An open architecture for complex event processing with machine learning

Nhan (Nathan) Tri Luong, Zoran Milosevic[1,2], Andrew Berry[1], Fethi Rabhi[3]
[1]Deontik, Australia
[2]Institute for Integrated and Intelligent Systems, Griffith University, Australia
[3]School of Computer Science and Engineering, University of New South Wales, Australia
{luongtrinhan@gmail.com, zoran@deontik.com, andyb@deontik.com, f.rabhi@unsw.edu.au}

Abstract — **This paper proposes an advanced, open architecture to augment streaming data platforms with both complex event processing (CEP) and predictive machine learning models. We leverage the power of CEP to preprocess streams using sophisticated event pattern expressions then present these preprocessed streams for downstream training and predictive computations. We demonstrate this approach using specific technology components.**

Keywords – streaming; real-time analytics; complex event processing; machine learning; artificial inteligence.

I. Introduction

The increasing availability of digitally accessible real-time data sources such as stock market data, news, social media and IoT systems, and the growing number of data streaming engines provide a basis for many new real-time analytics applications. These applications deliver real-time insights into changes occurring in business or social environments and enable rapid decision-making.

A data stream is a sequence of data items carrying information about events, each signifying an occurrence of an important change in the environment. There are two main algorithmic challenges when dealing with streaming: (1) the streams are large and have high velocity, and (2) we need to match patterns and make predictions incrementally in real-time. This implies that we often need to accept approximate solutions in order to use less time and memory.

It is possible to capture event relationships through *event pattern types*, specifying logical, temporal or causal relationships between events. Events can be emitted by multiple sources and it is often through combining these sources using sampling, filtering, correlation and aggregation that new insights can be delivered. For example, a sequence of events representing price movements can be correlated with news items coming from sources like social media to highlight influences on stock price.

The growing development and sophistication of machine learning (ML) and artificial intelligence (AI) provides data-driven inference and learning about activities in a particular domain. Considerable research in ML and AI has been centred around accessing, curating and pre-processing *static* data sets for model training purposes. For example, the use of ML and AI can help identify stocks whose historical trade prices move together in the same direction, known as *pairs trading*. This can be extended to help identify specific trading opportunities in real-time or select assets during portfolio construction. Processing of *continuously changing* data streams for ML purposes however brings many additional challenges [2].

Continuous learning is the ability of a model to learn continuously from a stream of data. This requires flexible architectures to support integration of pre-processing, training and prediction tasks. This paper provides an open architecture that supports complex pre-processing using CEP pattern matching capabilities and integrates it with the downstream ML and AI functions in a manner that supports real-time AI prediction and dynamic training.

The next section presents emerging trends and challenges in integrating streaming analytics with ML and AI, and presents a generic architecture in support of this. Section III describes our specific implementation approach, section IV provides evaluation of our approach and section V presents conclusions and future directions.

II. Integrating event-based streaming and AI

A. Related work

We address two distinct aspects of integrating event-based streaming and ML/AI solutions:

- Providing a flexible *architecture* to support multiple configurations and deployment options, and to accommodate different real-time analytics use cases.
- Developing *streaming algorithms* that support machine learning and prediction tasks on a real-time data stream.

The key related research efforts are summarised next.

1) Architecture aspects

A flexible architecture should accommodate a wide range of real-time analytics use cases. For example, some usage scenarios do not require sophisticated CEP functionality and can be implemented using out-of-the-box constructs from streaming engines, such as filtering and simple queries over events made possible by new real-time analytics engines. Examples are Microsoft Azure Stream Analytics [10], Amazon Streaming [11] with multiple open source streaming engines, including Kafka [3], Flume[6], Spark [7] and Storm[8], and Cloudera streaming analytics [12], based on Apache Flink [5]. These streaming services have evolved their ability to support processing of events, now including basic elements of some earlier complex event processing (CEP) technologies [1], as also discussed in our earlier analysis of real-time analytics solutions [14].

There are however only a small number of CEP engines able to support truly event-based and distributed processing with minimal latency and rich expressiveness of event relationships, and with integrated event and temporal relationships needed for various type of sliding window applications. At present, the best approach to supporting CEP functionality over a streaming engine is to provide separate CEP library components [12] to be used incrementally if the event-based processing requires more expressive event pattern semantics. This allows the use of sophisticated CEP solutions only when they are needed to add the CEP properties over streaming functionality, while allowing separate management of various properties dictated by the specific real-time analytics use cases.

Finally, adding ML capability to streaming platforms is a very recent development in response to new real-time predictive requirements. This can for example be demonstrated by the TensorFlow2 feature to support interfacing with the Kafka platform [23].

2) *Data streaming algorithms*

There are many challenges for the development of advanced algorithms that could support integration of streaming and CEP processing with ML tasks as discussed in [2]. These can be grouped in the following categories: *classification*, *regression*, *clustering*, and *frequent pattern mining*. In *classification* over continuously arriving stream of data, the problem is to assign a label from a set of nominal labels to each arriving item, as a function of the other features of the item. An example of classification is to label incoming email messages as spam or not spam. *Regression* is similar to classification, with the difference that the label to predict is a numeric value instead of a nominal one. An example of regression is predicting the value of a stock in the stock market tomorrow [2]. Classification and regression are supervised learning tasks because they need a set of properly labelled examples for training. When examples are not labelled however, one can use clustering algorithms to group them in homogeneous clusters, such as to group user profiles in a website. This is an example of an unsupervised learning task. Frequent pattern mining searches for the most relevant patterns within the examples, such as association between stocks bought together, e.g. banking and insurance stocks.

There is recent research effort for linking streaming and predictive analytics [13], through computing various kinds of forecasts based on a business process event log. The authors first introduce a static prediction workflow pattern, which separates the event log into one sub-stream for each process instance, and then compute a prediction based on the content of a sliding window of events. They further develop a predictive learning workflow for computing a function (i.e. a “learned” function) from previous instances of the process, which can then be used as a predictor for future instances. They do this by running an ML algorithm on a feature vector calculated over a sliding window of events. Finally, they develop a self-correlated prediction combining the previous two patterns, with the goal to compute a prediction on future events based on what has been learned on past events of the same log.

B. *Proposed architecture*

Our proposed architecture is based on the ideas of:

- Separating functionality associated with the CEP solutions from the emerging streaming solutions, which provide simpler event processing functionality; this allows for CEP functionality to be managed separately.
- Using pre-processed streaming data coming either from the streaming engine or from the CEP engine as input to the machine learning (training) capability.
- Allowing any prediction algorithm (predictor) to be linked with the machine learning (training) functions.
- Applying the prediction algorithms trained as above to the real-time, pre-processed data stream.

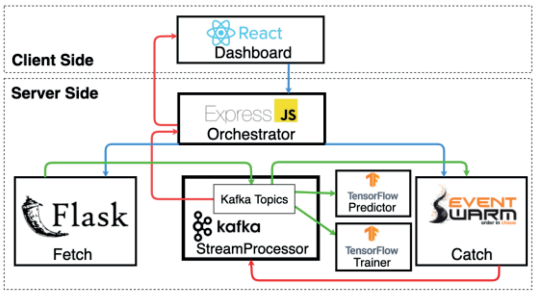


Figure 1: A specific instance of the architecture

This leads us to a general architecture which integrates streaming, CEP functionality and ML/AI with the following components (see Fig. 1):

- *Dashboard*, which allows a user-friendly interface for specification of event patterns, e.g. event sequence types, indicating increase or decrease of stock price and visualisation of relevant results, such as the extracted stock movement patterns.
- *Fetch* which delivers data (events) from external sources for processing.
- *Stream Processor*, for simple event processing, e.g. filtering, placing events of specific types on different event channels, possibly to be persisted via specific *messaging topics*, keeping persistent store of event occurrences, that can subsequently be processed for analytics purposes.
- *Catch*, which is an event pattern detection component that processes the events from the *Stream Processor* and matches patterns as per CEP rules. This component applies sophisticated CEP functionality over the events.
- *Trainer*, which is a ML training component intended to train models using datasets created incrementally on the *Stream Processor*. These datasets consist of either the data from the raw events received by the *Stream Processor* from *Fetch* or the event pattern instances matched by *Catch*. The machine learning model to which the dataset is supplied is determined by the user.
- *Predictor*, which uses the trained model produced by the *Trainer*, applying it to incoming raw events or detected event patterns to generate predictions. These predictions can be assessed by a separate monitoring component, and the output of these can be used to update training models to reflect the new statistical distribution of arriving data for example. This is part of the continuous learning approach (not shown in the diagram, but we plan to investigate this in the future).
- *Orchestrator*, which orchestrates the activities of the above components and manages the internal state of the application.

Continuous learning is thus supported by collecting the event pattern instances detected by the *Catch* component. These are then persisted in the messaging layer of the *Stream Processor* and can be used as training data sets for *Trainer* component. This component can perform retraining as dictated by the specifics of the application in question. For example, in the pairs trading scenario identified in section I, this can be done at the end of the trading day or in the extreme case, after each trade. We also separate the functionalities of the *Trainer* and *Predictor* to allow the model to be updated in the background by the *Trainer* while the *Predictor* continues without interruption.

III. Implementation

This general architecture can be used as a template for many end-to-end real-time analytics and ML scenarios. This section presents one specific implementation which leverages many years of our CEP research [15][16] and is in line with our earlier work in finance [9]. We experiment with detecting specific stock price patterns in real time, such as pairs trading scenarios, i.e. when stock prices from different companies consistently move in the same direction. This situation can help investors in making decisions about their stock portfolio. Examples of some typical stock price patterns are shown in Fig. 2 below.

Key elements of these patterns are gradients of stock price movement which can be up, down or flat. Each gradient line segment is defined by the stock prices carried by quote events. Two or more events in the same direction form an event sequence pattern (see section III.D). Fig. 3 depicts how two gradients can be combined.

The specific components of our implementation are:

A. Flask – realises Fetch

We use Flask [19] to integrate a web scraper which visits the Yahoo finance site and retrieves data every minute, encodes it in a JSON message and publishes it on a Kafka topic. Flask allows us to present the Fetch component as an API service that listens for user input from the Dashboard.

B. Express.js - realises Orchestrator

Express.js [20] allows developers to create server-side web applications quickly and efficiently. We chose Express.js because it integrates well with the Socket.io library [21] that we used in both the Dashboard (React.js) and the Orchestrator to enable bi-directional communications between the server and the client, which is vital to the experience of the users. When the user activates one of the CEP Display Tiles (see Fig. 4) the Orchestrator receives a request from the Dashboard then sends corresponding API requests to both Fetch and Catch to start the event pattern capture process. The Orchestrator will keep track of the overall state of the application and handles all the cleaning up when the user deletes a CEP Display Tile. It also facilitates the task of starting model training and deploying a trained model into the Predictor.

C. Kafka – realises Stream Processor

Kafka [3] was selected as a well-designed distributed streaming engine that presents a stream of events from the producers (Fetch and Catch). Kafka topics allow segregation of events of particular types so these events can be efficiently consumed by other components. Further, as all events are stored in a reliable manner. they can be queried by consumers that need historical data. This is very helpful for our Trainer component because it needs to train on past events.

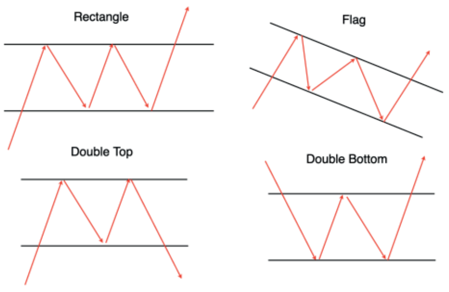


Figure 2: Common Stock Patterns

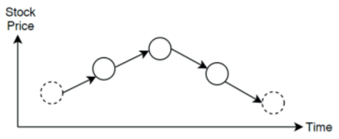


Figure 3: Gradient Event Sequence Visualization

We have two Kafka topics named Catch and Fetch which receive data from the Catch and the Fetch components. The Catch topic is consumed by both Dashboard (to provide real-time visual feedback to the users when a match happens) and the Trainer/Predictor (to train and use the ML models). The Fetch topic is consumed by both Catch (to capture event sequence matches from raw data) and Trainer/Predictor for the same reason as above.

D. EventSwarm – realises Catch

EventSwarm [4] is a lightweight, in-memory, open source CEP library supporting data-driven and low-latency processing, and designed for embedding in applications, even mobile applications. Key concepts include [4] [9]:

Event type – characterizes a set of events that share particular properties. Recall that an *event* captures information about some occurrences in the real or virtual world.

Event pattern – defines a relationship between events. Patterns are matched by feeding events through one or more processing graphs that select matching groups of events.

Triggers and Actions - event processing components are connected using triggers and actions. Upstream components offer triggers, downstream components have actions. Actions are registered against one or more upstream triggers defining the edges of the processing graph.

EventSet - the core aggregation construct that collects an arbitrary group of events in an order consistent with time. An EventSet is agnostic to the underlying event types: e.g. if the types are tweets, then a filter can be used to make sure the EventSet only contains tweets.

Window - a bounded EventSet is used to limit the scope of event pattern matches and control memory usage, based on time (e.g. a sliding time window) or number of events.

Expressions - describes event patterns to be matched, which can be either simple (single event) or complex (multiple events or event relationships). A *Complex Expression* is an expression that matches multiple events, e.g. a for example, a tweet followed by one or more retweets. There are two key ComplexExpression classes: ANDExpression, satisfied when it collects a set of events that match the component expressions, and a SequenceExpression, satisfied when it collects a set of events that match the component expressions and are strictly ordered in time (i.e. timestamp(a) < timestamp(b) < timestamp(c) …)

Abstractions perform calculations or compute derived values over an EventSet, e.g a StatisticsAbstraction that maintains statistics on numeric values extracted from individual events by a ValueRetriever. Most of the abstractions, including the StatisticsAbstraction, are updated incrementally as events flow through.

E. TensorFlow – realises Trainer and Predictor

TensorFlow was selected because it integrates well with Kafka data streams through the introduction of KafkaDataset library [23], allowing reading from and writing to Kafka topics. TensorFlow has a large number of pre-built libraries that help to build and train ML models effectively. In this implementation we used TensorFlow to build a Long Short Term Memory Network (LSTM) model that is capable of predicting stock prices based on the raw data coming in from the Kafka component. We also implemented a Dynamic Time Warping (DTW) algorithm that calculates the distance between two time series. This can be used to assist users to develop their own trading strategy as will be discussed in section III.F.

F. React.js – realises Dashboard

React.js is used because it renders the user interface efficiently, while encouraging reusability of components.

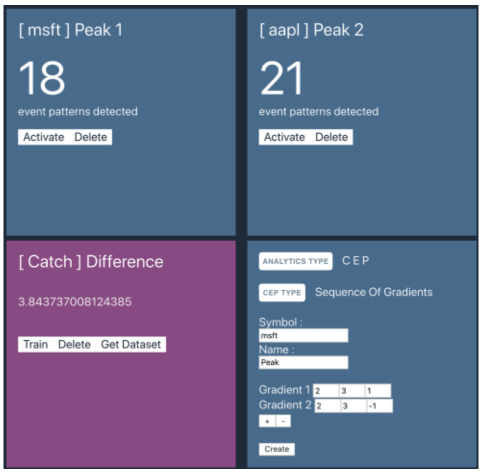
Fig. 4 shows the overall view of the Dashboard. It is a tile-based system with two main types of tiles:

1) The Define Tile

The Define Tile allows a user to choose "Analytics Type" to decide which Display Tile to create. When creating a Display Tile of type ML (Fig. 5), a user can specify the ML Type (supervised or unsupervised), a custom name for convenience, then choose the Data Source that the Training component can train on. At the time of writing, there are two main training data sources coming from either Fetch or Catch components, and this data is captured by the corresponding Kafka topics.

The Fetch source provides raw events from Yahoo finance data streams published on the corresponding Kafka topic (green line between Fetch and Kafka in Fig. 1). The Catch source publishes event patterns that are captured by the Catch component on the corresponding Kafka topic (red line between Catch and Kafka in Fig. 1). A user can select one of the ML Algorithms provided to experiment with the ideas. Currently, the platform supports Dynamic Time Warping (DTW) and Long Short-Term Memory Neural Network (LSTM). However, adding new algorithms is not difficult due to the expandability of the architecture.

When creating a Display Tile of type CEP, there are multiple CEP types which can be chosen using the "CEP Type" menu, such as the one shown in the bottom-right tile of Fig. 4, namely the "Sequence of Gradients" CEP type.


[msft] Peak 1

18

event patterns detected

[aapl] Peak 2

21

event patterns detected

[Catch] Difference

3.843737008124385

ANALYTICS TYPE CEP

CEP TYPE Sequence Of Gradients

Symbol : msft

Name : Peak

Gradient 1	2	3	1
Gradient 2	2	3	-1

Figure 4: The Dashboard

The Symbol parameter is needed to specify which stock data to fetch.

Note that Gradient is a special kind of EventSwarm event sequence which detects a continuously increasing, decreasing, or unchanged numerical sequence. This simple gradient pattern is sufficient to explore multiple stock price patterns no matter how complex they are (Fig. 2). The composition of gradients comprises a number of individual gradients that can be chained together using the sequence editor. These gradients are looking for the opening price of each stock event. All Gradients are concatenated back-to-back to each other starting from Gradient 1. Each Gradient has 3 main parameters: the minimum and maximum length of the Gradient, and its direction (+1 stands for going up, -1 for going down, and 0 for staying flat). Using Fig. 4 as an example, the values in the sequence editor correspond to the visualization graph in Fig. 3, where each event is depicted using a circle. Circles with the solid outline create a minimal sequence that can be matched by the specified parameters, while the maximal sequence includes both solid and dashed circles.

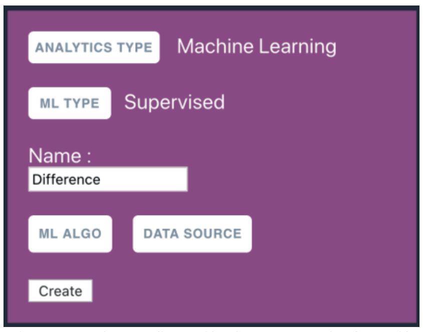
2) The Display Tile

The ML Display Tile (the bottom-left tile of Fig. 4) allows users to either start the training process or download the dataset in csv format. The Trainer pulls data from a Kafka topic based on the Data Source specified, then performs the training, after which the trained model is transferred to the Predictor for predictions on the new incoming data from the same Data Source. The result is shown directly on the tile. The example in Fig. 4 shows that the calculated DTW value is 3.84. Note that the smaller the distance, the more related these event patterns are, i.e. these two stocks tend to move in a similar way during this scenario. Two event patterns are identical when their distance is 0.

The CEP Display Tile (the top-left tile of Fig. 4) shows a counter for the number of matches detected. After a tile activation, raw stock data will flow from Fetch to Catch, and if a match occurs, Catch sends the matched data to the Front via Kafka then the counter ticks.

G. Docker environment - managing the components

With a large number of separate components, it is hard to manage all of their dependencies and to execute them in in a required order. In order to mitigate this problem, we packaged each component as a Docker image [22]. Docker is a good fit for a microservices architecture like ours because each image maps directly to a component, which allows faster software delivery cycles and expandability -


ANALYTICS TYPE Machine Learning

ML TYPE Supervised

Name : Difference

Figure 5: The Define Tile for a ML Display Tile

we can add a new component to the system just by building another Docker image. Managing a large number of docker components however is difficult, so we use Docker Compose, to define and execute the images in a specific order following our configuration. The application is available as open source [24].

IV. Evaluation

We used this architecture to experiment with two stock price analysis scenarios, described next.

A. Scenario I

This scenario did not involve CEP functionality, as the goal was to experiment with the connection of Fetch-StreamProcessor-Trainer-Predictor chain. The purpose was performing stock price prediction for a particular company and for the next trading day. The training algorithm accesses stock data obtained from Yahoo Finance via the Fetch component and persisted on a Kafka topic (JSON format). We considered a number of prediction algorithms and chose a Long Short-Term Memory (LSTM) Recurrent Neural Network in Python using TensorFlow 2 and Keras, based on the approach reported in [17]. We chose LSTM as this has become a prevailing algorithm for time-series prediction.

A user can create an ML Display Tile using the LSTM Algorithm, Fetch Source and click the "Train" button. The Trainer fetches all retrieved stock data from Kafka and starts the training phase. Once completed, it will show the prediction of the upcoming stock price based on the history it examined on the ML Display Tile. The user can re-train the model at any time to cover a wider range of data, which is delivered every minute from the Fetch component.

Investors can use this functionality to get the most informative indicators and make better predictions. They can use the platform to fetch data from multiple sources and companies, do ML training in parallel on all the data and get the result back using one unified platform.

B. Scenario II

Our second scenario addresses the pairs trading problem introduced in section 1, allowing the user to experiment with different parameters and assist them in detecting stock pairs that tend to move together without being distracted by all the market noise that makes the stock fluctuate.

In this case all components of architecture are used. The Catch effectively acts as a filter that captures meaningful stock patterns that the users are interested in. This filtering is extremely important since the ML algorithms tend to perform poorly on data that has a lot of noise. Specifically, we made use of ANDExpression and SequenceExpression described in III.D to pre-process raw data coming from Fetch before going to the Trainer. The combination of these two expressions allowed us to capture more meaningful information which enhances the effectiveness of the Trainer.

The user can create two different CEP Display Tiles that specify parameters to capture different stock patterns from different companies in real-time, e.g. "msft" (Microsoft) and "aapl" (Apple). All the detected event patterns are published in Kafka. Once the user is happy with the number of captured stock patterns, they can create an ML Display Tile that uses Catch as the data source and DTW as the ML Algorithm. The decision of when to start the ML Algorithm depends on the user rather than being fully automated. This is a good example of the combined actions of users and algorithms, often referred to as augmented intelligence, being used to assist humans with the decision making process. The output of this ML Display Tile is a number indicating the distance between the two series of event patterns captured by the previous CEP Display Tiles. This distance measure indicates how different these series are, with a distance of 0 indicating that the series are identical and the distance increasing as the series become less similar. A screenshot of this scenario is depicted in Fig. 4, in which the distance between Microsoft and Apple stocks is 3.84. This means that these two stock prices tended to peak together during the experiment.

This allows investors to experiment with a number of different event patterns to reflect investor interests, then use the system to calculate DTW distance between different pairs of stocks and determine whether these stocks are closely related (e.g. they move in a similar way) or not. The platform also allows the investors to download the training dataset by using the "Get Dataset" button on ML Display Tiles. This dataset is very valuable since it has been filtered through the Catch component and formatted in the popular csv format. This dataset is considerably more reliable for ML training than raw data because it only shows relevant stock patterns specified by the user. The resulting dataset contains less than a hundred records for each stock symbol per day compared to half a thousand raw records. Investors can manually analyse the dataset to make further decisions or they can import it into other systems for further investigation. The platform provides a large number of possible combinations of parameters that the investors can try until they obtain interesting results.

C. Implementation Remarks

One of the key observations in testing the system is the significant impact of when and how the retraining is done. As the data comes every minute from the Fetch, the trained model produced by the ML component changes every time the user clicks on the "train" button, and thus the dataset fetched into the ML component continuously grows larger over time. By default, the dataset includes everything retrieved by the Fetch from the beginning up to the current time. However, a larger dataset does not always improve the accuracy of the trained model because some older data might not be relevant to the predictions. Thus we need to identify an optimal way of determining how often the retraining should be done (e.g. every hour, or every day etc) and how much data should be included (e.g. one day, two days etc). The CEP platform can assist in this through detecting patterns to trigger retraining and constraining training datasets through sliding windows. We should also investigate whether continuous learning solutions are able to manage this problem algorithmically.

Another observation is that the ML component consumes significant CPU resources for training. Since the users are free to retrain the model as often as they want, doing so repeatedly can consume significant computational power and may affect the performance of the server cluster. This can be mitigated by limiting concurrent ML training processes based on the hardware configuration of the system or using cloud services for auto-scaling.

Each ML algorithm requires a different format for the input dataset and needs different types of attributes on each record of the training dataset. We had to limit each ML algorithm to perform training on a subset of all attributes available on the data. An additional component to transform the data being fetched to match input requirements of each ML algorithm would be helpful. The CEP component could be configured for this purpose, but an off-the-shelf tool optimised for such transformations is likely to be more efficient.

V. Conclusion and Future Work

The ultimate objective of this paper is to introduce a novel architecture that allows users to experiment with various combination of streaming, CEP and ML capabilities in real-time. This is achieved through providing a number of standard software components which can be combined according to the requirements of a specific use case, which speeds up the development cycle due to reduced dependencies and also improves the extensibility of the system.

We provided a specific realisation of the architecture using Kafka, TensorFlow 2, and EventSwarm. Our realisation provides users with a simple and straightforward interface that hides most of the complexity associated with the details of streaming, CEP and ML components. It allows analysts to focus on experimenting and gathering compelling results in their domain of expertise.

We have shown through our example scenarios that the inclusion of CEP for pre-processing and pattern matching extracts higher-level features from the raw data and provides more information to the ML algorithms to work on. This enables the trained model to provide new insights into the data.

This general architecture can be applied to support use cases in a wide range of domains, such as digital health, agriculture, climate change, and so on.

Our immediate goal is to further investigate the use of Kafka topics to support incremental learning. We also plan to support a larger variety of CEP rules and ML algorithms so that the users can have access to a more powerful platform for quick experimentation. During this process, we will increase the stability of the platform through better scaling and management of resources used in training processes. On a longer time horizon, we plan to automate some of the user choices related to ML technique selection and fine-tuning by integrating addition external components for data profiling, transformation and mining. This will enable more complex ML techniques (e.g. Deep Learning) to be deployed with EPS technology automatically applying complex data transformations as required by these techniques. This can be enhanced by offering the opportunity to perform continuous learning via automatic parameter-tuning based on analysing the results and user feedback.

References

[1] Real Time Intelligence & Complex Event Processing, https://complexevents.com (accessed 27 Apr 20)

[2] A. Bifet, R. Gavaldà, G. Holmes and B. Pfahringer Machine Learning for Data Streams, with Practical Examples in MOA, MIT Press, 2018.

[3] Apache Kafka, https://kafka.apache.org (accessed 27 Apr 20)

[4] EventSwarm, https://github.com/eventswarm (accessed 27 Apr 20)

[5] Apache Flink, https://flink.apache.org (accessed 27 Apr 20)

[6] Apache Flume, https://flume.apache.org (accessed 27 Apr 20)

[7] Apache Spark, https://spark.apache.org (accessed 27 Apr 20)

[8] Apache Storm, http://storm.apache.org (accessed 27 Apr 20)

[9] Z. Milosevic, W. Chen, A. Berry, F. A. Rabhi, An open architecture for event-based analytics, International Journal of Data Science and Analytics 2 (1-2), 13-27

[10] Azure Streaming Analytics, https://azure.microsoft.com/en-au/services/stream-analytics/, (accessed 27 Apr 20)

[11] Amazon Streaming, https://aws.amazon.com/streaming-data/, (accessed 27 Apr 20)

[12] Cloudera Streaming analytics, https://docs.cloudera.com/csa, (accessed 27 Apr 20)

[13] M. Roudjane, D. Rebaïne, R. Khoury and S. Hallé, "Predictive Analytics for Event Stream Processing," EDOC2019 Conference (EDOC), Paris, France, 2019, pp. 171-182.

[14] Z. Milosevic, W. Chen, A. Berry, F. A. Rabhi, *Real-Time Analytics*, in Big Data: Principles and Paradigms. Morgan Kaufmann/Elsevier, 2016.

[15] A. Berry, Z. Milosevic, Extending choreography with business contract constraints, IJCIS 14 (02n03), 131-179

[16] A. Berry, Z. Milosevic, Real-time analytics for legacy data streams in health: monitoring health data quality, EDOC2013 Conference, Vancouver, Canada, 2013.

[17] How to Predict Stock Prices in Python using TensorFlow 2 and Keras, https://www.thepythoncode.com/article/stock-price-prediction-in-python-using-tensorflow-2-and-keras (accessed 15 May 20)

[18] Yahoo Finance, https://finance.yahoo.com/ (accessed 01 May 20)

[19] Flask, https://flask.palletsprojects.com/en/1.1.x/ (accessed 01 May 20)

[20] Express.js, https://expressjs.com/ (accessed 01 May 20)

[21] Socket.io, https://socket.io/ (accessed 01 May 20)

[22] Docker, https://www.docker.com/ (accessed 01 May 20)

[23] TensorFlow Kafka Dataset docuemntation https://www.tensorflow.org/io/api_docs/python/tfio/kafka/KafkaDataset

[24] Stockswarm, https://gitlab.com/Treenhan/stockswarm (accessed 05 May 20)