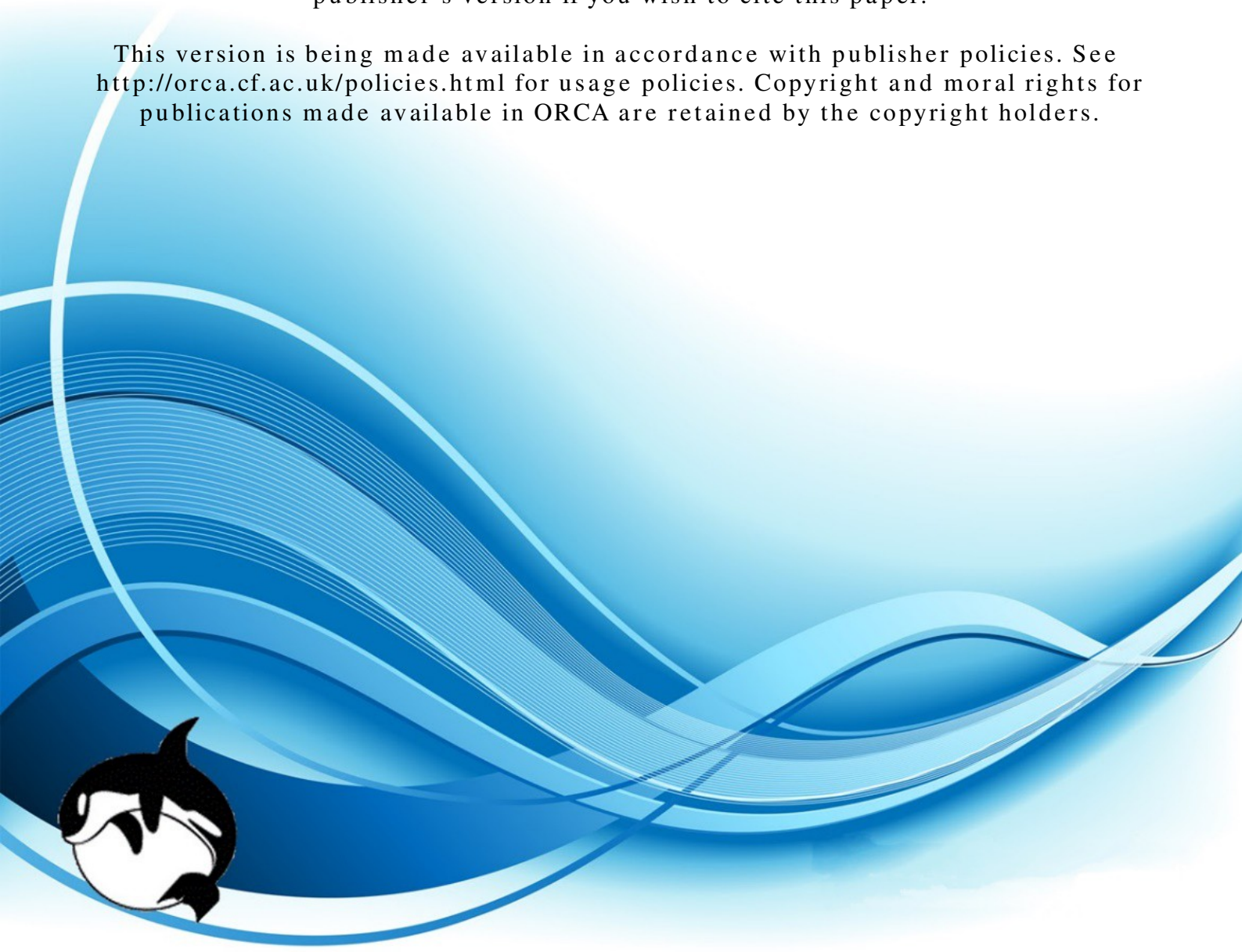

Resource-Aware Edge-Based Stream Analytics

Ioan Petri, Ioan Chirila, Heitor Murilo Gomes, Albert Bifet, Omer F. Rana, *Member, IEEE*

Understanding how machine learning algorithms can be used for stream processing on edge devices remains an important challenge. Such ML algorithms can be represented as *operators* and dynamically adapted based on the resources on which they are hosted. Deploying machine learning algorithms on edge resources often focuses on carrying out inference on the edge, whilst learning and model development takes place on a cloud data center. We describe TinyMOA, a modified version of the open-source Massive Online Analytics (MOA) library for stream processing, that can be deployed across both local and remote edge resources using the Parsl and Kafka systems. Using an experimental testbed, we demonstrate how machine learning stream processing operators can be configured based on the resource on which they are hosted, and discuss subsequent implications for edge-based stream processing systems.

***Index Terms*—edge computing, MOA, sensor data processing, stream processing, machine learning.**

I. INTRODUCTION

AN INCREASE in performance and reliability of edge devices provides computational capacity in proximity to an end user and to the data capture / generation source. Applications that can be hosted directly on these devices has also continued to increase over recent years, ranging from real time event processing to the identification of triggers that can initiate cloud-based execution, service and data migration to/from mobile devices and cloud systems, leading to more complex data processing of stream-based (e.g. audio and video) data. IoT applications can also generate vast amounts of data, often streamed from a device to a data processing system. The ability to run machine learning on resource-constrained IoT devices provides an alternative to connecting to the cloud. Data movement from edge to the cloud brings cost inefficiencies including latency, connectivity and security. The ability to host machine learning algorithms on constrained hardware can enable developers to change the way applications interact with computational infrastructure [1]. Given the resource constraints of IoT devices, their capability needs to be enhanced using edge computing to ensure efficiency, accuracy, productivity and cost reductions [2].

We analyse how machine learning (ML) algorithms can be applied to data streams on edge devices. We note that ML libraries can vary in computational complexity, based on a variety of hyperparameters that influence their construction and subsequent deployment. The complexity of an ML algorithm is also determined by the mode of usage, e.g. whether the algorithm is being used in "inference" or "learning" mode. Existing ML libraries, such as TensorFlow-Lite (TF-Lite) often involve a data center-based model development, and then an edge-based inference. This choice is made based on the computational and data storage capacity of edge devices compared to data center capacity (the former is expected to have substantially lower capacity than the latter). This discrepancy between edge and data center resources is therefore used to modify the types of operations carried out on an edge device compared to a data center. An ML model undergoes "adaptation" to make it execute on an edge resource. In TensorFlow to TF-Light, the number of bits used to encode weight parameters of a learned model are reduced (quantization) and the range of weight values are constrained (distillation) before moving the model to a resource constrained device. Our approach is focused on modifying the size and range of Java classes that can be hosted on a resource, realised using a custom Java class loader as explained in section II-B. This forms the key focus of this paper, i.e. how the resource characteristics and properties of a computational resource can dynamically modify the ML algorithm executed on a resource. We investigate, in particular, how ML applied to data streams can be adapted based on the resources on which ML algorithms are hosted, using the MOA (Massive On-line Analytics) software toolkit. Using stream processing terminology. We refer to ML algorithms used for analysing the properties of a stream as "operators", capable of being placed on edge or data center resources. In particular, we describe the development of TinyMOA, a version of MOA that can be hosted on edge resources [15]. MOA includes a number of machine learning algorithms, including classification (e.g. Hoeffding Trees, Naive Bayes, Stochastic Gradient Descent), clustering (e.g. StreamKM++, CluStream/micro-clusters), outlier detection, concept drift detection and tools for evaluation. The rest of this paper is organised as follows: Sections II compares our approach with other related efforts, highlightly key novelty of this work. An application scenario to support machine-aware edge orchestration is presented in section III followed by results in Section IV. We conclude and identify future research directions in Sections VI.

Ioan Petri, Ioan Chirila and Omer Rana are with Cardiff University, UK
Heitor Murilo Gomes and Albert Bifet are with the University of Waikato, NZ

II. RELATED WORK

A. Background

Edge computing builds on the growth in IoT infrastructures where heterogeneous and networked devices collaborate to achieve particular data monitoring/ processing objectives. *Fog cells* have been used to group IoT devices based on vicinity, i.e., single IoT device coordinating a group of other IoT devices, to support security, performance and data analysis. Such fog cells can lead to the development of IoT services to process data in close vicinity to data sources/ sinks as an alternative to transmission of data to a cloud system. The use of fog (edge)

cells can reduce communication delays, to provide a more efficient use of computational, storage and networking resources. Application scenarios include supporting pre-processing of data streams from sensor nodes [3]. Edge computing must also take into consideration user mobility, geo-distribution, latency and variable connectivity within a communication network [4]. In edge processing, quality of service metrics such as minimising network latency are supported through the use of cloudlets – which can bring edge capacity closer to IoT and mobile devices. Complementary to edge computing, "Cloud of Things" [5] provides *in-network* capacity between IoT and Cloud Computing for data processing. Similarly, operations on data (such as sampling or filtering), as it is being transmitted from an IoT device to a Cloud data center, can be carried out using in-transit processing, thereby reducing data volumes and limiting the size of the data being moved across the network [6]. This is equivalent to executing a number of stream operations on data along its path from a source to a destination.

Specialist services can also be provided on IoT/sensor devices based on their operating environment, such as provision of specialist operating systems (e.g. Contiki, Wind River VxWorks etc) which enable dynamic modification of code running on sensor nodes within a network. LooCI (LOOsely coupled Component Interfaces) supports the integration of software components across different sensor nodes. Edge nodes can also be viewed as hosting environments for code, the functionality of such nodes can be modified dynamically. Examples of sensor-hosted, lightweight Virtual Machine environments include DAViM (Dynamically Adaptable Virtual Machine) [7], and Mate [8] – a Java byte-code interpreter that executes over Contiki. Mate breaks an application into *capsules* that can be distributed throughout the network at runtime. Additional examples include Parsl & funcX, which enable (serverless) functions to be executed across different types of resources. The hosting environment can be modified depending on the resource being considered using a specialist *Executor*.

R-Pulsar is a software library for IoT systems that extends the capabilities of a cloud environment, by aggregating resources from local devices using a programming model for data collection and processing. R-Pulsar has been tested on various embedded devices such as Raspberry Pis (RPis) and Android phones to enable timely stream analytics by exploiting edge and data center resources. R-Pulsar can also perform and orchestrate data analytics between the edge and the cloud for computing continuum applications [9]. R-MStorm is another stream processing system that allocates tasks to edge devices within *stream groups* – to partition the stream among counterpart tasks. The system implements stream selection to support approximate computation, e.g. filtering of stream units to reduce processing workload by optimising the processing accuracy [10]. Microsoft-AI [11] supports stream processing applications for Android devices. The platform supports a wide range of AI scenarios configured to support application-specific model execution. The AI platform provides features for optimising performance and accuracy when using machine learning models with edge devices.

IoT data streams require processing on real time data which often require machine learning operations (i.e. inference) to be executed at the edge [12]. When sensor readings are trasferred from IoT devices to the Cloud, a reduction in data volume is needed. Dynamic clustering using centroids and fuzzy join on data streams can address data motion and online changes in data streams [13]. The volatility in data streams with data drifts are some of the challenges that impact on the performances of machine learning. The execution of machine learning at the edge [14] imposes several constraints on the learning algorithms, which is expected to yield accurate prediction while updating its model in an online fashion without surpassing strict memory and processing time constraints. Other challenges related to learning from streaming data concern the occurrence of concept drifts in the data, the lack of labelled data amongst others [15]. An *ideal* machine learning algorithm developed for streaming data must be resource-aware and able to maintain an accurate model based on the data on which it has been trained. The algorithm should monitor its computational resources usage and limit them appropriately.

B. MOA & TinyMOA Software Library

MOA is a software environment for implementing machine learning algorithms for processing streaming data. MOA contains a number of pre-built algorithms that can be configured by a user, ranging from decision trees (such as Hoeffding Trees) to dynamic clustering algorithms. A number of stream generators are also included to enable experiments to be carried out on synthetically generated data (such as a rotating hyperplane). MOA also provides a number of evaluation metrics (such as the Kappa statistic, accuracy, precision, etc) as a way to evaluate the performance of a learning algorithm and provide a basis for comparison between algorithms. MOA is popular among researchers as it is simple to configure and run experiments, but it can also be used in practical applications to process incoming instances using Kafka Streams. Kafka[1] is an Apache project that enables development of applications that make use of topic-based event publishers and subscribers. Kafka streams[2] enable dynamic partitioning of incoming data streams based on the resources on which these streams must be processed, providing both scalability and fault tolerance.

TinyMOA is a variant of MOA that can be hosted on edge-based devices (e.g. a RPi). TinyMOA fragments the MOA library into sub-components, enabling distribution of key MOA components to edge devices. Using custom classloaders, TinyMOA enables dynamic adaptation of stream processing algorithms based on the properties of the resources on which these algorithms are hosted – achieved in practice by limiting the number of Java classes that are hosted on the resource. TinyMOA only includes classifiers used in the specific experiment of interest, as indicated in section IV, and the core classes required for these classifiers to execute – such as the Instance representation to support data stream processing. The file size of MOA is 2.8MB while TinyMOA is only 970KB.

[1] https://kafka.apache.org/

[2] https://kafka.apache.org/documentation/streams/

Figure 1 illustrates the class loading mechanism in TinyMOA, where a limited number of MOA classes are uploaded to a RPi device, depending on the type of stream processing algorithm being considered.

We consider different edge, fog and cloud resources to host ML algorithms (hereby referred to as "operators"), combing the use of the MOA toolkit for real-time stream processing with Parsl [16]. This enables us to dynamically deploy MOA algorithms on the available computational resources. Parsl allows a variety of different types of resources to be integrated into the hosting environment (through the use of Parsl executors), such as high performance computing resources and edge devices (consisting of RPi). Each resource is treated as a Parsl endpoint, able to support a *Parsl Executor* that can execute an algorithm from MOA. We benchmark different machine learning algorithms (as stream processing operators) and measure their execution performance on different edge configurations. Configuration of the ML algorithm is modified based on the type of resource on which it will be hosted.

III. Methodology & Deployment

In this section we provide details about the edge experimental testbed and subsequent integration with MOA. Figure 2 identifies experiments and the associated data flow which changes at different stage in the process. We have implemented a Python script decorator, scheduled as an asynchronous task using the DataFlowKernel in Parsl. This enables a *future* object to be identified, i.e. placeholder data structure that is associated with an outcome that will occur once computation has completed. We also configure components of the DataFlowKernel such as the `Task Table` keeping account of scheduled tasks, `Dependency Check` which verifies Parsl has fulfilled all the data dependencies before execution, `Memoization Lookup` to prevent the same App from being executed many times with the same input parameters. An `Executor Selection` can be used to specify the type of task executor we want to deploy on the desired machine (e.g. a Raspberry Pi or an HPC resource). We make use of the `HighThroughputExecutor` in all our scenarios.

We use the Kafka software library on the same device as the Parsl-Executor to support stream management. Kafka is used to generate data streams for edge processing with improved latency and cost-efficiency. In our deployment, a Kafka server acts as a data streaming engine for the MOA classification algorithms, comprising of a zookeeper client and a server. Using the Zookeeper pub/sub. framework, we support a number of "Brokers" which hold a "Topic" that a listener can subscribe to. In our case the Topic contains the "Census" training data[3], which has been added via a Producer push request and which can be consumed by the MOA machine learning algorithm. The Census.arff file is 40MB in size, containing 68 numeric attributes, and can be streamed from a data producer to a consumer running the MOA algorithm (stream processing operator). This provides a useful benchmark data set that can be used to compare the performance of stream processing on a RPi vs. a desktop/cloud-based processing platform.

We have configured an experimental testbed that supports deployment and testing of several scenarios, across two main environments: **Configuration 1:** Local environment (Workstation) : ThinkPad T410i with 8GB RAM, Linux Ubuntu 18.04 Virtual Environment1 (Python3.7, Parsl1.1.0, MOA Release 2021.07) Virtual Environment2 (Python3.6, Parsl0.9.0, MOA Release 2021.07). **Configuration 2:** Remote environment (Raspberry Pi): RPi 4 model B with 2GB RAM - Raspbian OS/Linux 10 Virtual Environment1 (Python3.7, Parsl1.1.0, MOA Release 2021.07) Virtual Environment2 (Python3.6, Parsl0.9.0, MOA Release 2021.07). Hence, we implement and test MOA algorithms and their deployment on (i) a local resource, and (ii) remote Raspberry Pi (RPi) environment. These experiments demonstrate the behaviour of MOA machine learning on edge devices deployed on their own, locally or remotely through Parsl. In addition, we compare a simplified stream processing pipeline using TinyMOA. We therefore investigate reducing the size of the deployed software library (TinyMOA vs. MOA) on the execution time of the analysis tasks. For conducting these experiments we have defined a scenario that will be run on both systems locally and remotely.

The choice of tools and architectural design of our system allows for scalable remote deployment of stream processing on heterogeneous edge devices which can be scaled to take advantage of cheaper computational resources. This is achieved by fragmenting MOA into portable Java classes, that can be combined with custom classloaders. This coupled with Parsl enables an adaptive framework that can be supported across a number of different types of computational resources.

IV. Evaluation & Results

This section reports findings for *what-if* scenarios that use datastreams with machine learning algorithms and their subsequent execution on customized edge environments. The reported experiments utilise time-to-complete metrics for measuring the efficiency of the ML algorithms on local vs. remote resources (*local* resources have greater computational capacity than *remote* resources). The experiements aim to benchmark different machine learning algorithms and measure their execution performance on different edge configurations.

a) **Experiment 1: Hoeffding tree and KMeans:** In this experiment we measure the execution time of Hoeffding tree [17] and KMeans [18] implementation in MOA. As indicated in figure 3, the number of examples (on y-axis) to train the learner rang from 5K to 10M. The MOA stream generator was `generators.WaveformGenerator` which generates a stream for predicting one of three waveform types. As illustrated in Figure 3, we report the execution time on a local systems with spec. Intel Core i5-5200U, 8RAM compared to execution time on a RPi4 model B with 2GB RAM. As expected the local system with the better computational specs outperforms the edge device. Also Kmeans requires more computational resources and more time to complete.

[3]Available from: https://moa.cms.waikato.ac.nz/kdd-2017-hands-on-tutorial/

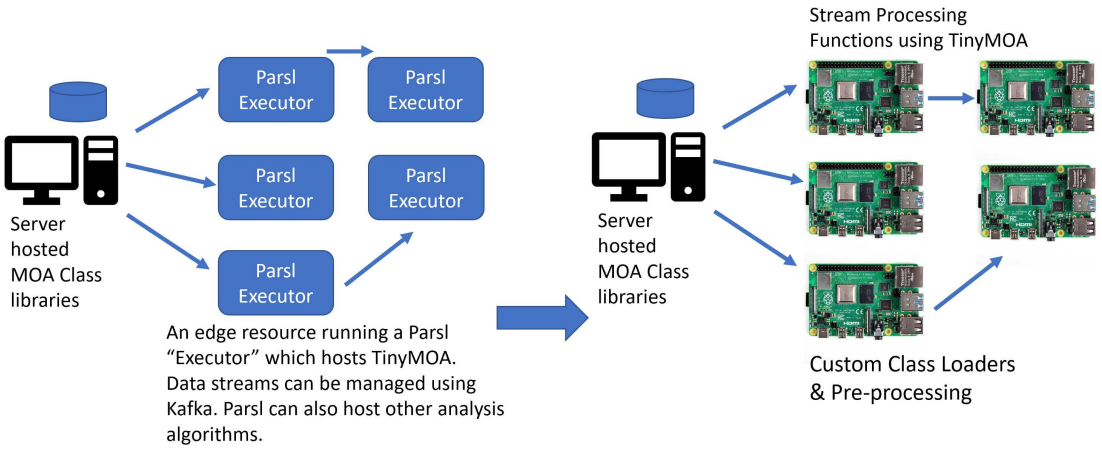

Fig. 1. Left: a number of edge nodes running Parsl executor to host MOA libraries + Kafka; Right: realisation of this architecture using custom class loading mechanisms via a "core" TinyMOA implementation

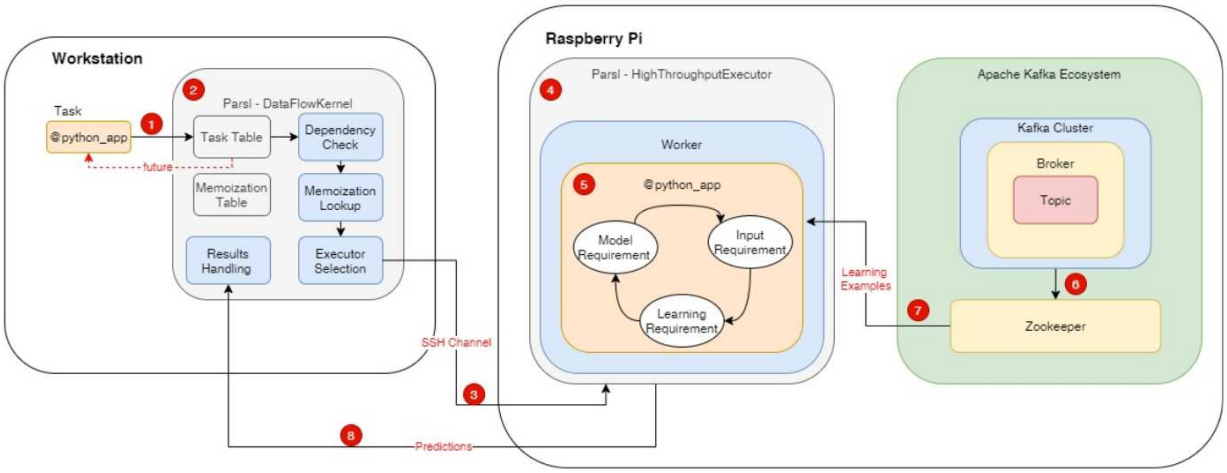

Fig. 2. Experimental testbed components – where MOA algorithms are executed on the Raspberry Pi (RPi) as a Parsl application. A RPi therefore hosts a Parsl Executor and a Kafka Broker. The Parsl DataFlowKernel uses a secure data channel to interact with the Parsl Executor to carry out data processing and execute a stream processing operator. This enables a MOA algorithm to be directly migrated to a RPi based on the capacity of the resource on which it is hosted.

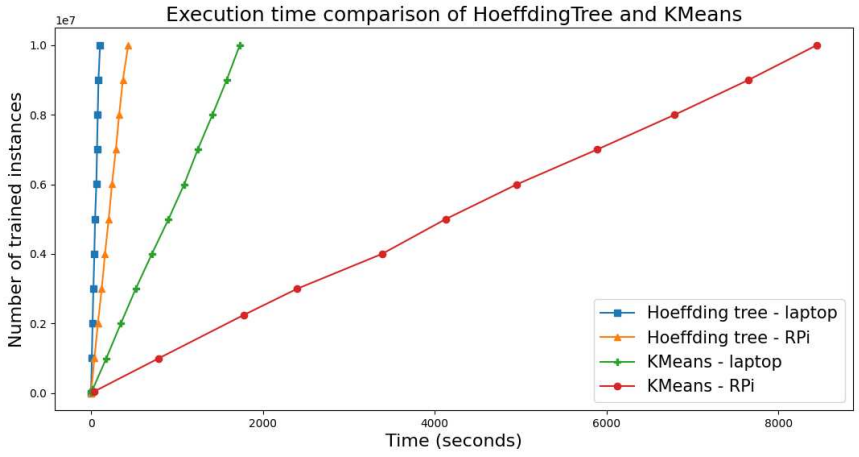

Fig. 3. Execution time Hoeffding Tree and Kmeans: Local execution vs Remote RPi

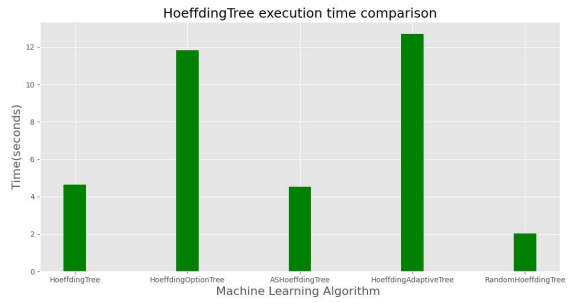

Fig. 4. MOA Hoeffding tree derivations and WaveformStream

b) **Experiment 2: MOA Hoeffding tree derivations and WaveformStream:** In this experiment we use different Hoeffding Tree algorithms including Hoeffding Option Tree [19], Adaptive Size Hoeffding Tree [20], Hoeffding Adaptive Tree [21], and Random Hoeffding Tree [22]. All were tested using the Waveform data stream generator (WaveformStream). In the experiment below, the x-axis represents the type of machine learning algorithm used while the y-axis shows the execution time in seconds. All the learners in this example were trained with 100K instances using the `generators.WaveformGenerator` stream. As illustrated in Figure 4, the execution time of different HoeffdingTree types uses MOA on a RPi4 model B with 2GB RAM and the best performer is the RandomHoeffdingTree followed by the HoeffdingTree and ASHoeffdingTree. The diagram describes only one set of trained examples because the different classification trees display the same relative execution behaviour regardless of the size of the training data.

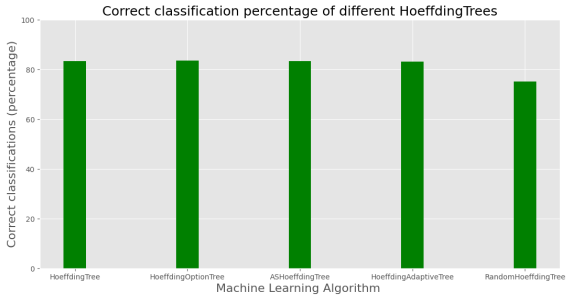

Fig. 5. Percentage of correct classification for different Heoffding Tree types

c) **Experiment 3: Percentage of correct classifications for different HoeffdingTree types:** This scenario is a continuation of the previous experiment in Figure 4, and analyses the accuracy of different Hoeffding tree types using the `EvaluateMethod` provided by MOA, which is used to measure the percentage of correct classifications. The objective of this experiment was to determine which models achieve the best accuracy and speed performance. The experiment configures all learners as trained with 100K instances using the generators.WaveformGenerator stream. As illustrated in Figure 5, the RandomHoeffdingTrees displayed a faster execution time however this is at the cost of accuracy as can be seen in Figure 8 because it has the lowest accuracy score out of the different types of HoeffdingTrees. The remaining HoeffdingTrees displays a very similar accuracy score all having a rating above 80% accuracy.

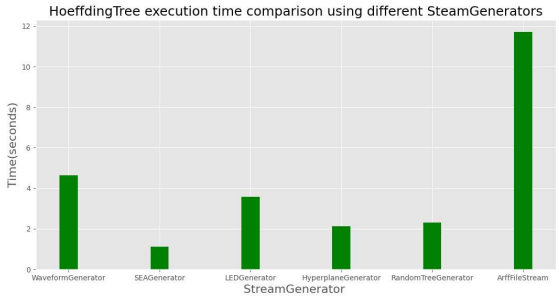

Fig. 6. Comparison of Streamgenerators with HoeffdingTrees

d) **Experiment 4: Hoeffding tree vs related MOA algorithms:** This scenario aims to explore the performances of various MOA algorithms in comparison to the Hoeffding tree algorithms based on different edge configurations. A Kafka installation on RaspberryPis produce data streams and subsequent data changes as required for the algorithms analysis. This experiment conducts comparisons across different stream generators including WaveformGenerator, SEAGenerator, LEDGenerator,HyperplaneGenerator, RandomTreeGenerator and ArffFileStream. The x-axis represents the set of heterogeneous data streams used while the y-axis represents the time-to-completion in seconds. The learner used in this example was a HoeffdingTree which was trained with 100K instances of each data stream. As illustrated in Figure 6,the biggest fluctuations in our results came from the SEAGenerator and the ArffFileStream which show a percentage decrease of 75.8% and a percentage increase of 153.13%, respectively, as time-to-completion when compared to the WaveformGenerator.

e) **Experiment 5: Hoeffding tree with ArffFileStream via Kafka and Parsl:** In this experiment we provide analysis for the execution of HoeffdingTree algorihtm with ArffFileStream with Kafka and Parsl. The y-axis measures the maximum number of examples to train the learner ranging from 5K to 1M. The stream generator used was `generators.ArffFileStream` which stream an .arff file used for training the model in this case is Census.arff. As illustrated in Figure7, we report the execution time on a RPi4 model B with 2GB RAM. The experiment also shows how the execution time fluctuates when MOA algorithm is wrapped in a Parsl file and executed locally through Parsl, when MOA algorithm streams data from a local Kafka server and lastly when MOA together with the Kafka stream is wrapped in a Parsl and executed remotely. The scenarios where MOA has been trained using kafka as a data streaming engine takes a longer time to complete due to the MOA having to consume the Topic data rather than have access to it from a local source on the computer memory.

f) **Experiment 6: Hoeffding tree with Arff FileStream via Websockets and Parsl:** In this experiment we test the

Fig. 7. Comparison of MOA local, MOA with Kafka and MOA with Remote Parsl and Kafka

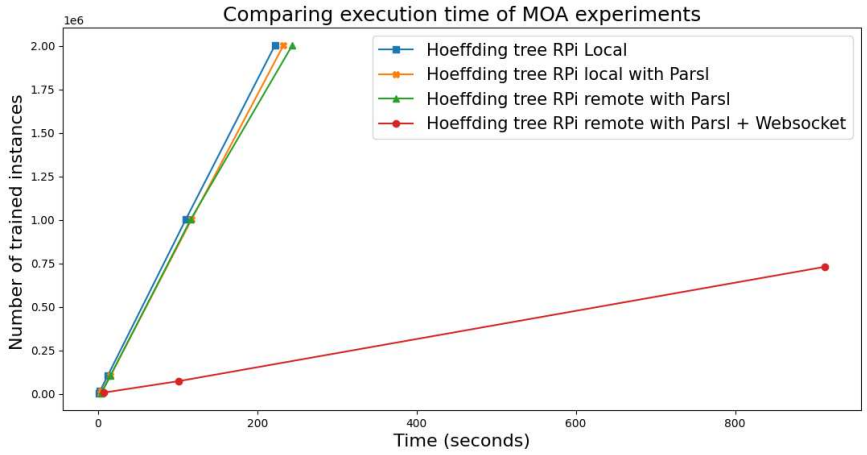

Fig. 8. Comparison of MOA local, MOA with Parsl and MOA with Websocket

execution of a HoeffdingTree algorithm using Parsl and a websocket to channel the incoming set of data for execution. The y-axis measures the number of examples to train the learner ranging from 5000 to 2M. The stream generator used was `generators.ArffFileStream` which streams the Census.arff file for training the model. As illustrated in Figure8, the execution time fluctuates when MOA algorithm is wrapped in a Parsl file and executed locally and remotely – showing the variation in times between Parsl and the Websocket implementation. The limitation with this experiment is that unlike data streams which feed the learning model during its training, in this scenario the file is firstly transported via a TCP-based network connection before the data file can be used by the learning model thus adding an additional delay to the execution time.

g) **Experiment 7: Comparing MOA and TinyMOA on WaveformStream via Parsl:** In this experiment, we consider a set of homogenous machine learning tasks that were trained with 500K training instances of the WaveformGenerators stream. The objective is to analyse machine learning algorithms and their behaviour when executing on edge systems. As illustrated in Figure 9, we observe a 0.02-0.2 second difference in execution time between MOA and TinyMOA. The file size difference does not affect the execution time of the machine learning algorithm when executed through Parsl, but rather follows the same behaviour of the algorithm

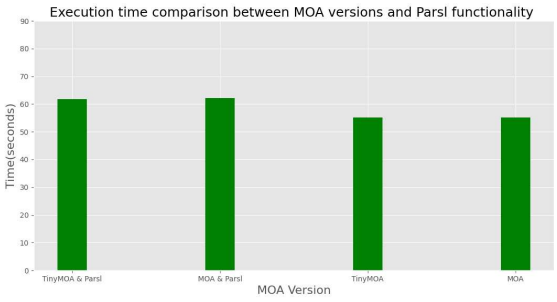

Fig. 9. Comparison of MOA and TinyMOA

when deployed without Parsl. This demonstrates that although significantly smaller in deployment size, TinyMOA offers a viable alternative for a much larger MOA deployment, but on resource constrained devices.

V. Discussion

When applying machine learning to a data stream, the learning algorithm is expected to yield accurate predictions while updating its model in an online fashion without surpassing strict memory and processing time constraints [16]. This raises some key challenges in relation to the functionality in use of the entire workflow. One such challenge is that the model will be processing a massive stream of data in short periods of time, very often with limited computational resources available at the edge. This 'processing' refers to both making predictions and updates to the underlying machine learning model (i.e. training). On top of that, the data may be susceptible to unexpected changes, also known as concept drifts, which must be detected and actioned upon to avoid a catastrophic decrease to the machine learning model predictive performance. In this paper we have used the abstraction of data streams [14] and explore how machine learning tasks can be accommodated at the edge to enable more flexible and autonomous stream execution. We have used the MOA framework on a costumized Parsl edge computing infrastructure to enable the execution of different machine learning algorithms for data stream analysis.

VI. Conclusion

The developement of resource aware stream processing offers a number of benefits: (i) effective use of resources in proximity to data stream generation to support latency-aware application requirements; (ii) limiting the need to transmit a stream across a lossy network to a data centre for analysis. We describe how these benefits can be realised by supporting stream processing on edge resources. We present benchmarks for the execution of different machine learning algorithms in edge configured environments, using an extension to the MOA library – referred to as TinyMOA. Overall, the paper provides the following contributions: a seamless integration of machine learning algorithms, as stream processing operators, using a custom class loading mechanism for MOA. The Parsl system is used to to deploy TinyMOA on edge resources and Kafka is used to partition a stream across different resources executing TinyMOA.

Using several stream analytics experiments on a sample data set, we demonstrate how stream operators can be deployed at the edge by taking into consideration resource constraints. Our experiments can be used to configure an edge environment to make more effective use of such operators. The proposed aproach can be used to manage stream processing on edge resources in a wide range of applications where computation needs to be completed closer to data source.

References

[1] Gopinath, Sridhar, et al. "Compiling KB-sized machine learning models to tiny IoT devices." Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2019.

[2] D. Puschmann, P. Barnaghi and R. Tafazolli, "Adaptive Clustering for Dynamic IoT Data Streams," in IEEE Internet of Things Journal, vol. 4, no. 1, pp. 64-74, Feb. 2017, doi: 10.1109/JIOT.2016.2618909.

[3] Dastjerdi AV, Gupta H, Calheiros RN, Ghosh SK, Buyya R (2016) Fog computing: principles, architectures, and applications. In: Internet of things: principles and paradigms, chap. 4, MorganKaufmann

[4] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," in Big Data and Internet of Things: A Roadmap for Smart Environments. Springer, 2014, pp. 169–186.

[5] M. Aazam and E.-N. Huh, "Fog computing and smart gateway based communication for cloud of things," in Intl. Conference on Future Internet of Things and Cloud (FiCloud). IEEE, 2014, pp. 464–470.

[6] A. R. Zamani, M. Zou, J. Diaz-Montes, I. Petri, O. Rana, A. Anjum and M. Parashar. "Deadline constrained video analysis via in-transit computational environments." IEEE Transactions on Services Computing, 2017.

[7] S. Michiels, W. Horre, W. Joosen and P. Verbaeten, "DAViM: a dynamically adaptable virtual machine for sensor networks", Proc. of Int. Work. on Middleware for sensor networks (MidSens), pp 7–12, Melbourne, Australia, November 28 2006. ACM Press.

[8] P. Levis and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks", Proc. of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X), Oct. 2002. Available at: https://sing.stanford.edu/site/publications/2

[9] D. Balouek-Thomert, et al. "Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows." The Int. Journal of High Performance Computing Applications 33.6 (2019): 1159-1174.

[10] M. Chao and R. Stoleru. "R-mstorm: A resilient mobile stream processing system for dynamic edge networks." IEEE International Conference on Fog Computing (ICFC). IEEE, 2020.

[11] Microsoft AI, Available at https://cloudblogs.microsoft.com/opensource/2021/12/14/add-ai-to-mobile-applications-with-xamarin-and-onnx-runtime/, last accessed Dec. 2021

[12] Adi, Erwin, et al. "Machine learning and data analytics for the IoT." Neural Computing and Applications 32.20 (2020): 16205-16233.

[13] D. Mrozek et al. "A hopping umbrella for fuzzy joining data streams from IoT devices in the cloud and on the edge." IEEE Transactions on Fuzzy Systems 28.5 (2019): 916-928.

[14] A. Bifet, R. Gavalda, G. Holmes, B. Pfahringer, "Machine learning for data streams: with practical examples in MOA". MIT Press; 2018.

[15] Gomes HM, Read J, Bifet A, Barddal JP, Gama J. Machine learning for streaming data: state of the art, challenges, and opportunities. ACM SIGKDD Explorations Newsletter. 2019 Nov 26;21(2):6-22.

[16] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J.M. Wozniak, I. Foster and M. Wilde, "Parsl: Pervasive parallel programming in Python". In Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (pp. 25-36), 2019.

[17] P. Domingos and G. Hulten. "Mining high-speed data streams." Proc. of the 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 2000.

[18] J. MacQueen, "Some methods for classification and analysis of multivariate observations." Proc. of 5th Berkeley symposium on mathematical statistics and probability. Vol. 1. No. 14. 1967.

[19] B. Pfahringer, G. Holmes, and R. Kirkby. "New options for hoeffding trees." Australasian Joint Conference on Artificial Intelligence. Springer, Berlin, Heidelberg, 2007.

[20] A. Bifet, G. Holmes, B. Pfahringer, and R. Gavalda. "Improving adaptive bagging methods for evolving data streams." Asian conference on machine learning, pp. 23-37. Springer, Berlin, Heidelberg, 2009.

[21] A. Bifet, and R. Gavalda. "Adaptive learning from evolving data streams." Int. Symposium on Intelligent Data Analysis, pp. 249-260. Springer, Berlin, Heidelberg, 2009.

[22] A. Bifet, G. Holmes, and B. Pfahringer. "Leveraging bagging for evolving data streams." Joint European conference on machine learning and knowledge discovery in databases, pp. 135-150. Springer, Berlin, Heidelberg, 2010.