Anomaly Detection over Streaming Data: Indy500 Case Study

Chathura Widanage[1] Jiayu Li[1] Sahil Tyagi[1] Ravi Teja[1] Bo Peng[2]
Supun Kamburugamuve[2] Jon Koskey[3] Dan Baum[4] Dayle M. Smith[4] Judy Qiu[2]
Department of Intelligent Systems Engineering
Indiana University
[1]{cdwidana, jl145, styagi, rbingi}@iu.edu
[2]{pengb, skamburu, xqiu}@indiana.edu
[3]{jkoskey}@indycar.com
[4]{dan.baum, dayle.m.smith}@intel.com

***Abstract*—Sports racing is attracting billions of audiences each year. It is powered and transformed by the latest data analysis technologies, from race car design, driving skill improvements to audience engagement on social media. However, most of the data processing are off-line and retrospective analysis. The emerging real-time data analysis from the Internet of Things (IoT) result in fast data streams generated from distributed sensors. Applying advanced Machine Learning/Artificial Intelligence over such data streams to discover new information, predict future insights and make control decision is a crucial process. In this paper, we start by articulating racing car big data characteristics and present time-critical anomaly detection of the racing cars with the real-time sensors of cars and the tracks from actual racing events. We build a scalable system infrastructure based on neuro-morphic Hierarchical Temporal Memory Algorithm (HTM) algorithm and Storm stream processing engine. By courtesy of historical Indy500 racing logs, evaluation experiments on this prototype system demonstrate good performance in terms of anomaly detection accuracy and service level objective (SLO) of latency for a real-world streaming application.**

***Index Terms*—big data, stream processing, anomaly detection, neuro-morphic computing, edge computing**

I. Introduction

The IndyCar Series, currently known as the NTT IndyCar Series under sponsorship, is the premier level of open-wheel racing in North America. Featuring racing at a combination of superspeedways, short ovals, road courses and temporary street circuits, the IndyCar Series offers its international lineup of drivers the most diverse challenges in motorsports. Indy500 is its premier event at Indianapolis Motor Speedway where the racing cars reach speeds up to 235 mph.

INDYCAR, the sanctioning body for the IndyCar Series, utilizes a Timing & Scoring application that monitors lap times of cars to the ten-thousandth of a second, the closest in motorsports. With the advent of smaller but powerful computational devices, the cars and race tracks come fitted with hundreds of sensors and actuators. The sensors in the cars record and transmit various metrics (speed, engine rpm, gear, steering direction, brake et al.) to the main server present on premises of the Indy 500 race track. These advanced information technology infrastructures support the racing management and the communication between the drivers and their teams. Each race generates a large volume of the telemetry and timing & scoring data, for example in the race of May 27, 2018, it contains 4,871,355 records with consecutive data arrival interval of 6 to 8 records per second for each car on average.

To build a system to support real-time data analysis, such as anomalies detection on the Indycar timing & scoring data is a challenging task. First, we must have a learning algorithm capable of capturing the drifting of data patterns in real-time. Static pre-trained neural network models are not capable of making correct decisions or inference on the continuously evolving data streams which have their patterns changing over time. The desirable algorithm should keep learning and detecting from the streaming data in an online fashion, i.e., without looking at data forward. Second, we must adhere to the time constraints of a real-time application with a reasonable execution latency. The IndyCar application needs a real-time response with latency below 100 milliseconds, in order to cope with the sensor data arrival rate of [80,90] milliseconds. As the learning algorithm keeps learning from the data stream which is resource intensive, dealing with multiple metrics across all racing cars requires a scalable distributed system.

One such avenue lies at the intersection of real-time stream processing and machine learning. We aim to address this problem here, developing an application tailored to the data and requirements of the Indy500 race. We leverage an online learning algorithm called Hierarchical Temporal Memory (HTM) [14], developed by Numenta and deploy it on Apache Storm. Our main contributions are summarized as follows:

- Propose a scalable system design that supports real-time stream processing.
- Implement a prototype system that achieves good performance in terms of detection accuracy and service of the objective of latency.
- Performance analysis on HTM Java package and its deployment in storm cluster.
- Annotate Indy500 dataset on anomalies with known events and evaluate the performance of detection.

II. Problem Statement

A. Anomaly Detection of Telemetry in auto racing

Telemetry in auto racing has improved the domain very much in the last decade [18] [21]. Broadcast sports such as motor racing have brought opportunities for spectators to monitor the performance of cars in real time. Mikhail Grachev says data is the winning force in motor racing and that telemetry data is very valuable to them. This allows the racing car team to analyze the existing data and identify the next move. Specifically, telemetry data allows the team to be synchronized with the car [15]. Not only can the sensor readings be used in basic electromechanical operations, but the data transmitted over the network can also be used to perform data mining to identify anomalies in the system, component malfunctions or statistics generation.

To better understand the requirements for Anomaly Detection over IndyCar streaming data, we need to explore the properties of the sensors data and how they are different from those of general big data. IndyCar data exhibits the following characteristics:

- Large-Scale Streaming Data: over 150 sensors per car of 33 cars are generating streams of data continuously.
- Heterogeneity: Various sensors data from different cars, the tracks, GPS, 36 video cameras and racing information such as weather and wind resulting in data heterogeneity.
- Time and space correlation: sensor devices are logging to a specific time-stamp for each of the data items.
- Noise data: Indy500 dataset may be subject to errors and noise during acquisition and transmission.

B. Hierarchical Temporal Memory Algorithm (HTM)

HTM is capable of detecting anomalies from data streams in real-time and performs well on the concept drift problems. Related works using HTM [6] [19] [25] [26] demonstrate that it excels many other state-of-the-art anomaly detection algorithms. We adopt HTM as the core anomaly detection algorithm in our system.

HTM imitates the process of sequential learning in the neocortex of the brain, which is involved in higher cognitive functions such as reasoning, conscious thoughts, language, and motor commands [4]–[6], [16]. HTM sequence memory models one layer of the cortex, which is organized into a set of columns of cells, or neurons, as shown in Fig. 1. Each neuron models the dendritic structure of neuron in the cortex. Sufficient activity from lateral dendrite will cause a neuron to enter an active state, and a cell activated by lateral connections prevents other cells in the same column to enter an active state, leading to sparse data representation in HTM. Sparse representations enables HTM to model sequences with long-term dependencies, as in Fig. 1c, the same input "C" in two sequences invokes different prediction of either D or Y depending on the context several steps ago.

The connections between the neurons are learned from input data continuously. The input, $\mathbf{x_t}$, is fed to an encoder and create a sparse binary vector representation $\mathbf{a}(\mathbf{x_t})$. Then,

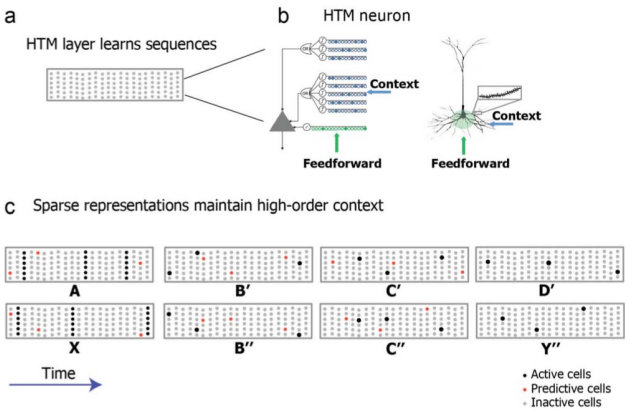

Fig. 1: Working of HTM sequence memory [6].

all neurons update their status by the inputs from connected neurons with active cells. It outputs predictions in the form of another sparse vector $\pi(\mathbf{x_t})$. The prediction error, S_t, is calculated by the number of bits common between the actual and predicted binary vectors, as

$$S_t = 1 - \frac{\pi(\mathbf{x_{t-1}}) \cdot \alpha(\mathbf{x_t})}{|\alpha(\mathbf{x_t})|} \tag{1}$$

where $|\alpha(\mathbf{x_t})|$ is the scalar norm, i.e. the total number of 1 bits in $\mathbf{a}(\mathbf{x_t})$. Furthermore, anomaly likelihood can be calculated from the prediction error by assuming it follows a normal distribution which is estimated in a previous window. As the likelihoods are very small numbers, a log transform is used to output the final anomaly score. For example, a likelihood of 0.00001 means we see this much predictability about one out of every 10,000 records, and the final anomaly score is 0.5.

C. Streaming Infrastructures

Successful big data processing systems, such as Hadoop and Spark were not built to process and take actions on continuous data streams flowing in at fluctuating rates. Such requirements and constraints for real-time processing led to the development of Distributed Stream Processing Systems [13] [17] like Apache Storm [24], Flink [10], Spark Streaming [27]. Spark Streaming is an extension to Spark as it uses a standard API to process incoming records as a set of mini-batches rather than process one tuple (or record) at a time. On the other hand, Storm and Flink follow a tuple-wise processing paradigm where we define the topology as a DAG (Directed Acyclic Graph) composed of parallel running tasks. Flink provides a unified API for batch and stream processing with pipelined data transfers. The message guarantee offered in Flink is exactly-once, while Storm offers at-least-once, exactly-once, and at-most-once guarantees.

As HTM is a sequential online learning algorithm, we will apply different metrics (e.g. SPEED, RPM and THROTTLE) in the same telemetry stream that can be processed by multiple HTM networks in pleasingly parallel. Given the application

requirements and topology design, we decided to proceed with Apache Storm as the stream processing engine.

III. System Architecture and Implementation

A. System Architecture

While anomaly detection is the core module that we focus on in this paper, the application needs a real-time response with latency below 100 ms, in order to cope with the arrival rate of [80,90]ms. It requires an end-to-end system as the testbed of streaming infrastructure. Fig. 2 shows the system architecture of five components. 1) We split IndyCar's TCP stream into two new streams at Event Publisher component and one goes directly to the database and other one is fed to the Message Queuing Telemetry Transport (MQTT) broker. 2) We use MQTT as the communication protocol within our infrastructure due to its high quality of service (QoS) [20] and lower bandwidth consumption. Apache Apollo is used as the message broker implementation due to its simplicity and performance. 3) Data processing or heavy lifting is done by a distributed HTM network which has been deployed over an Apache storm topology. Storm consumes topics from the message broker and feeds in real-time to the HTM network. The output from the HTM is published back to the message broker, which will be consumed by SocketServer and finally broadcast to the clients. HTM network is powered by a community managed Java implementation of the algorithm, HTM.java [1]. 4) We utilize a MongoDB database to persist all raw data and computed data in real-time for offline analysis. 5) We built a front end application to visualize the results of the processed data stream in real-time. The primary objective of this front end application is to make decision making easier and support drivers, pit-crew, engineers, and entertain remotely connected motor sports fans. We have made this application responsive, so it can be viewed in any modern web browser, including most of the mobile web browsers. Our system prototype online demo can be accessed at [2].

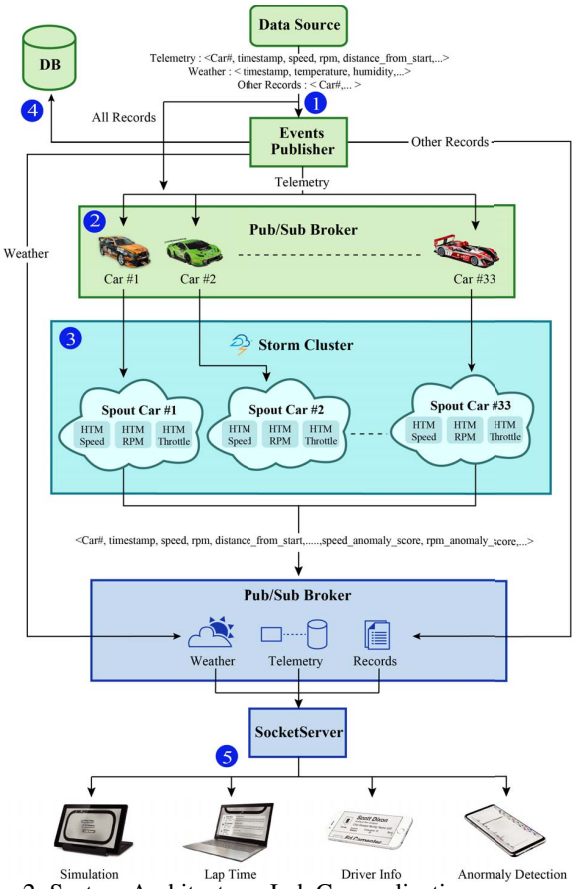

Fig. 2: System Architecture. IndyCar application processes the timing&score data streams of the race, detects anomalies and responses in real-time. Multiple types of clients are supported.

B. HTM Deployment in Storm Cluster

The central research problem we address in the system design is, ***How to deploy the HTM neural networks in a storm streaming cluster in order to achieve specific SLO of latency and scaling?*** HTM network provides good performance in detecting anomalies, and it needs relatively more computation resources when it keeps learning and inferring. The processing time for each incoming data record is not constant but depends on the context of the stream and the current learned model. In the Indy500 data streams, there are 33 cars and several telemetry metrics for each car. For example, when we use three metrics, SPEED (vehicle speed), RPM (engine speed) and THROTTLE, there are 99 HTM networks that should be deployed in the system with each network dealing with one metric. A trade-off between resource allocation and SLO of latency violation is the major factor in our system design.

First, the processing time of the HTM network should on average less than the application SLO requirement. We do extensive data analysis and performance evaluation on HTM in section IV.

Second, HTM.java provides an asynchronous interface for input and output. Internally, each network spawns a long-running thread. Thus, a thread level synchronization is still needed even when all the metrics of the same car are deployed on the same worker. This would introduce overhead and latency to the overall processing time.

Third, in order to reduce the unnecessary overhead of thread level synchronization and improve CPU utilization, we optimize the HTM.java library by changing the threading model. By default, HTM.java spawns a thread per layer in HTM network. Since anomaly detection is a one layer network, HTM.java builds one network for each metric and spawns one thread accordingly. In Fig. 3, three threads are spawned for three metrics for each car, and one instance of MQTT message client is created per car (per storm task), where it internally spawns four threads for, sending, receiving, pinging and for calling-back. With this default threading model, our setup spawns 8 threads per car including the Storm's threads. Hence if we schedule to process 33 cars within a single machine, it spawns a total of 264 threads (33 storm executor threads, 4*33

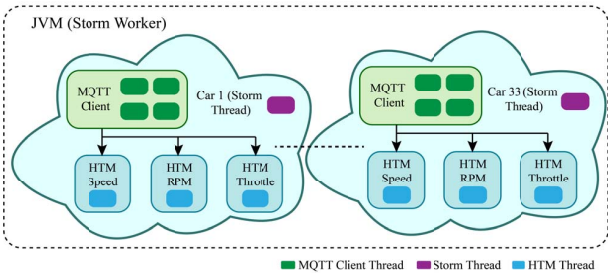

Fig. 3: HTM.java default threading model

message client threads, 3*33 HTM threads), which creates a significant resource contention issues.

By analyzing the thread utilization of each car, we identified that due to the arrival rate in the range of [80,90] ms, most of the threads remain in the waiting state. This adversely affects latency since, this behavior increases the amount of context switching and at the same time in-order to process a single event, three HTM threads need to be returned to the running state from waiting state. Since we need to combine outputs from all three HTM networks, before sending an event back to the broker, this drastically increases the latency for processing a single tuple.

As shown in Fig. 4, an improvement for this problem is to customize the threading model of the HTM.java library and handle multiple layers of multiple HTM networks by a group of long-running threads, instead of scheduling one thread per layer in 3. When a new HTM network is instantiated within the same Java virtual machine(JVM), we add the layers of that networks to a shared queue, which is visible globally across all the instances of HTM networks within that JVM. We also keep a globally visible counter which keeps the number of HTM networks instantiated within the JVM. Based on this count, we spawn threads on demand, to match one thread per three networks (configurable) rule. Each of these threads iteratively polls (takes the first in the queue) a layer from the queue and compute or process that layer and adds that back to the queue. If there is nothing to process in a particular layer, instead of waiting for data, the thread moves on to the next available layer in the queue. Along with the alterations of the HTM threading model, we configured storm tasks within the same JVM to share a single instance of MQTT client instead of creating an instance per task. We even replaced the default TCP connection factory of message client with our implementation to configure clients with TCP_NODELAY, in order to improve the latency of messages. These modifications reduced the threads per JVM significantly, and for 33 cars scheduled in the same machine, the total thread count was reduced down to 48 (4 MQTT client threads, 11 HTM processing threads, 33 storm executor threads). Intuitively, the best performance we can get is to dedicate one CPU core to one HTM network. In the assumption that the HTM network processing time is less than SLO, more compact deployment strategies are possible. We compare several deployment strategies in section IV.

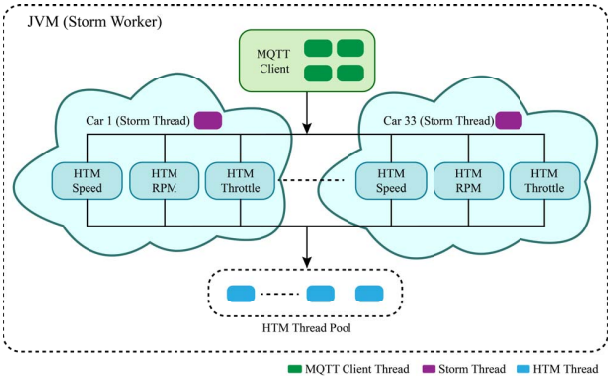

Fig. 4: HTM.java customized threading model

IV. Performance Evaluation and Case Study

In regards to hardware configuration, all experiments are conducted on a 10-node Intel Haswell cluster at Indiana University. Each node has two 24-core Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz processors and 128GB memory.

A. Indycar dataset

Results Protocol(RP) is being used in the INDYCAR timing system. RP definition provides telemetry and specific details for each session results, such as rankings with markers, team information, pit stop stats and many more. Table. I shows the record format for the telemetry records. We use RP log of the

TABLE I: Enhanced Results Protocol and Telemetry.

Fieldname	Data type	Comments
No.	Character	Car number 4 characters max
Time	Integer	Time of Day in ms
Position	Float	Metres since start of lap (1234.56)
Speed	Float	MPH ie. 123.456
Engine	Float	RPM ie. 12345
Throttle	Float	% throttle
Brake	Float	% brake
Steering	Float	-1.00 .. 0.00 .. 1.00
Gear	Integer	0 = Neutral, 1..6 = Gear 1 through 6

Indy500 final on the 27th of May 2018, which contained a total of 4,871,355 records (4,464,043 are telemetry records including warm-up rounds). The actual race has 2,373,400 records for 33 cars, that is over 75,000 telemetry records per each car on average, and these records span over 3.5 hours. During the racing, speed metrics varies a lot due to different types of events with Vehicle speed spans [0,238.95] miles/hour and Engine Speed spans [0,12920] RPM. Other metrics have smaller range, or can be discrete as Gear that falls into the range of [0,6]. We have observed variable data arrival intervals and missing or delayed events for all the cars. Fig. 5 shows the distribution of time gaps between two consecutive events (SPEED, ENGINE/RPM, THROTTLE) for all 33 cars.

B. System Latency

Analysis on the latency introduced by individual components of our architecture diagram, as denoted in Fig. 2, is the first step to enable our design deliver required SLO of latency reduction for the IndyCar real-time application.

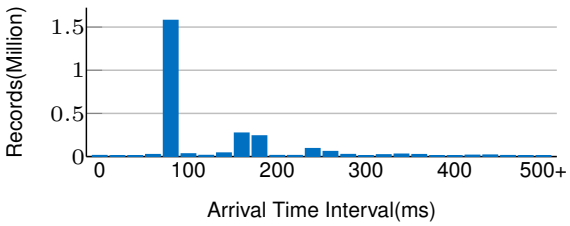

Fig. 5: Distribution of time gap between two consecutive events. Most of the records arrive in the range [80,90] ms, less than 0.05% of the records delay more than one second.

1) Network Latency: The Indycar application is based on a distributed streaming system having multiple components which are interconnected over the network. Some components utilize local area network(LAN) while some components connect to the system through the internet. Hence we carried out benchmarks covering both of these connection types.

TABLE II: IndyCar message Latency and Jitter evaluation

No. of Cars	Client-Server(ms)			LAN / PubSub(ms)		
	min	max	avg	min	max	avg
1	6	223	18.45	0.06	18.71	0.26
8	1	237	19.72	0.05	67.83	0.32
16	12	247	23.88	0.001	55.51	0.34
24	2	248	24.81	0.13	57.87	0.31
33	2	246	31.57	0.001	122.78	0.30

Intranet Latency. Since we have been initially using a file-based data source, messages from record reader (an application capable for reading log files and stream while keeping real timing between consecutive events) to WebSocket server goes through the LAN. We performed the evaluation for streaming events of 1, 8 , 16, 24 and 33 cars to analyze the variation of latency with respect to data volume. The "LAN/PubSub" latency in Table II shows that the average message flow latency within the LAN lies within microseconds range even for the maximum expected data volume for a race of 33 cars. Hence, the latency of LAN messages over message broker is negligible. **Internet Latency.** We implemented a web-sockets based web application to visualize the results of anomaly detection in real-time. As the columns of "Client-Server" in Table II, the latency of the message flow increase proportional to the volume of data. However, we identify that this variation is not only because of network latency, but also due to the single threaded nature of the web client application. In the real field setup, we'll be able to minimize the network overhead via data compression and multi-threading at client.

2) Latency from standalone HTM Module: HTM keeps learning and inference on incoming data streams. The execution time depends on the input and the current neural network status. We run experiments on the individual data stream and draw the distribution of the processing time which is the latency the HTM.Java module introduced. Fig. 6 shows the performance evaluation on SPEED, RPM and THROTTLE of Car20. 98% of the data is less than 20 milliseconds, and 99.9% of the data is less than 80 milliseconds. The average is less than 8 ms. If including the network latency and other

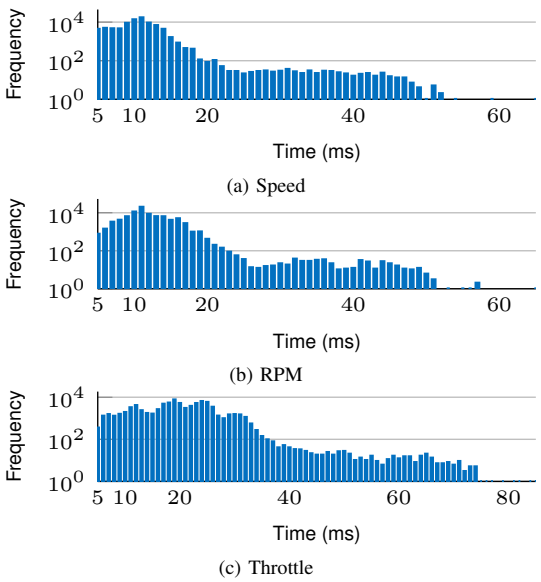

Fig. 6: Latency introduced by HTM Module for Car20 and three metrics separately and presented in log-scale.

overhead, deploying HTM networks for IndyCar data streams can potentially guarantee an SLO of latency. Fig. 9 shows the distribution of processing time of HTM.Java module, within 20 ms except for a few spikes occur at the beginning in which the HTM network is in the learning phase.

C. Deployment of HTM Networks in a Storm cluster

We run experiments to explore the candidates of deployment strategies - what is the SLO of latency can be achieved under a specific deployment of HTM in Storm? Two extreme cases include a). The best SLO if provide an unlimited resource. We allocate one CPU core for each HTM network. b). The SLO if given a tightly limited resource in which only one single node is allocated. The rightmost side of the x-axis of Fig. 7 represents latency of all 33 car data, corresponding to the OPT-N1, N2, N4 setup. The max values are 8329, 4373, and 1895 respectively, due to incidents such as Java Garbage collection.

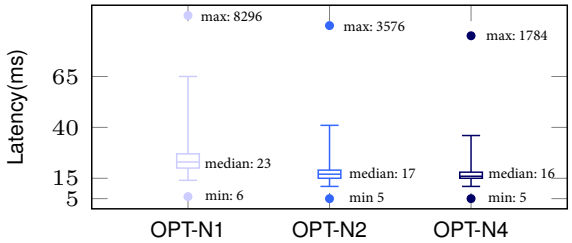

Fig. 7: Latency in the Storm+HTM phase. The lower & upper whisker represents the 2 & 98 percentile.

Fig. 8 shows the results of deployment for Indy500 with 33 cars and three metrics for each car for end-to-end solutions. Under OPT-N2, 99% percent of records can be processed less than 50 ms, 99.9% percent of records can be processed in less

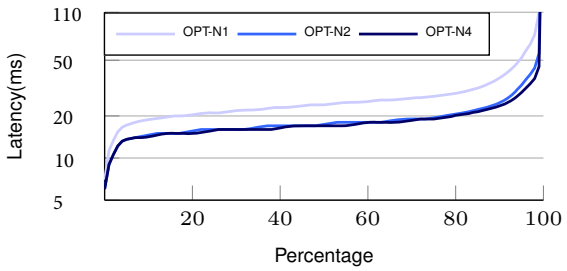

Fig. 8: Cumulative distribution function of latency illustrates Service Level Objective of Latency. Deploy strategy denoted as OPT-N# , where OPT is the HTM.Java module optimized version, # is number of worker nodes.

than 110 ms, with the average latency of 21 ms. The OPT threading models of HTM.java enable our proposed systems to process IndyCar data streams in real-time, at the data arrival rate in the range of [80,90] ms as shown in Fig. 5. Note that the latency increases with respect to the cumulative distribution function shift, especially when fewer resources are allocated. However, with the optimizations on HTM.java module, we can observe that the deployment with limited resource, OPT-N1 (on single node), still achieves decent SLO of latency at the time range of the average data arrival interval.

D. Anomaly Detection Evaluation

1) Anomaly Annotations: In order to evaluate the effectiveness of the anomaly detection system, the ground truth of the anomalies are needed. However, to make accurate annotations on millions of data instances is difficult if not impossible. Moreover, the purpose of our system is to help to discover novel anomalies that are previously unknown.

We adopt a partial annotation approach on the IndyCar dataset where only some known events are selected to be labeled as anomalies. We focus on the following three events. **1. Crash**: From the video replay of the race, there are 7 crashes in total found during the Indy500 racing event. Furthermore, by analysis on the flag information recorded in the timing and scoring logs, the time window when the crashes happened can be identified. **2. PitStop**:Like the event of crash, pit stops are events that can be identified by the video replay and timing and scoring logs. Usually, a pit stop lasts for 40 seconds of which SPEED will be zero for 14 seconds. There are 6 pit stops for each car in average. **3. GreenFlagUp**: When a crash happens, the race turns into a controlled mode to keep safe. All the cars follow the rules to slow down and wait for the signal of green flag. Then, the cars speed up back to the normal racing speed. All these three type events accompany large variations of the metrics of the racing cars, thus, many anomalies.

Table. III presents an example of the annotation results for Car-1. There are 6 pit-stops and 7 crashes during the whole race. We use an time window about 30 seconds centered with the event Time. The detection algorithm which detects as many as anomalies in the left side of the time window should be scored higher. Anomalies reported outside the time windows are either false positives or UNKNOWN events.

TABLE III: Annotation of Known Events for Car1

PitStop		Crash		GreenFlagUp	
ID	Time	ID	Time	ID	Time
1	16:45:40	car-33,30	16:56:03	1	16:23:00
2	17:01:01	car-10	17:11:26	2	17:09:06
3	17:26:43	car-13	17:23:07	3	17:19:14
4	17:51:36	car-18	18:17:52	4	17:31:03
5	18:15:48	car-3	18:29:26	5	18:28:27
6	18:57:40	car-24	18:40:55	6	18:39:22
		car-14	19:10:18	7	18:50:45

2) Accuracy of HTM algorithm: Fig. 10 demonstrates the anomaly detection result of HTM on IndyCar dataset. For the annotated three types of known events, it shows that detection on SPEED and RPM are accurate. Missing data at 16:50:15 and 17:27:00 can be observed in the figure but HTM shows stable performance when the missing data present. Some false negatives exist in the results for each metric, e.g., the GreenFlagUp event between 17:17 and 17:22 in RPM result. A fine-tuned anomaly threshold would deliver better accuracy and keep the balance between false positives and false negatives, which is one of our future work.

THROTTLE control is a critical technique to operate a racing car at its limits. While single metric with THROTTLE performs not as good as the other two metrics in the task of detecting anomalies of the known event types, it detects more subtle anomalies that the other two metrics failed to report. As in Fig. 10, UNKNOWN-G matches UNKNOWN-H while RPM fails, UNKOWN-C reports an area that both RPM and SPEED contains visible abnormal patterns. This evidence suggests that the other UNKOWN anomalies detected by THROTTLE might provide valuable information.

3) Case Study of Crash Event: Fig. 11 demonstrates the anomaly detection results on six metrics and is centered around a Crash event for car-13. Within the vertical red time window of the Crash event, all metrics can report a few anomalies. SPEED, RPM, and STEERING response mostly after the event, which indicates that the variance of the behavior of these metrics is the results of the crash. GEAR is not as sensitive as the other metrics. THROTTLE and BRAKE are interesting that some anomalies are detected at the left side of the center, before the collision. To verify these (unknown) anomalies as a means to provide a warning to severe events need further collaboration with domain experts on the IndyCar data.

V. Related Work

Anomaly detection in time-series is a heavily studied area of data science and machine learning [7], [11], but a vast majority of anomaly detection methods, both supervised (e.g. SVM and decision trees) and unsupervised (e.g. clustering), are for batch data processing. In practice, statistical techniques are used and they are computationally lightweight: sliding thresholds, outlier tests such as extreme studentized deviate (ESD or Grubbs) [22] and k-sigma, changepoint detection, statistical hypotheses testing, and exponential smoothing such

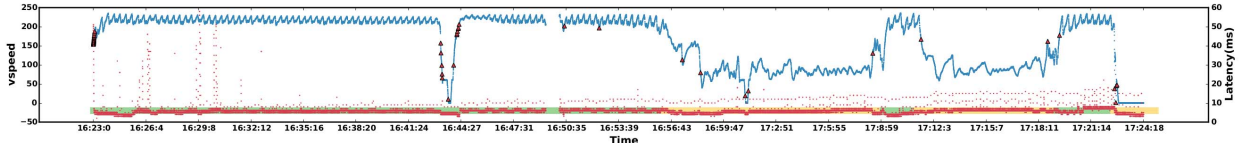

Fig. 9: Distribution of Anomaly Detection Processing Time. Car13 SPEED. Anomaly Likelihood threshold set to 0.5. Red dot scatter plot is the processing time. Red triangle are anomalies.

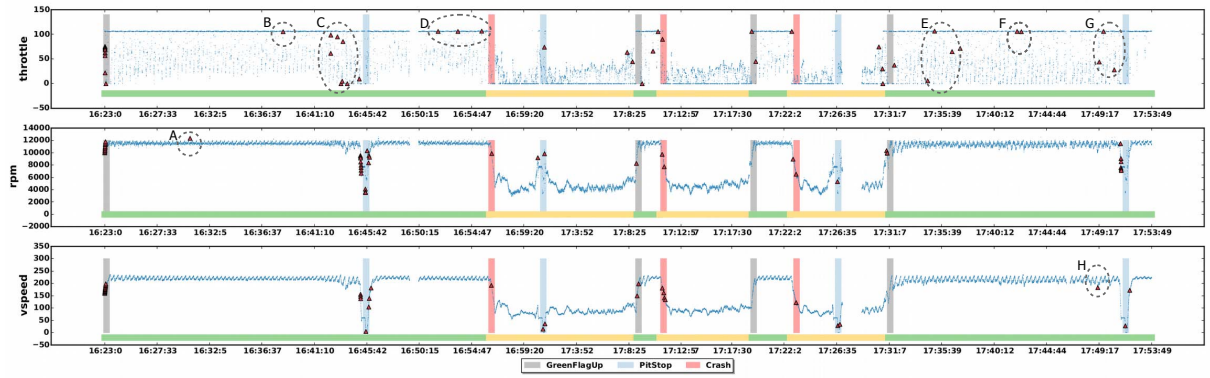

Fig. 10: Anomaly Detection Result for Car-1. Timeline for three metrics, SPEED, RPM and THROTTLE. The horizontal green/yellow bar represents the flag status during the race. Vertical bars represent the three anomalies types from annotations, including GreenFlagUp, PitStop, and Crash. System report anomalies are red triangles. Dotted circles indicate UNKNOWN anomalies. Only results from beginning to 17:54:00 are presented due to limit of space. Anomaly Likelihood threshold set to 0.5

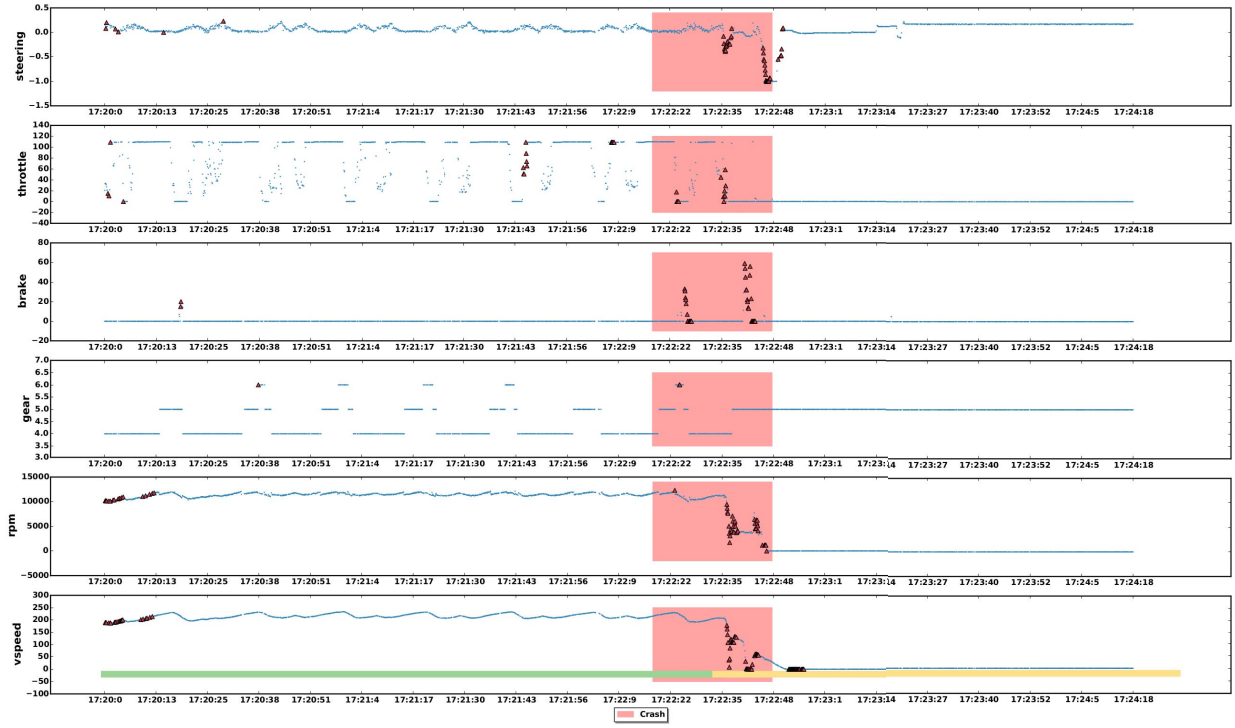

Fig. 11: Anomaly Detection Result for Car-13 on Crash Event. Six metrics, SPEED, RPM, THROTTLE, BRAKE, GEAR and STEERING are included. Centered vertical red bar is the crash event of Car-13. Horizontal green/yellow bar represents the flag status during the race. Anomaly Likelihood threshold set to 0.2.

as Holt-Winters [12]. Most of these techniques focus on spatial anomalies, limiting their usefulness in applications with temporal dependencies.

Algorithms for real-time anomaly detection include HTM, Skyline, Twitter ADVec, KNN CAD, Relative Entropy, Windowed Gaussian. NAB benchmark shows that HTM is one of the state-of-the-art algorithms that provide stable and high accuracy. We also tested three other real-time anomaly detection algorithms, namely online Bayesian Changepoint [3], Random Cut Forest, and EXPected Similarity Estimation [23]. Both HTM and EXPoSE have achieved good discover rate. Collective anomaly detection is another important area in which a collection of related data instances is anomalous concerning the entire dataset [28] [9]. Multimodal anomaly detection detects from multiple sources of data stream [8]. In this paper, we focus on contextual anomaly detection and report correlations among multiple metrics. Investigating collective anomaly detection with multi metrics will be our future work.

VI. Conclusions

Real-time anomaly detection on Indy500 racing event provides an interesting and challenging problem on machine learning algorithms and distributed systems. The heterogeneous streams with high velocity puts stringent time constraints on the processing time and require a scalable system for HTM neural networks on anomaly detection. We investigate the SLO requirements and reduce the latency for streaming data analysis. We show under different deployment strategies, our proposed distributed system is capable to run complex anomaly detection algorithm in real-time. The validation shows that HTM provides stable performance and is promising in detecting anomalies over high speed streaming data. We will collaborate with domain experts to leverage the experiences and findings in this work and make it a useful tool for anomaly detection in automotive applications. We made our source code available online at https://github.com/DSC-SPIDAL/IndyCar.

Acknowledgment

We gratefully acknowledge support from the Intel Parallel Computing Center (IPCC) grant, NSF CIF-DIBBS 143054, EEC 1720625 and IIS 1838083 Grants. We appreciate the support from IU PHI, FutureSystems team and ISE Modelling and Simulation Lab.

References

[1] Hierarchical Temporal Memory implementation in Java. https://github.com/numenta/htm.java/. [Online; accessed 1-Mar-2019].

[2] IndyCar Demo. http://indycar.demo.2.s3-website-us-east-1.amazonaws.com/. [Online; accessed 15-Apr-2019].

[3] R. P. Adams and D. J. MacKay. Bayesian online changepoint detection. *arXiv preprint arXiv:0710.3742*, 2007.

[4] S. Ahmad and J. Hawkins. Properties of sparse distributed representations and their application to hierarchical temporal memory. *arXiv preprint arXiv:1503.07469*, 2015.

[5] S. Ahmad and J. Hawkins. How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites. *arXiv preprint arXiv:1601.00720*, 2016.

[6] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262:134–147, Nov. 2017.

[7] R. A. Ariyaluran Habeeb, F. Nasaruddin, A. Gani, I. A. Targio Hashem, E. Ahmed, and M. Imran. Real-time big data processing for anomaly detection: A Survey. *International Journal of Information Management*, Sept. 2018.

[8] T. Banerjee, G. Whipps, P. Gurram, and V. Tarokh. Sequential event detection using multimodal data in nonstationary environments. In *2018 21st International Conference on Information Fusion (FUSION)*, pages 1940–1947. IEEE, 2018.

[9] L. Bontemps, J. McDermott, and N.-A. Le-Khac. Collective anomaly detection based on long short-term memory recurrent neural networks. In *International Conference on Future Data and Security Engineering*, pages 141–152. Springer, 2016.

[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[11] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[12] C. Chatfield. The Holt-winters forecasting procedure. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 27(3):264–279, 1978.

[13] X. Gao, E. Ferrara, and J. Qiu. Parallel clustering of high-dimensional social media data streams. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 323–332. IEEE, 2015.

[14] D. George and J. Hawkins. A hierarchical Bayesian model of invariant pattern recognition in the visual cortex. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 3, pages 1812–1817. IEEE, 2005.

[15] Guennadi Moukine. Mikhail Grachev: data Is the winning force in motor racing. https://motorsport.acronis.com/articles/en/mikhail-grachev-data-winning-force-motor-racing. [Online; accessed 1-Mar-2019].

[16] J. Hawkins and S. Ahmad. Why neurons have thousands of synapses, a theory of sequence memory in neocortex. *Frontiers in neural circuits*, 10:23, 2016.

[17] S. Kamburugamuve and G. Fox. Survey of distributed stream processing.

[18] Y. Kataoka and D. Junkins. Mining Muscle Use Data for Fatigue Reduction in IndyCar. Mar. 2017.

[19] A. Lavin and S. Ahmad. Evaluating real-time anomaly detection algorithms–the numenta anomaly benchmark. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 38–44. IEEE, 2015.

[20] S. Lee, H. Kim, D.-k. Hong, and H. Ju. Correlation analysis of mqtt loss and delay according to qos level. 2013.

[21] Lynnette Reese. Telemetry in Auto Racing. https://www.mouser.com/applications/automotive-racing-telemetry/. [Online; accessed 1-Mar-2019].

[22] B. Rosner. Percentage points for a generalized ESD many-outlier procedure. *Technometrics*, 25(2):165–172, 1983.

[23] M. Schneider, W. Ertel, and F. Ramos. Expected similarity estimation for large-scale batch and streaming anomaly detection. *Machine Learning*, 105(3):305–333, 2016.

[24] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM. event-place: Snowbird, Utah, USA.

[25] A. Vivmond. Utilizing the HTM algorithms for weather forecasting and anomaly detection. Master's thesis, The University of Bergen, 2016.

[26] C. Wang, Z. Zhao, L. Gong, L. Zhu, Z. Liu, and X. Cheng. A Distributed Anomaly Detection System for In-Vehicle Network Using HTM. *IEEE ACCESS*, 6:9091–9098, 2018.

[27] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM. event-place: Farminton, Pennsylvania.

[28] Y. Zheng, H. Zhang, and Y. Yu. Detecting collective anomalies from multiple spatio-temporal datasets across different domains. In *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*, page 2. ACM, 2015.