Synchronized Preprocessing of Sensor Data

1st Amal Tawakuli
Department of Computer Science
University of Luxembourg
amal.tawakuli@uni.lu

2nd Daniel Kaiser
Department of Computer Science
University of Luxembourg
daniel.kaiser@uni.lu

3rd Thomas Engel
Department of Computer Science
University of Luxembourg
thomas.engel@uni.lu

***Abstract*—Sensor data whether collected for machine learning, deep learning or other applications must be preprocessed to fit input requirements or improve performance and accuracy. Data preparation is an expensive, resource consuming and complex phase often performed centrally on raw data for a specific application. The dataflow between the edge and the cloud can be enhanced in terms of efficiency, reliability and lineage by preprocessing the datasets closer to their data sources. We propose a dedicated data preprocessing framework that distributes preprocessing tasks between a cloud stage and two edge stages to create a dataflow with progressively improving quality. The framework handles heterogenous data and dynamic preprocessing plans simultaneously targeting diverse applications and use cases from different domains. Each stage autonomously executes sensor specific preprocessing plans in parallel while synchronizing the progressive execution and dynamic updates of the preprocessing plans with the other stages. Our approach minimizes the workload on central infrastructures and reduces the resources used for transferring raw data from the edge. We also demonstrate that preprocessing data can be sensor specific rather than application specific and thus can be performed prior to knowing a specific application.**

***Index Terms*—Data Quality, Data Preprocessing, Sensor Data, Edge Computing, Data Management**

I. Introduction

Industries are increasingly relying on Internet of Things (IoT) sensing devices for data-driven and intelligent solutions. This wide spread adoption of sensors is coupled with an exponential increase in the volume and variety of sensor data. Transferring raw sensor data to a central infrastructure for storage and processing consumes valuable resources (bandwidth, storage, CPU time , etc.). Optimizing the consumption of these resources will reduce costs and improve efficiency. Raw data may be incompatible (in terms of size, format, etc.), biased or may consist of outliers. One can identify from literature a common understanding that data preprocessing is the set of operations that transform raw data into quality input. It includes operations such as feature extraction, normalization and noise reduction [1]–[3]. Preprocessing data is necessary to obtain quality input and ultimately quality output. Quality data spans many characteristics including reliability, traceability, compatibility and completeness [4], [5]. Data with such characteristics contribute to the application's performance and prevent errors that could propagate the pipeline causing negative impact and losses. From the technical perspective, quality data contribute to better predictions and classifications and faster convergence (more efficient learning). From the business perspective, quality data result in more reliable decision making, achieve legal compliance and boost operation's efficiency [6]. In many scenarios addressing data preprocessing as early as possible is more effective (e.g. keeping private data at the source) and more efficient (e.g. less bandwidth used when resizing images to fit the input requirements of a convolutional neural network).

Data preprocessing is complex and typically tailored for a specific problem. This is due to the heterogeneity of the data, applications and the contexts [7], [8]. Data is often preprocessed when applications are initiated, which increases preprocessing costs and complexity due to the accumulation of raw data. Batch preprocessing raw sensor data in a central system may take up to 80% of available resources [1], [7]. Edge and fog computing paradigms are popular solutions to the limitations and challenges of the centralized cloud computing model in the context of IoT applications [9]–[11]. Our first hypothesis is that by dividing data preprocessing tasks into stages executed by different entities at different locations (cloud and edge), the challenges of maintaining quality data can be incrementally addressed and conquered. Our second hypothesis is that decoupling data preprocessing from application (processing), not only accelerates data preprocessing but also provides for needed versatility and extensibility to handle new hardware deployments (new sensors) and new requirements (new preprocessing tasks). This means that improving data quality can be generalized and performed as data are generated. The third hypothesis is that by coordinating and synchronizing the preprocessing tasks between the edge and the cloud, it is possible to offer a resilient and dynamic solution that addresses the characteristics of sensor data and IoT.

We propose a three stages framework that progressively improves the quality of the data flowing from sensors (Stage 1), through a central edge preprocessing system (Stage 2) to a central cloud system (Stage 3) creating more structured, maintainable and sustainable dataflow and data lakes. The solution is dedicated for data preprocessing and it is not specific to an application or dataset. The preprocessing between the three stages is synchronized from cloud to edge and coordinated from edge to cloud. The framework can be used under three scenarios. The first is when the best possible sequence of preprocessing tasks has been identified after a thorough analysis of a specific dataset for a specific application or problem. The

978-1-7281-6251-5/20/$31.00 ©2020 IEEE

preprocessing plan is then deployed to our framework to be executed between the different stages. The second scenario involves deploying preprocessing tasks specific to a sensor, data type, data semantic (e.g. car engine temperatures), context or processing technology (e.g. deep learning) but without prior data analysis or identification of an application or problem. For example, by knowing that a sensor is measuring the engine temperature of a passenger car, it is possible to apply quantile clipping to remove outliers. Many sensors deployed at the edge collect data for one or more applications identified before sensor platform deployment. With our framework common preprocessing tasks can be executed once and future applications can benefit from better quality data retrieved from existing sensors thus improving efficiency and reducing preprocessing complexity. The third usage scenario is when the framework is used as part of the problem and data analysis phase to aid data scientists and engineers in testing the devised preprocessing plans in a distributed environment. The contributions of this paper are as follow:

1) Dynamic handling of heterogeneous data and versatile types and numbers of sensors reinforced by the concurrent execution of diverse preprocessing plans.
2) Masterless yet coordinated execution of preprocessing plans and synchronized yet efficient updates of the plans between the cloud and two edge stages.

The above contributions are manifested in a dedicated data preprocessing framework consisting of three stages that progressively improve the quality of sensor data with new communication and data management mechanisms. In addition to the above contributions, we also present our evaluation of a prototype to prove the feasibility of the hypotheses and the effectiveness of the proposed solution. In this paper, we particularly focus on the edge stages of the framework, which are Stage 1 (Smart Sensors Units) and Stage 2 (Edge Engines) and the dataflow from edge to cloud.

II. The Framework

A. Overview

Our framework consists of three stages; one central stage and two stages at the edge. The first edge stage consists of sensors emitting data and it is where sensor data are first handled and preprocessed hence the similarity with the notion of smart sensors. Existing commercial "smart" sensors perform predefined and fixed tasks. This is why we followed the footsteps of many researchers [8], [12]–[14] in developing our own customized smart sensors using a more powerful single-board computer, namely the Raspberry Pi that can be connected with several different types of sensors. We call a Raspberry Pi with connected sensors a Smart Sensor Unit (SSU). The edge consists of multiple sensors as part of one or more SSU(s) making up Stage 1 of the framework. The edge also consists of Stage 2; a middle layer of one or more edge engines, which are devices with greater computational capabilities and storage capacities that can perform further preprocessing on the data received from Stage 1. Data collected from the edge are transferred from Stage 2 to a central or cloud system, which we label as Stage 3. In our prototype, data collected and preprocessed at SSUs are transferred via WiFi to Stage 2 and Stage 3, however other communication technologies can be used depending on the hardware used. Figure 1 depicts an overview of the framework deployed in an automotive setting.

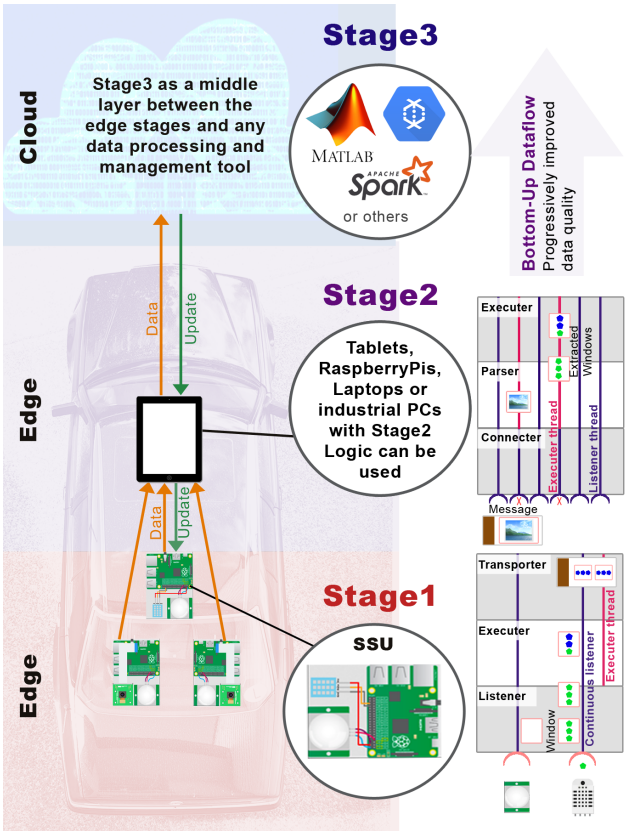

Fig. 1: Framework Architecture.

Each sensor has a preprocessing plan associated to it. The plans are shared and synchronized between all stages and are distributed from Stage 3 down to Stage 1. Data emitted from a connected sensor are preprocessed according to the preprocessing plan associated to that sensor. The framework separates between sensor's preprocessing plans and the preprocessing tasks' logic. Each stage consists of pre-installed preprocessing tasks that can be updated and synchronized down the stages via software updates rather than constantly deploying containers from a central entity thus removing bottlenecks and reducing dependencies. A preprocessing plan instructs each stage on which task to execute, in what order and using which parameters and data inputs (data dependencies). This decoupling facilitates the continuous change of sensors' preprocessing plans to cope with the dynamic nature of data. Our new mechanism to update and synchronize preprocessing plans is efficient. Assuming the relevant tasks are already installed at the edge stages, adding or updating a preprocessing plan of a sensor requires the transmission of one message from Stage 3 down to the edge stages. The framework dynamically

determines the sensor's data type and invokes the relevant functions for data handling and management. Details about the preprocessing plans are discussed in section II-C. The Framework's two-ways communication mechanism is discussed in section II-D. Details about each stage will be presented in sections II-F, II-G, II-H.

B. Requirements and Objectives

The main requirement of the framework is to synchronize and parallelize the execution of diverse preprocessing plans between different stages to transfer quality sensor data to the cloud. We also wanted to use portable edge components and designed the framework to be agnostic to the infrastructure (operating systems and hardware). All stages have certain level of autonomy in the sense that a stage can continue operating and executing the preprocessing plans without a connection with or scheduling from an upper stage. In this paper we focus on the synchronization of preprocessing tasks from Stage 1 to Stage 2 and evaluate the impact of distributed data preprocessing in achieving the following objectives:

1) Reduce resources (e.g. computational time, memory, bandwidth) required to obtain quality data.
2) Handle heterogenous stream data and sensor data from multiple sources concurrently.
3) Facilitate the deployment of preprocessing tasks as soon as data are collected including prior to knowing specific applications to extract information or value.

Achieving the aforementioned objectives will facilitate and automate data preprocessing thus removing the "data preparation" hurdle stopping many organizations from commissioning AI projects and obtaining results faster. The objectives are also in alignment with industry 4.0 principles namely "interconnection" and "decentralized decisions" [15].

C. Preprocessing Plans

The preprocessing plan is a compilation of preprocessing tasks to be executed on data coming from a particular sensor. The intra-stage representation of the preprocessing plan is a Directed Acyclic Graph (DAG). The nodes represent the preprocessing tasks and retain information about the data flowing in and out. They also consist of lists of their predecessors and successors. Nodes that do not depend on predecessors for input are called roots while nodes that do not have successors are identified as sinks. Edges between nodes represent data dependencies between the preprocessing tasks. A preprocessing plan is exchanged between the stages as an encoding specific to our framework (inter-stage representation). The DAG of an active device is constructed locally in each stage at system initialization or prompted by an update using the received inter-stage representation. This approach enables synchronization between the stages while maintaining certain autonomy within the edge stages. It also facilitates DAG updates and their propagation down the stages as changing the preprocessing plan of a sensor is a matter of sending a message with the inter-stage representation of the modified preprocessing plan. Changes may be as simple as updating task parameters or as significant as having completely new preprocessing plans with new tasks and execution sequences. The complexity of the update does not affect the update mechanism; meaning any change will require one message. Other than the message, the edge stages do not further rely on the cloud to deploy and execute the modified or new preprocessing plans. The only exception to this is if any of the tasks within an updated or new preprocessing plan is not installed or deployed at the edge. We assume an identified pool of preprocessing tasks that are deployed in each stage. As future work, new tasks may either be deployed from Stage 3 to the edge stages as containerized applications or directly installed at each edge entity.

DAG Generation A preprocessing plan is transferred from the upper stages as a communication internal to the framework. It is then used by the edge stages to construct the DAG object and its member node objects with properties and functionalities specific to our framework. For such reasons, we designed our own simple and compact text format of the inter-stage representation of the preprocessing plans. The format was specifically designed for the representation of DAG objects consisting of nodes that represent preprocessing tasks and their parameters. In the format, nodes are represented with a unique identifier. Each parent node is associated with its child nodes separated with semicolons. This structure defines the data dependencies between tasks. Only forward (parent to child) relationships need to be represented in the text format. The different relationships are separated with colons. Task parameters for each node are placed between brackets and are separated with commas. The first parameter of each task is mandatory and represents the stage where it should be executed (task allocation). The following is the template:

$$t_1(p_{11}); ...; t_n(p_{n1}, p_{n2}) : ... : t_n(); ...; t_m(p_{m1}, ..., p_{mi}).$$

Where m is the number of nodes, t_m represents the last task and p_{m1} represents the first parameter of task t_m. For demonstration purposes the template includes a task with one parameter, a task with two parameters and one with multiple parameters in that order, however, the sequence and location of tasks does not affect the number of parameters; this is merely dependent on the functionality. Nodes repeated in the encoding do not need to have their parameters restated making the structure more compact. Figure 2 shows an example of a DAG representation of a preprocessing plan with the following inter-stage representation:

$$A1(1); B(1, 105.7); C(1) : B; A2(1, 0, 100) : C; A2.$$

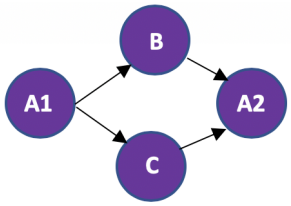

Fig. 2: Intra-stage representation

Given the inter-stage representation of a preprocessing plan, each stage generates the DAG (intra-stage representation). The DAG deserialization function recognizes nodes that have been already added to the DAG object, and only constructs one instance of a node. The function only extracts parameters of newly created nodes and thus triggers dictionary updates of task parameters once per node. The function also automatically determines backward relationships (child to parent) and adds them to the relevant nodes.

DAG Execution Order There are dependencies between most preprocessing tasks and thus they must be executed sequentially. For example, standardization precedes principle component analysis (PCA) and PCA precedes features reduction. This makes our three stages architecture well suited for data preprocessing. Stage 2 and Stage 3 can also execute preprocessing tasks in parallel given there are no dependencies between them. Edge stages produce one output for each input received from the previous stage or a data source. This design choice avoids amplifying the volume of data transmitted between the stages which would defeat one of our main objectives. To control the volume of the transmitted data and ensure operations' integrity and completeness, the following rules are applied for the execution of the shared DAG:

1) Stage 1 must execute all the root nodes as they expect the raw data as input or otherwise none of the tasks are executed in Stage 1 (if the latter case applies then rule 1 applies to Stage 2). Exception to this rule if only one of the stage outputs is a vector or matrix and the rest of the outputs are scalar values.
2) A single collider node (inverted fork) and its predecessors as far as the last single collider node must be executed at the same stage (applicable on edge stages).
3) A node can be executed if all of its parent nodes have completed execution and produced output.
4) At an edge stage, the execution of nodes would result in one stage output to be transferred over the network.

These rules in addition to ongoing work would help us further extend the framework to partially automate the allocation of the preprocessing tasks to the different stages, which is an advantage over existing solutions.

D. 2-way Communication

The data flows within the framework in two directions; each direction serves a purpose as explained in the following:

Top-Down Dataflow Messages sent from Stage 3 down to Stage 1 are for the purpose of propagating updates across the framework's stages; mainly for preprocessing plan and task parameter updates. The messages consist of the identification of one or more sensors and the inter-stage representation of their preprocessing plans. All stages receive the same copy of the preprocessing plan for each sensor. The messages are either updates of existing preprocessing plans or initialization of new sensors. Initialization messages standardize sensor details across the stages as they invoke local stage dictionary updates. These dictionaries hold information about connected sensors including their identification, data type, data semantic and preprocessing plan. Due to the minimalist text representation of the preprocessing plan, top-down messages are compact even when they contain updates for multiple sensors. The lower stages do not require further control messages to perform their operations and preprocess sensor data and thus top-down dataflow is more efficient and less likely to suffer from bottlenecks or disruption from upper stage failure.

Bottom-Up Dataflow Data sent from Stage 1 up to Stage 3 are sensor data that have been completely or partially preprocessed. It is also possible for the data to arrive at Stage 3 without preprocessing due to intended configuration or a fault. Such case has an impact on efficiency and performance but not on system integrity as the preprocessing plans can be completely executed at Stage 3. A synchronized execution of shared preprocessing plans makes the framework fault tolerant and resilient as upper stages can pick-up where the previous stage has stopped. If a lower stage fails to execute the tasks designated to it, then the next stage executes these tasks in addition to its own task allocation. This mechanism is enabled by the bottom-up messages, which consist of information about the DAG status and the execution progress including the stage at which each task was executed and when they were executed. The messages also consist of the sensor data, labels (if applicable), the time the messages were created, the sensor and SSU identifications. The metadata ensures data provenance and lineage. A message consists of one or more windows of data generated by one sensor. The data is encapsulated with their metadata using the following format:

$$sensorID, ssuID, engineID, time, label, DAGStatus, \{data\}.$$

The following is the DAG status format:

$$n_1 : sn : t_1; n_2 : sn : t_2; ...; n_{\mathrm{n}} : sn : t_{\mathrm{n}}.$$

Where n_i represents a task in the preprocessing plan, *sn* is the stage number at which the task was executed and t_i is the timestamp when the task n_i execution was completed. All sensor data are encoded before being incorporated to the messages. At the destination, the data are decoded back to their original type or format during task execution. The edge stages can be configured so that the data are accumulated in a buffer (batched) until a certain threshold is reached before being sent off to the upper stage or otherwise tuples are sent off once their preprocessing has been completed (streamed). The size of the message can be changed per sensor. Our prototype sends fixed sized messages containing encoded image data and smaller variable sized messages containing encoded numerical data from one SSU. Some preprocessing tasks may not result in the reduction of data volume, which means that our framework increases the message's payload with the added metadata. In this case, performance enhancement is observed during data processing and model training with added insights and data lineage. This direction of the dataflow has more traffic and requires more bandwidth per message. Stage 3 may receive multiple messages from multiple edge engines and Stage 2 may receive multiple streams of data from multiple SSUs. Listener threads are created at Stage 2 and Stage 3 to handle multiple messages concurrently and cope with the larger traffic volume. Details are discussed in sections II-G, II-H.

Inter-stage Network Setup All stages can act as client and server. For bottom-up dataflow the respective lower stage acts

as client. The roles are reversed for the top-down dataflow. TCP is used to transmit messages between stages as we require reliable data streams. While our current prototype uses fixed IP addresses, our framework is designed for configurationless service discovery. We plan to use DNS Service Discovery over Multicast DNS (DNS-SD/mDNS) [16], [17]. It provides our framework with the necessary flexibility also in terms of network configuration. Since we focus on supporting mobile setups, it is important to take privacy aspects of configuration-less service discovery into consideration [18], [19].

E. The Data

The framework handles structured data (dates, numbers, etc.) and unstructured data (images, audio, etc.). In addition to the possibility of collecting spatial data, the framework includes temporal and data source information with the collected sensor data. This integral part of the framework allows for the deployment of spatio-temporal correlation and filtering as part of a preprocessing plan and facilitates data lineage. Semantic and contextual information about the sensed data facilitate the identification of sensor specific preprocessing and the automation of data normalization and cleaning techniques prior to identifying any applications to extract value. Data collected from sensors undergo several phases at each stage including windowing to slice unbounded data into finite chunks. Stage 1 and Stage 2 create fixed-sized tuple-based windows, which are collected in messages to be transferred to an upper stage. In our prototype, the receiving stage extracts window(s) from a message and preprocesses window content directly without further windowing or partitioning. This makes our windowing approach unique as it occurs at the sending end rather than at the receiving end. It is also possible to configure the framework to perform temporal windowing at the receiving stage should it be required in future projects. The window size, which is the maximum volume of data a window can contain is based on data volume in bytes. The window size (threshold) can be modified to have one tuple per window for stream execution or several tuples for batch execution. Each sensor can have its own window and message sizes and the framework can concurrently preprocess different sized windows from different data sources.

In some cases, access to raw data may be needed due to ineffective or erroneous outcomes of the preprocessing tasks applied. There is an inverse correlation between maintaining resources and keeping raw data. The problem domain and data owner's objectives and goals determine whether priority is given to data value or resource value. Our project addresses the challenge of optimally maintaining limited resources consumed by raw data during their storage, transmission and preprocessing. Particularly when raw data eventually ends up being reduced, transformed or cleaned before being used as input. Having said that, there are few techniques integrated into the framework to achieve a better balance between keeping the raw data and reducing resources. One technique is preserving raw data during the execution of the preprocessing plan to enable rollbacks. The SSU can also be configured to store raw data temporarily and for a certain duration. It is also an option to send the raw data to edge engines via a local network. Edge engines can then store the raw data locally or send batches of raw data to an external storage system via a separate communication channel. With this option both preprocessed and raw data are transmitted via the network, however, the burden of storing and preprocessing data on the central system is still reduced. Lastly, because all stages share the same copy of the preprocessing plan including task parameters, it is also possible for some preprocessing tasks to be reverted and thus obtain the data prior to their execution.

F. Stage 1

The framework exploits the notion of smart sensors to perform frontline preprocessing as data are collected. Stage 1 may consist of one or many sensors connected to SSUs. Each sensor has a preprocessing plan associated to it and predefined information stored in the SSU. Sensor information include sensor identification, device type, data type, data unit, data semantic and GPIO pins, which are all initialized as part of defining new sensors. Adding a new sensor requires physically connecting the sensor to an SSU and updating the relevant dictionaries with the aforementioned information. The new sensor can be associated with its preprocessing plan via an update sent from Stage 3. The preprocessing plans of sensors can also be dynamically and seamlessly updated whenever required. The SSU logic does not require code modifications to recognize the new sensor or handle its data. An SSU listens for data from each sensor and execute the relevant preprocessing plans on new data concurrently. We used Raspberry Pi 4 Model B with 4GB LPDDR4 RAM and 256 GB memory to create a prototype SSU. We implemented Stage 1 using C++ programming language and used WiringPi GPIO access library and OpenCV framework to handle image data. The software architecture of Stage 1 consists of three main components as illustrated in figures 1 and 3.

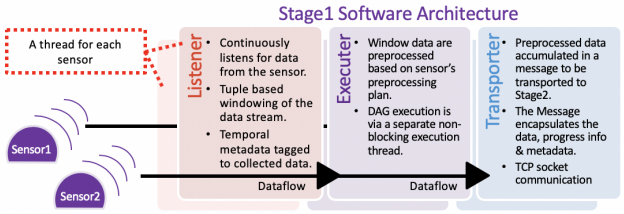

Fig. 3: Stage1 Software Architecture

The listener: The listener component creates a thread for each sensor to concurrently listen for their data. The listener continuously listens for input from sensors without interruption or blocking. Any collected data is tagged with a timestamp. This preprocessing task is one of few tasks the system performs that is not part of a DAG. In other words, it is mandatory and applied to all data. The listener adds collected data to the window. Once the window's size reaches exactly or nearly (if data volume varies) the threshold, the current window is shipped to the executer component

while the listener continues listening for data and fills the next windows. For our prototype, windows' sizes vary and depend on the sensors and their data types, for example the maximum window size for numerical data is 128 bytes while it is hundreds of kilobytes for image data. The system can be configured to modify the window size for each sensor.

The Executer: A full window is shipped to the executer for the data to be preprocessed based on the device's specific preprocessing plan. One of the advantages of the framework's design is that the DAG of each sensor is only constructed once at initialization or after an update. This means that the data are preprocessed directly without the delays of constructing the DAG each time a window is shipped. Another design choice that was made to promptly preprocess real-time data streams is the creation of a separate thread for the execution of the preprocessing plan. The execution thread runs concurrently with and independently from the sensor's listener thread. The execution thread, however, terminates when the executions of all the preprocessing tasks allocated for Stage 1 are complete. The execution of preprocessing tasks on the current window occurs in a sequential order using depth-first traversal of the DAG. Assuming for illustrative purposes that the complete preprocessing plan of a sensor is to be executed at Stage 1, the sequential execution of the DAG starts with finding the root nodes and executing one root node at a time and propagating down its child nodes. Child nodes are executed recursively until the last sink node is reached and executed. A child node is only executed when all the data from its parent nodes are received, otherwise execution moves to the next child node or an upper node. This is important to ensure the credibility of the outcome and completeness of the execution. At the same time, it guarantees correct and smooth coordination between the stages. If the preprocessing plan is partly executing at Stage 1 then Stage 2 will continue executing the preprocessing plan without having execution gaps in the DAG and by receiving valid and complete data for the next task(s). During execution, each node is updated with its execution progress and success, which is information useful for the coordination between the different stages.

The Transporter: The result of Stage 1 execution is passed to the transporter, which is the component responsible for the preparation and transfer of messages containing Stage 1 output to Stage 2 for further preprocessing or data relay. The transporter receives an output window and based on its size and data type determines whether to send it directly to Stage 2 or add it to a buffer of accumulated ready to send data. Once the buffer reaches the threshold, the transporter prepares the message. The message includes one or more output window and the metadata as described in section II-D. The transporter creates a TCP socket to connect with Stage 2 and attempts to send the message. If the connection or transmission fails, the transporter stores the message for later attempts.

G. Stage 2

Stage 2 adds another preprocessing layer at the edge. An Edge Engine is designed to handle multiple data streams from multiple SSUs. Stage 2 may consists of one or more edge engine(s). The data received at Stage 2 may be batches or streams of raw or preprocessed data from Stage 1. The execution of the preprocessing plan at Stage 2 depends on two factors; the first is the allocation of tasks extracted from the preprocessing plan and the second is the progress status of the DAG received within messages from Stage 1. The engine either continues the execution of the preprocessing plan based on the original task allocation or starts with the tasks allocated to Stage 1 but were not executed due to an error. Most of the components of Stage 2 are written in C++. OpenCV library is used to handle image data and POSIX socket is used for the communications with Stage 1 and Stage 3. Stage 2 can run in any hardware with memory capacity, computation capability and connectivity sufficient for users' projects. This may be a RaspberryPi, Tablet, Laptop or an industry-grade compact PC. For our prototype, a key consideration, in addition to sufficient computation power, large storage capacity and wireless connectivity, was mobility as we plan to deploy the prototype in an automotive setup. Based on these requirements we chose to use a tablet with 6GB RAM and 1TB memory. The starting point of an edge engine involves creating listener threads that receive messages of varying sizes from Stage 1. In our prototype, the number of threads is equal to the number of sensors connected to the framework's SSUs. A thread, however is not dedicated to a sensor but rather to the SSU. This means multiple threads may be handling data generated by one sensor while having at least one thread listening to data from each SSU. Figure 1 includes an illustration of Stage 2 architecture.

The Connector: At each thread a connector establishes a connection with an SSU. The connector component also establishes a connection with Stage 3 to send sensor data and receive updates of the preprocessing plans.

The Parser: The connector ships messages to the parser component, which decodes the messages and extracts the window(s). A message consists of data from one sensor encapsulated in one or more window(s). The parser ships the extracted window(s) to the executer to execute the sensor's preprocessing plan on them.

The Executer: At the executer, a thread switches from listener mode to execution mode. Stage 2 creates concurrent execution flows for data received from different sensors within a single SSU. This level of concurrency allows for simultaneous execution and progression of independent preprocessing plans preventing unnecessary queueing and delays. The execution mechanism implemented in Stage 2 differs from Stage 1 in that the DAG is not executed sequentially. Nodes that are independent from each other and have received all their input data are executed concurrently via separate threads. Assuming for illustrative reasons that the preprocessing plan is executed completely at Stage 2, the executer starts with the root nodes by creating execution threads for each root and executing their preprocessing tasks. The outputs of the root nodes are added to their child nodes. Child nodes are executed concurrently when all the data from their predecessors are received. The nodes of the DAG are recursively executed under the later condition

until the last sink node is executed. The thread then switches back to listener mode. This switching mechanism does not stop the process of receiving data even from the current sensor as other active threads dedicated for the current and other SSUs can be in listening mode and receiving data. The number of threads created can be increased or decreased according to the application and activity of sensors. Data preprocessed at Stage 2 are sent to Stage 3 or stored locally for later transfers.

H. Stage 3

Relative to the framework's edge stages, sensor data received at Stage 3 are no longer characterized as real-time, however, compared with existing stream processing frameworks (e.g. Apache Flink) data arriving at this stage would still be within the range of real-time. Stage 3 executes the remaining preprocessing tasks on arriving data. It acts as a middle layer between our framework and existing big data tools for storage, processing or analytics. Data can either be processed directly after Stage 3 preprocessing or stored for batch processing using tools selected by the user. We currently use MATLAB (R2019b) to perform batch processing experiments and evaluate the impact of using our framework on performance, resource consumption, model training and prediction accuracy. Our experiments included using machine learning (e.g. Multi-class Support Vector Machine) and deep learning (e.g. GoogLeNet) models on raw and preprocessed data received from the framework. Details about some of the experiments are discussed in section III.

One of the main functions of Stage 3 is to allow users to control the preprocessing applied to the data emitted by each sensor remotely. For example, changing the per channel means to normalize production images can be done by sending to Stage 2 the inter-stage representation of the preprocessing plan consisting of the new mean values as parameters of the normalization task. The updated parameters will be relayed from Stage 2 to Stage 1 where the sensors reside. Changes to the preprocessing plan may be more drastic such as adding completely new preprocessing tasks to be executed on data generated from a particular sensor. Such changes may be the result of identifying new applications and opportunities to extract value from the data. Stage 3 also sends synchronization messages to the lower stages when new sensors and SSUs are added. These messages ensure dictionaries at all stages consist of uniform and up-to-date information about sensors and SSUs. We plan to integrate a partially automatic recommendation mechanism of stages at which tasks within a preprocessing plan can be optimally executed. The conditions discussed in sections II-C will be used as the basis of the automatic recommendation mechanism. Currently, the task allocations are manually included in the inter-stage messages.

III. Experimental Results

We tested the framework with different data types including numerical data and images. For the image data use case we assumed an identified application, which flower image classification. We simulated image capturing at the SSU (Stage 1). We used the Flower dataset from the university of Oxford and extracted 12 categories. Each category consists of 75 training images and 5 test images. The dataset included images of varying dimensions. We standardized the size of all images to 260*260*3 for two reasons, to control the maximum size of messages transmitted between the stages and to provide more accurate bandwidth calculations. Image data are classified at Stage 3 (MATLAB) using the convolutional neural network (CNN) AlexNet [20]. Transfer learning technique is used on a pre-trained AlexNet by replacing the last three layers (Fully Connected Layer, Softmax Layer and Classification Layer) to adapt the network to our classification application and its categories. The data layer of AlexNet takes images with 277 by 277 by 3 dimensions. Thus, image resizing is a mandatory preprocessing task on data with different dimensions to fit the input requirements. The authors of AlexNet also normalized the input data by zero-centering the pixels of each image [20]. We implemented the preprocessing tasks image resize and pixel zero-centering normalization and deployed them at Stage 1 and Stage 2. The image resize task takes the new dimensions as parameters while the normalization task takes the channel means as parameters. The per-channel means were calculated from the training dataset and used on all the datasets (training, testing and validation). The first experiment we performed was to evaluate the impact of sending preprocessed images on resource consumption and efficiency. The image dataset was first sent from Stage 1 to Stage 2 without preprocessing. The same dataset was then preprocessed at Stage 1 and transmitted to Stage 2. Both image-resize and zero-centering tasks were executed at Stage 1. Reducing the size of images has an obvious impact on transfer time and bandwidth usage, which is visualized in diagrams 4. Another benefit of the two preprocessing tasks applied is that all information has been maintained, meaning none of the images have been deleted and the original pixel values can be restored at the upper stages since all stages share the same per-channel mean values.

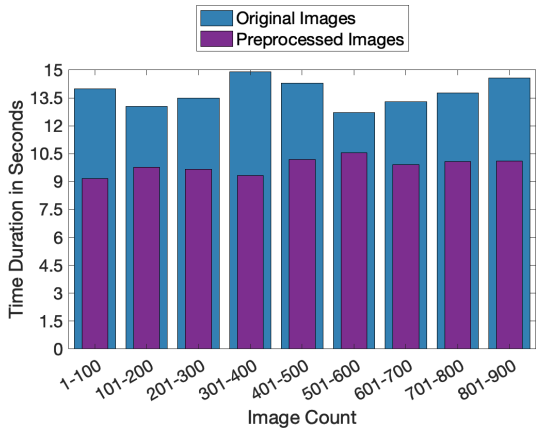

Fig. 4: Data Transfer Duration per 100 Images

The second set of experiments was designed to evaluate the impact of distributing the workload to the edge that would otherwise be solely executed at a central system. We measured

the execution time of the preprocessing tasks *image-resize* and *zero-centering* on different volumes of training datasets. We started with a 20MB volume dataset and squared the volumes up to 640MB using the same flower dataset. Figure 5 shows the average execution duration for each volume. The preprocessing time of data larger than 160MB is no longer proportional to the data volume and that there is a significant increase in the duration of execution. This means that distributed preprocessing would have a greater impact on overall training time as the size of the data increases. The results reflect the time required to perform relatively simple preprocessing tasks. We plan to investigate the impact of distributing more complex preprocessing tasks to the edge. In our image classification experiments, we worked with relatively small datasets (24.7 MB training dataset and 1.7 MB test dataset), however, the results remain promising and a good indication of preprocessing datasets in giga bytes volume.

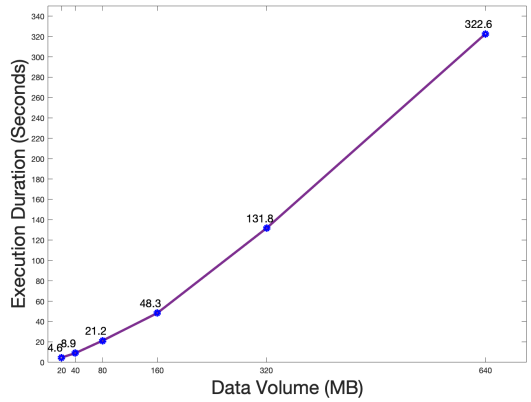

Fig. 5: Preprocessing (resizing & centering) Time of Different Volumes

The average execution time of resizing 900 images is approximately 25% less in Stage 2 than in Stage 1 and the average execution time of normalziation is approximately 33% less. This is expected as the SSUs have lower computational capabilities than the upper stages, however, utilizing the computation power of many edge devices not only reduce bandwidth usage but also reduce workload on central systems. We also measured the prediction time difference between raw data, which are preprocessed in MATLAB, and ready to predict data that were preprocessed at Stage 1. The prediction duration of data preprocessed at the edge saw an average reduction of 56.5% compared to raw data. The results also show more consistent prediction times for images preprocessed at the edge. The prediction durations of test data preprocessed at Stage 1 and Stage 3 are illustrated in figure 6. The figure reflects the results of tests repeated 14 times on each dataset.

For our numerical data use case we assumed an application has not yet been identified. We connected a DHT22 Temperature and Humidity sensor to the same SSU used to simulate the image capturing use case. The framework handled data generated from both data sources and executed the relevant preprocessing plans concurrently on data windows. Two preprocessing tasks were applied on the data collected

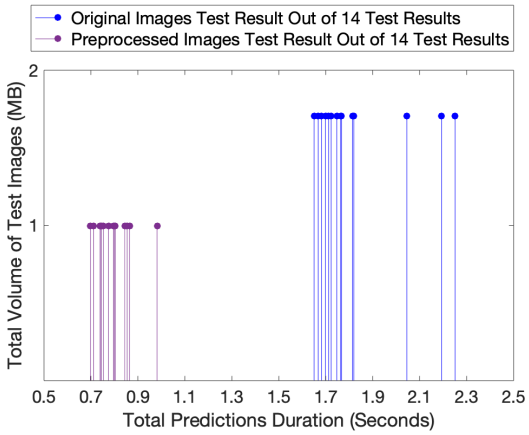

Fig. 6: Impact of Edge Preprocessing on Prediction Time

from the DHT22 sensor. The first is rounding the readings to a specified decimal place(s) and the second is scaling the readings using min-max normalization technique to keep both types of readings (temperature and humidity) on a similar scale. We placed the SSU in a vehicle and collected temperature and humidity data inside the vehicle. Both preprocessing tasks were executed on the data as they were collected in Stage 1. The raw and preprocessed data were sent to Stage 3 for analysis. In this use case per message volume reduction is not impactful especially since the preprocessing tasks do not result in significant reduction compared to the added metadata. The edge preprocessing, however, removed unnecessary data volumes and replaced it with metadata about data lineage. MATLAB function was used to fit linear regression models using the raw data and the preprocessed data. Performing analysis of variance (ANOVA) for each model exhibit a smaller mean squared error (MSE) for preprocessed data model than for the raw data model.

IV. Related Work

The majority of existing data preprocessing solutions are central or cloud-based libraries and tools. MATLAB provides an extensive list of preprocessing functions under different toolboxes depending on the data type and application [21]. Python machine learning library scikit-learn includes a preprocessing package that provides several preprocessing functions to perform standardization, normalization, encoding categorical features and imputation of missing values [22]. Cloud DataPrep is a cloud service that imports raw data and provides users with visual tools to explore, clean and prepare data providing estimations and recommendations of the next preprocessing tasks. The tool also automatically detects schemas and data anomalies [23]. Google Data Validation system is a cloud tool deployed in Google Tensorflow ecosystem that automatically generates data schemas to validate batches of data, detect anomalies and minimizes feedback loop [24]. Bosch IoT Analytics provides in cloud preprocessing of data collected from IoT devices. Preprocessing services include anomaly detection, normalization and data aggregation [25].

Authors in [26] proposed an Apache Flink library for preprocessing streaming data. The library includes six preprocessing algorithms implemented for Apache Flink. Some of the selected algorithms are for feature selection such as the Fast Correlation-Based Filter (FCBF) and Online Feature Selection (OFS). The other selected algorithms are for data discretization such as Incremental Discretization Algorithm (IDA) and Local Online Fusion Discretizer (LOFD) [26]. Featuretools is a central python framework that automates feature engineering. "It excels at transforming temporal and relational datasets into feature matrices for machine learning" [27]

Recently, solutions have been proposed to move data preprocessing and processing to the edge. Microsoft Azure IoT edge is a fully managed service built on Azure IoT Hub that deploys containerized applications to edge devices. Users must then define workload descriptions which are files that lists the containers to be deployed to a particular edge device type. Once online, edge devices communicate with Azure IoT Hub to obtain workload descriptions from which the edge devices download Docker containers from the cloud. Any update to the preprocessing plan requires the deployment of the containerized services even if they were previously deployed. Health status messages of edge devices are also communicated to the cloud for monitoring [28]. Amazon offers AWS IoT Greengrass which extends AWS to edge devices to utilize AWS Lambda functions or Docker containers. The solution is not a dedicated preprocessing framework and does not coordinate data preparation between the cloud and the edge. It is up to the user to configure the edge from the cloud to deploy application-specific preprocessing tasks to be executed by one device or a cluster of edge devices. The execution of the preprocessing tasks, however, is independent within a device and not collective and coordinated between devices within a cluster [29]. Dell also offers an Edge Gateway for IoT which enables customers to preprocess and exploit data at the edge [30]. The edge solutions offered by Microsoft, Amazon and Dell assume applications and their optimal preprocessing tasks are identified. It is also not clear whether partial deployment of the preprocessing plans is supported and how users can identify whether the data received in the cloud are preprocessed or not. Users also need to either have an account at their cloud infrastructure or use vendor specific software and hardware. Big Automotive Data (BAuD) [31] is a framework for automotive telematics and data analytics, which consist of an in-vehicle telematics components connected with a cloud-based analytics framework. In addition to transmitting data generated in the vehicle to the cloud, the telematics component can perform preprocessing and analytics on the collected data. BAuD is a two-way-communication system where measurement tasks (for data capturing and preprocessing) are sent from the cloud to the telematics system for execution and at the same time data captured in the vehicles are streamed or sent as batches to the cloud. Authors of [8] proposed a three-tier framework for Big Data preprocessing in smart cities applications. The framework consists of a command center that is connected to one or more cluster(s) of edge agents. Each cluster has a cluster head which is an agent with greater computation power and storage capacity. The cluster head relays requests from the command center to the edge entities and transfers data from agents to the command center. They adopted the MapReduce programming model and Bulk Synchronous Parallel (BSP) model for the execution of the preprocessing tasks at the edge and cloud. MapReduce and BSP are not optimal options, performance wise, when it comes to stream processing. Authors of [32] proposed a framework that preforms edge preprocessing on sound data to filter features that compromise privacy. By transforming emotion feature and preserving features related to speech content at the edge, only non-sensitive features can be transferred to the cloud for speech analysis. The framework can be extended to distribute other data preprocessing tasks but it is restricted to signal data. Distributing Machine Learning and Deep Learning to the edge or fog, which involves preprocessing, is also an active research field [10], [33]–[35].

From our literature review, we observe a lack of clear and strong synchronization between the cloud and the edge particularly in terms of collective and coordinated execution of the preprocessing tasks. There is also an imbalance of preprocessing tasks distribution, in other words, preprocessing tasks are either completely performed in the cloud or completely imported to the edge. The later scenario deems any failure of executing preprocessing tasks at an edge component impactful particularly with a lack of a dynamic mitigation mechanism, as the central components assume data is ready for consumption. Solutions that distribute preprocessing tasks at the edge are designed for manual configuration and updates and require users to have prior knowledge of the application. It is often that use cases and applications are identified after data collection, which render many of the solutions with such assumption ineffective and data will continue to be collected as raw. We also noticed that edge components have limited autonomy and depend on a central entity to repeatedly retrieve tasks for execution. Additionally, continuous communication is required between edge components and the cloud for monitoring purposes. Our framework addresses these limitations and, to the best of our knowledge, it is the first to provide a dedicated, synchronized and coordinated data preprocessing that can be both application specific and agnostic. The framework facilitates our research in generalizing applicable preprocessing operations and progressively improving the quality of the dataflow with prior preprocessing tasks determining the parameters or techniques of the next preprocessing tasks.

V. Conclusion and Future Work

Many opportunities to capitalize data are hindered by an expensive and complex data preprocessing phase. The majority of data collected from the edge are moved and stored as raw data. With the growing number of applications for IoT devices, addressing the challenge of preprocessing sensor data efficiently and effectively is becoming more vital. In this paper, we introduced a dedicated data preprocessing framework with three stages that synchronize and coordinate the execution

of sensor specific preprocessing plans. We demonstrated the feasibility of concurrently executing different preprocessing plans close to the data sources. Our new approach facilitates the updating of preprocessing tasks to cope with the dynamic nature of data. Our first prototype achieved promising results in reducing resources required to preprocess data and reduce the time to obtain information and value. The preprocessed data produced by the framework can be easily traced and extracted from storage and data lakes based on data source, data provenance or temporal criteria.

We plan to deploy and test more complex preprocessing plans at the edge including feature extraction and missing data analysis. We intend to extend the diversity of the data collected to include audio signals and video streams directly from connected cameras and microphones. We are also looking to address data privacy and fairness and deploy preprocessing tasks that eliminate data biases and protect personal data at the edge. The framework will particularly excel in applications where mobility and diversity of data are main requirements. We plan to experiment and evaluate the use of the framework within an automotive application as a next step. We also plan to evaluate our framework against related tools to assess the validity of our framework and contributions with regards to efficiency, performance and data quality.

REFERENCES

[1] Y. Huang, M. Milani, and F. Chiang, "PACAS: Privacy-aware, data cleaning-as-a-service," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, dec 2018.

[2] C. Li, "Preprocessing methods and pipelines of data mining: An overview."

[3] S. Ruan, R. Li, J. Bao, T. He, and Y. Zheng, "CloudTP: A cloud-based flexible trajectory preprocessing framework," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, apr 2018.

[4] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 01 2012.

[5] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino, "Methodologies for data quality assessment and improvement," *ACM Computing Surveys*, vol. 41, no. 3, Jul. 2009. [Online]. Available: https://doi.org/10.1145/1541880.1541883

[6] H. Moreno. (2017) The importance of data quality - good, bad or ugly. [Online]. Available: https://www.forbes.com/sites/forbesinsights/2017/06/05/the-importance-of-data-quality-good-bad-or-ugly/#3fe7d65010c4

[7] S. Krishnan, M. J. Franklin, K. Goldberg, J. Wang, and E. Wu, "Activeclean: An interactive data cleaning framework for modern machine learning," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 21172120. [Online]. Available: https://doi.org/10.1145/2882903.2899409

[8] B. Mohebali, A. Tahmassebi, A. H. Gandomi, and A. Meyer-Bse, "A big data inspired preprocessing scheme for bandwidth use optimization in smart cities applications using raspberry pi," in *Big Data: Learning, Analytics, and Applications*, F. Ahmad, Ed. SPIE, may 2019.

[9] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, "The emerging landscape of edge computing," *GetMobile: Mobile Computing and Communications*, vol. 23, no. 4, pp. 11–20, may 2020.

[10] H. Li, K. Ota, and M. Dong, "Learning IoT in edge: Deep learning for the internet of things with edge computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, jan 2018.

[11] M. G. S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, "Machine learning at the network edge: A survey."

[12] R. Aloufi, H. Haddadi, and D. Boyle, "Privacy preserving speech analysis using emotion filtering at the edge," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*. ACM, nov 2019.

[13] Q. Luo and M. Xie, "Temperature and humidity detection system of communication system based on raspberry pi," in *2018 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS)*. IEEE, jan 2018.

[14] D. A. Winkler and A. E. Cerpa, "WISDOM," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*. ACM, nov 2019.

[15] X. Zheng, M. Wang, and J. Ordieres-Meré, "Comparison of data preprocessing approaches for applying deep learning to human activity recognition in the context of industry 4.0," *Sensors*, vol. 18, no. 7, p. 2146, jul 2018.

[16] S. Cheshire and M. Krochmal, "DNS-Based Service Discovery," RFC 6763, Feb. 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc6763.txt

[17] ——, "Multicast DNS," RFC 6762, Feb. 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc6762.txt

[18] D. Kaiser and M. Waldvogel, "Adding privacy to multicast DNS service discovery," in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, sep 2014. [Online]. Available: https://doi.org/10.1109%2Ftrustcom.2014.107

[19] ——, "Efficient privacy preserving multicast DNS service discovery," in *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*. IEEE, aug 2014. [Online]. Available: https://doi.org/10.1109%2Fhpcc.2014.141

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, may 2017.

[21] MathWorks. Preprocessing data. [Online]. Available: https://nl.mathworks.com/help/matlab/preprocessing-data.html

[22] scikit learn. Preprocessing data. [Online]. Available: https://scikit-learn.org/stable/modules/preprocessing.html

[23] Google and Trifacta. Cloud dataprep by trifacta. [Online]. Available: https://cloud.google.com/dataprep

[24] E. Breck, N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data validation for machine learning," in *Proceedings of the 2nd Conference on Machine Learning and Systems*, 2019.

[25] Bosch. What is bosch iot analytics anomaly detection? [Online]. Available: https://developer.bosch-iot-suite.com/service/analytics/

[26] A. Alcalde-Barros, D. García-Gil, S. García, and F. Herrera, "DPASF: a flink library for streaming data preprocessing," *Big Data Analytics*, vol. 4, no. 1, jun 2019.

[27] Featuretools. What is featuretools? [Online]. Available: https://docs.featuretools.com/en/stable/

[28] Microsoft, *Azure IoT Edge documentation*. [Online]. Available: https://docs.microsoft.com/en-us/azure/opbuildpdf/iot-edge/toc.pdf?branch=live

[29] Amazon. Aws iot greengrass documentation. [Online]. Available: https://docs.aws.amazon.com/greengrass/latest/developerguide/stream-manager.html

[30] Dell. Dell edge gateways for iot. [Online]. Available: https://www.dell.com/en-us/work/shop/gateways-embedded-computing/sf/edge-gateway

[31] M. Johanson, S. Belenki, J. Jalminger, M. Fant, and M. Gjertz, "Big automotive data: Leveraging large volumes of data for knowledge-driven product development," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, oct 2014.

[32] R. Aloufi, H. Haddadi, and D. Boyle, "Emotionless: Privacy-preserving speech analysis for voice assistants."

[33] S. A. Miraftabzadeh, P. Rad, K.-K. R. Choo, and M. Jamshidi, "A privacy-aware architecture at the edge for autonomous real-time identity reidentification in crowds," *IEEE Internet of Things Journal*, vol. 5, no. 4, pp. 2936–2946, aug 2018.

[34] J. Hochstetler, R. Padidela, Q. Chen, Q. Yang, and S. Fu, "Embedded deep learning for vehicular edge computing," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, oct 2018.

[35] I. Kholod, M. Efimova, A. Rukavitsyn, and S. Andrey, "Time series distributed analysis in IoT with ETL and data mining technologies," in *Lecture Notes in Computer Science*. Springer International Publishing, 2017, pp. 97–108.