 8 presented above categories. We can see that static clustering struggles with differentiating between *high* and *low* signal values. Live clustering over-performs in this scenario probably because it sees the data as they come (one-by-one) therefore slightly simplifying the classification.

3.2 Overall performance

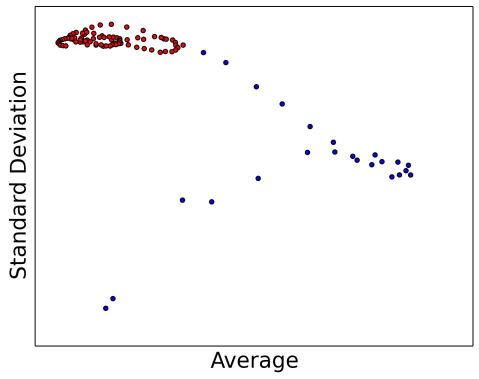
All presented above experiments indicate that proposed in this paper live stream clustering framework performs at least as well as static approach. These results show that my approach is worth further investigation and should be tested more carefully with use of real-world data streams to determine its capabilities and possible applications.



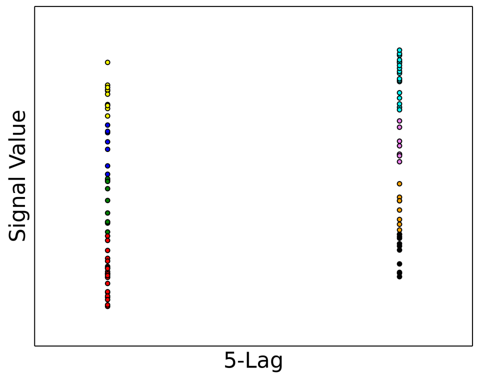
(a) Average clustering.



(b) Standard deviation clustering.

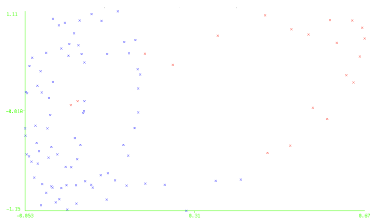


(c) Average—Standard Deviation clustering.

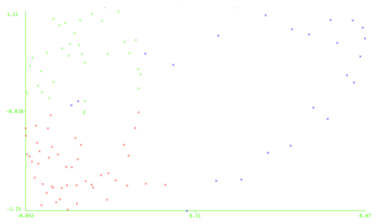


(d) 5-Lag clustering.

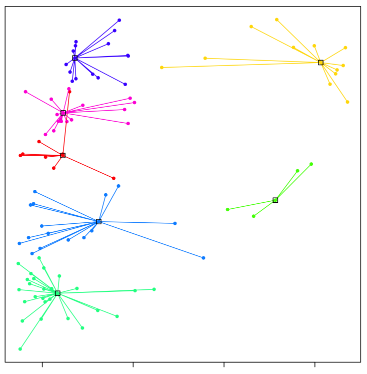
Figure 5: Signal clustering based on extracted features.



(a) 2-means clustering.

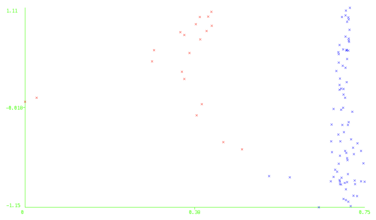


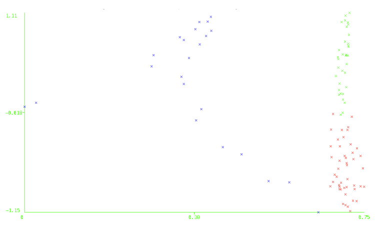
(b) EM clustering.



(c) Affinity Propagation clustering.

Figure 6: Signal Value vs. Average clustering.


(a) 2-means clustering.


(b) EM clustering.

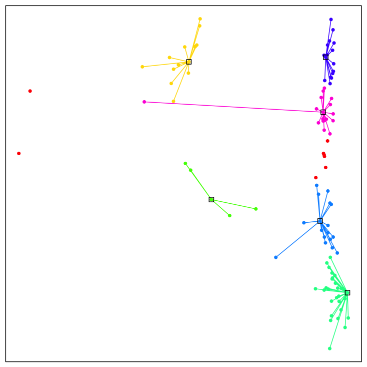
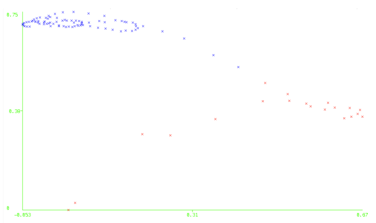
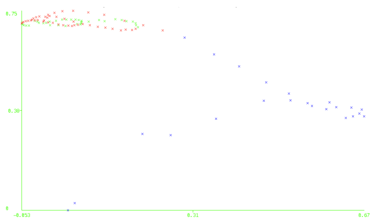

(c) Affinity Propagation clustering.

Figure 7: Signal Value vs. Standard Deviation clustering.


(a) 2-means clustering.


(b) EM clustering.

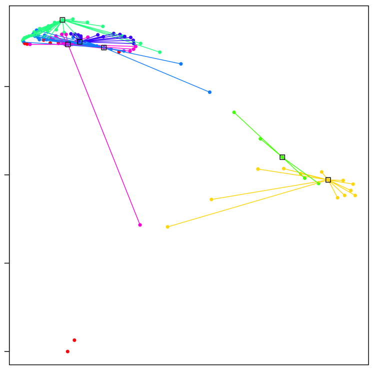
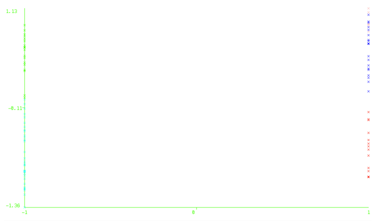

(c) Affinity Propagation clustering.

Figure 8: Average vs. Standard Deviation clustering.

(a) 2-means clustering.


(b) EM clustering.

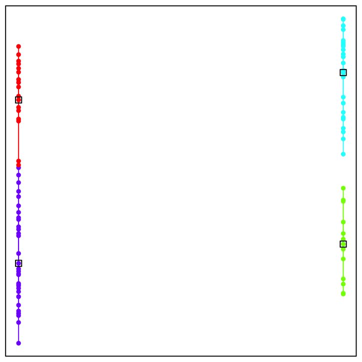

(c) Affinity Propagation clustering.

Figure 9: Signal Value vs. 5-Lag clustering.

4 Summary

4.1 Future work

Created application only scratches the surface of a vast field of live signal processing. In future study I would like to carry out throughout comparison of different clustering algorithm that can be used with proposed in this paper framework. Furthermore, presented here solution has not been tested with real-world signals therefore investigating the efficiency and accuracy of my clustering scheme in such scenario is crucial.

Produced application is fully functional but lacks graphics user interface to show status of incoming signal, behaviour of extracted features, and current cluster structure.
Moreover, directing clustering results back to `Esper` might be worth a closer look as it would facilitate live analysis of patterns emerging in the data. Such approach would allow to trigger actions based on cluster assignment of currently processed datum point.
Finally, converting my application to server based service managed via API could facilitate performing multiple independent analyses, and new rules deployment on the running application i.e. adding/removing/changing features of interest without stopping the signal processing.

4.2 Conclusions

This paper proposes an application of *Complex Event Processing* tool: `Esper`, to help make sense of data streams. It describes how to *generate signals samples*, easily *extract meaningful features* from them, and *cluster the data* to discover its underlying structure. Moreover, all presented here processing steps contribute to the goal of better understanding the behaviour of a signals.

Use of `Esper` brings all the advantages of efficient, multi-platform, and simple to use library, which can process at once thousands of events per second produced by multiple signals. Furthermore, it gives possibility to apply different filters, functions and time-windows in modular manner to extract relevant information. Last but not least, presented here *live stream clustering* model of processing, allows to use number of different clustering algorithms without significant alterations to the programme.

Finally, designed approach is capable of efficient live streams analysis and automatic signal status recognition. Moreover, presented in this paper results seem promising as in artificial setting (with use of signal generators) its performance was shown to be no worse than the one of static clustering.
With wide variety of application areas which demand more than simple signal thresholding e.g. system monitoring, presented here approach might be worth further investigation.

References

Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. ISBN 1935182218, 9781935182214.

Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107422221, 9781107422223.

Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, 315:972–976, 2007. URL `www.psi.toronto.edu/affinitypropagation`.

Floyd Marinescu. Esper: High volume event stream processing and correlation in java. Online article, July 2006. URL `http://www.infoq.com/news/Esper--ESP-CEP`.

Xiangliang Zhang, Cyril Furtlehner, Cécile Germain-Renaud, and Michele Sebag. Data stream clustering with affinity propagation. 2013.