CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

On-board vehicle data analysis

An evaluation of possibilities using stream processing on an on-board low-powered computer

Master's thesis in Computer Science – Algorithms, Languages and Logic

ERIK NYBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

On-board vehicle data analysis
An evaluation of possibilities using stream processing on an on-board low-powered computer
ERIK NYBERG

Supervisor: Vincenzo Gulisano, Computer Science and Engineering Dept.
Advisor: Ashok Chaitanya Koppisetty, Volvo Car Corporation
Examiner: Marina Papatriantafilou, Computer Science and Engineering Dept.

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Abstract

Automotive vehicles have had an extensive journey in development, from simple mechanical parts in the early days to complex systems built up by several dozens of computers and thousands of sensors. The technical development has increased the security both for people traveling in the car as well as people and animals in close proximity to the traffic. The new technology also allows additional features to be developed, such as vehicle-to-vehicle communication. Nevertheless, the technological development has not only increased the safety and user experience, it has also made the development, testing and verification processes more complex. Because of this, recording and analyzing sensor data to verify that the vehicle is working as intended is a necessary process today. When it comes to analyzing vehicle data a classic approach is to record, store and process data in batches. With this approach the vehicle would collect data over a period of time, eventually transfer the data to some storage facility and later a large batch of data would be processed. With vehicles connected to the internet the data could be streamed to the cloud in real time. This could not only reduce the lead time before an analysis can be made but also enable real time analysis. However, the store-and-process approach come with large demand for storage and compute resources as the data volume increases. The first approach is also not suitable for cases that require real-time results. While the latter approach is suitable for real-time analyses, it requires good cellular coverage and will infer large cost for data transfer. A third option is to run the analysis in the vehicle and only send the result to the cloud. With such approach, each vehicle would run an analysis on its own data and the results could be combined by compute resources in the cloud, performing a very large distributed analysis. This can allow use cases with real-time requirements and will reduce the amount of data that needs to be transferred. However, the computational resources in a vehicle is limited and it is not known how an on-board devices perform when it comes to analyzing data. This thesis focus on the latter approach with a goal of evaluating possibilities and constraints for analyzing data on-board on embedded, resource constrained hardware, using mainstream stream processing engine software.

Keywords: Data-streaming, real-time, data analysis, linear regression, prediction, stochastic gradient descent, phev

Acknowledgements

I would like to thank my advisor at Volvo Car Corporation, Dr. Ashok Chaitanya Koppisetty, for support and feedback of this thesis work. I would also like to thank my academical supervisor Vincenzo Gulisano for advice and support regarding the context of streaming data and academic writing.

Erik Nyberg, Gothenburg June 2018

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Background . . . 1
1.2 Problem statement . . . 1
1.3 Goal . . . 2
1.4 Limitations . . . 3
1.5 Organization . . . 3

2 Use case: Stream processing of high voltage battery data **5**
2.1 Background . . . 5
2.2 Purpose . . . 6

3 Preliminaries **9**
3.1 Modern car networks architecture and data collection . . . 9
3.2 Data streaming . . . 10
3.3 Stream processing engines . . . 11
3.4 Algorithms . . . 12
3.4.1 Correlation coefficient . . . 12
3.4.2 Linear regression . . . 13
3.4.3 Stochastic gradient descent . . . 13

4 Related work **15**
4.1 Edge/fog computing . . . 15
4.2 Stream processing related to vehicles and IoT . . . 16
4.3 Distributed learning . . . 16

5 Architecture/Model **19**
5.1 Assumptions . . . 19
5.2 Pre-process existing test data . . . 19
5.3 Operations and data flow . . . 19

6 Implementation **23**
6.1 Architecture . . . 23
6.2 Producer . . . 24

6.3 Apache Flink 24
6.4 Metrics collection 26

7 Evaluation **27**
7.1 Setup 27
7.1.1 Hardware 27
7.1.2 Software 27
7.1.3 Data set 28
7.2 Measurements 29
7.2.1 Throughput 30
7.2.2 Queue length/Lag 31
7.2.3 Latency 34
7.3 Summary 37

8 Conclusion **39**
8.1 Conclusion 39
8.2 Future work 39

Bibliography **41**

A Practical limitations in connection to using the Kafka system **I**

B Performance of analysis on x64 hardware **V**

List of Figures

5.1 Data flow of implemented use case in an SPE 20

6.1 System architecture 23

7.1 System setup 28
7.2 Example of re-sampling 29
7.3 Throughput 31
7.4 Queue length/Lag 33
7.5 Latency per output rate 35
7.6 Latency per model update 36

A.1 Error rate II
A.2 Window size(s) III

B.1 Performance on x64 hardware VI
B.2 Performance on x64 hardware VII

List of Tables

7.1 Window properties based on their purpose 30
7.2 Linear regression with SGD algorithm settings 30

Glossary

CAN Controller area network. 9, 10

DAG Directed acyclic graph. 10
DOD Depth of Discharge. 6

ECU Electronic control unit. 1, 9, 10

JMX Java Management Extension. 27

LIN Local interconnect network. 9

NTP Network Time Protocol. 27

SOC State of Charge. 6
SOH State of Health. 6
SPE Stream processing engine. 2, 3, 10

1
Introduction

1.1 Background

Automotive vehicles today are more than just mechanical components, they are built up by lots of embedded computers connected on different networks. Ever since the 1970s, the number of electronic systems and computers in vehicles has been steadily increasing. The electronic system can be used to control devices in the car such as the lights, breaking and traction as well as handle multimedia. The large development of the electrical parts have resulted in advanced complex systems in modern cars, such as active safety features with automatic breaking systems, collision avoidance and detecting obstacles, people or animals on the road.

Today, a modern car can consist of more than 2500 sensors and 70 Electronic control unit (ECU)'s [1]. The data that is registered by the sensors can be recorded by using specifically designed diagnostic systems. The sensor data might reveal details about the vehicle status and operations, such as speed and status of vehicle functions, as well as data for the surrounding environment.
Data recorded in vehicles can be used for various cases of analysis. For example, the data can be used for function verification and validation and also to gain deeper knowledge about user behaviour in order to better understand how the consumer is using the vehicle. The data can also be used to further improve the service level, such as predictive maintenance if the data can indicate that a certain part in the vehicle is on the way to fail. A final example requires real-time processing of the vehicle data. If data from vehicles can be analyzed in real time it can be used to analyze the road conditions and possibly alert nearby vehicles. For example, crowd sourced vehicle data could be used to detect traffic jams and alert vehicles going in that direction so they can navigate to other roads. Additionally, in a report regarding connected cars by Pwc[2], they list "the use of connected car data for increase of internal efficiency, quality and product differentiation" as one of the five most likely ways to monetize on connected services.

1.2 Problem statement

A traditional way of handling the sensor data is to record the data when the vehicle is used and then transfer the data to central data warehouse for permanent storage in batches. Doing this enables the possibility to analyze data from a fleet of vehicles, for example to detect drive patterns, how the vehicle performs and behaves and

how it is used by the driver. However, with an increasing number of vehicles that records sensor data, and with more complex vehicles and more sensitive systems to monitor, the amount of data will grow and it will be hard to analyze the data in an efficient manner without using large clusters of computers. According to a white paper published by Hitachi in 2015[3], a connected car generates more than 25GB of data in one hour.

With modern technology, a connected vehicle can stream this data in real-time to central computational resources in the cloud where the data can be processed in a Stream processing engine (SPE). This can reduce the amount of data that needs to be stored persistently and also lead to lower latency for an analysis and conclusions since the data can be analyzed in real-time. However, an effect of this will also be a large increase of wireless data transfers as each vehicle will generate lots of data. The research company Gartner reported in 2015[4] that by 2020 there will be 250 000 000 connected vehicles on the road. With a combination of such large amount of vehicles and Hitachi's reported data size a possible problem with overloaded wireless network is apparent.

This report will evaluate a new approach for connected vehicles. Instead of streaming all the sensor data to the cloud, it will make use of an on-board diagnostic computer to process the data as a stream in the car when the vehicle is in use and generates new data. Only the analysis result need to be transmitted from the car to the cloud. If this is successful the approach can have several positive effects. A lot of drawbacks from batch collecting can be avoided; less data needs to stored, analyses can be made in real-time and give faster result than today. This approach would also counter some negative sides of streaming all data as it would reduce the amount of data that needs to be transferred via a mobile broadband connection. It could also possibly reduce privacy issues as sensitive data does not have to leave the vehicle.

A typical diagnostics computer used on-board an automotive often has a limited capacity compared to regular consumer or server grade computers and cannot provide the same computation power. However, it is desired to be able to run advanced computations, such as machine learning algorithms, in real time when the vehicle is used.

1.3 Goal

The goal of this thesis is to measure and evaluate the performance of a modern stream processing engine running on an on-board diagnostic computer in an automotive vehicle. It is important to measure the performance in order to understand how the system can be used and to which extent. To measure this, a specific use case has been selected and will be implemented as a streaming analysis. The use case includes one of the most common machine learning methods and is described in details in section 2.

1.4 Limitations

This report will only focus on the problem with processing the data in a stream-fashion on the vehicle and the performance of the processing. The report will not focus on the result of the analysis in the chosen use case nor the results possible impact on vehicle design decisions.

In a production implementation of the use case the analysis result would potentially be uploaded to a cloud device, such as a central stream processing framework, to be evaluated on a larger set of data but this is not in the scope of this project. It is also out of the scope of this project to handle the extraction of sensor data from the vehicle as this is expected to come from an already existing system (flight recorder).

1.5 Organization

The organization of the rest of the thesis is the following. The use case that is chosen to be implemented for the basis of the evaluation is described in details in chapter 2. Chapter 3 presents useful background information for the scope of the thesis including a short overview of on-board automotive networks, streaming data and some algorithms. Chapter 4 present existing work that is related to the areas of this thesis. An architectural overview and explanation of the operations in the SPE is presented in chapter 5 and the implementation is described in chapter 6. The evaluation of the implementation is presented in chapter 7, and finally the conclusion and future work is presented in chapter 8.

2

Use case: Stream processing of high voltage battery data

The problem this thesis is facing can be divided into two parts. The main problem is to understand the capabilities and limitations in terms of performance when running streaming data analysis, including machine learning algorithms, on a resource constrained embedded device. One challenge when it comes to evaluating the performance is that the performance depends on the type of workload as the computational capacity and the available memory of the device is quite low. An analysis such as an aggregated count of events, e.g gear changes, is expected to result in a higher throughput than more advanced cases, e.g the ones including machine learning algorithms. The more advanced use cases is expected to push the device to its maximum capacity. However, the advanced use cases is expected to be of more interest than the simpler workloads. Due to this we have selected a use case for implementation that will set a base workload that will be used for the evaluation. The use case represent an expected type of workload that includes a simple machine learning algorithm in order to train a prediction model. In this chapter we will describe the use case in detail and the background of it.

2.1 Background

During the last few years the sales of electric and plugin hybrid electric vehicles has increased. Statistics from the International Energy Agency show a 50% year-to-year growth rate between 2010 and 2015 and a growth rate of 40% for 2016 with a total of 750 thousand sold vehicles [5]. There is a high chance that this trend will continue in the thrive to reduce emissions from vehicles. However, there are demands on electric vehicles regarding driving range and the battery life time but the battery has a limited life time and looses capacity in a degradation process. The battery in electric cars have an estimated life time based on current knowledge about the battery and its degradation. The warranty conditions varies among manufacturers, but there is an average of 8 years and 100 000 miles driven at which the battery should have at least 70% of the original capacity left[6].

The battery lifetime depends on different factors and extensive research has been made to understand the factors[7][8][9][10]. Marano et al. [7] identified factors such as overcharging, consistent operation in high temperature and a charge sustaining mode in low state of charge to have an impact on the life time. Evelina Wikner [8]

identified large aging effects for large depth of discharge, large differences in the aging mechanism of different state of charge levels and that the temperatures importance varies with State of Charge (SOC) and Depth of Discharge (DOD) levels. Smith et al. [10] make an interesting report of the battery life related to geographies and calendar aging and drive cycles.

Some factors, such as operating temperature and charging behaviour can be verified in a lab environment. Other factors, such as user behaviour and real world environment can be harder to verify or anticipate when doing laboratory test.
For example, it may be feasible to verify that different charging behaviours have different impact on the battery State of Health (SOH) in a lab environment but not feasible to simulate or anticipate real world usage.
Because of the long warranties and degradation, it is desired to be able to develop a model based on real world measurements. A reliable model could be used to verify laboratory tests but also give better insights into other factors that have an impact on the SOH.
In order to develop such real world usage model it is necessary to use data collected from vehicles on the road. While it can be made in a classic store-and-process procedure, it is a use case that fits in an on-board analysis scenario.

2.2 Purpose

The purpose of the use case is to train a model that can predict the battery health based on sensor data. For this purpose we have selected a few signals that is known or expected to affect the battery health in some way. For the training we have chosen sensor readings representing the following data:

1. State of Health (SOH)
2. State of Charge (SOC)
3. Ambient temperature.
4. Usable charge energy.
5. Usable discharge energy.

We will use this data to do two operations. The first is to calculate the Pearson correlation coefficient, the second is to train a prediction model using linear regression with stochastic gradient descent. The motivation for the first operation is to get a better understanding about the correlation between the sensors used in the model training and possibly modify the model training algorithm. For example, if two sensors have a very high correlation, maybe one of the sensors can be removed from the model. That could possibly lead to less necessary load when doing the training and prediction without affecting the prediction accuracy. The training part with linear regression should use sensors 2-5 as explanatory variables and sensor 1 as the dependant variable.

In the end it is expected that there is a model in form of a linear equation that can be used to calculate/predict the battery health based on values from sensors 2-5.

These two operations should continuously run over a window of specific time. This will yield multiple models over time and it makes it possible to see if and how the model, and correlations, change over time.

3
Preliminaries

This chapter will explain necessary technical background and techniques relevant for the work of this thesis.

3.1 Modern car networks architecture and data collection

As mentioned in the introduction, vehicles has had an extensive technical evolution and is much more complex today than ever before. The architecture today contain lots of heterogeneous data and devices that transmit data on buses in the vehicle. However, due to limited bus capacity, what data a device want to transmit may not always be what the device is able to transmit.

A vehicle produced today often have multiple different ECUs and additional devices, such as advanced radios and navigation systems, video cameras and radars for safety features and possibly autonomous driving. These are mostly divided into sub-networks or sub-domains with a bus that is adapted for the specific need for those devices[11]. For example, the video cameras and radars can be the source of large amount of data, and they are related to a safety critical feature and requires a large bandwidth on the bus, while features such as turning on the seat heater requires less bandwidth. The sub-networks is usually interconnected with each other via a high capacity backbone bus.

One of the first, and still today the most widely used, network protocols for vehicles is the Controller area network (CAN) protocol. A CAN bus allows any connected node (ECU) to transmit data on the bus if it is not used by any other node. The protocol make sure that messages can be prioritized such that messages with higher priority get access to the bus if two devices transmit at the same time. The technical evolution of the vehicles has lead to increased demands on the communication inside the vehicle. The demands includes, but is not limited to, increased bandwidth to handle more data and increased complexity of the networks with an increasing number of ECUs[11]. As a result of this there are several different network protocols used today on the market and they are often mixed in the vehicles. Some of these include Local interconnect network (LIN) which is a low bandwidth network, low-speed and high-speed CAN, Flexray which is flexible in the configuration and setup. Ethernet for the automotive industry have been developed lately and one implementation of this is BroadR-Reach which can supply a bandwidth of 100Mb/s[12].

The protocol of the CAN bus and the network complexity can make it complicated to record data traffic that occurs in the vehicles and it can also result in very large quantities of data, especially from vehicles with systems for autonomous driving features. The data recorded by a vehicle is registered by sensors on-board the vehicle. This includes data related to the actual vehicle performance (such as vehicle and engine speed), the vehicle position and surrounding environment (such as gps position and ambient temperature) and user interactions and function usage (such as gear shifts and usage of functions such as cruise control, windshield washer or turn signals). Vehicles with advanced features such as automatic breaking or autonomous driving may even record more data such as video or radar feeds.

The sensor data can be recorded by attaching a "flight recorder" to buses on the vehicle. However, the limited bandwidth on the buses can have an impact on the data recording. As the bandwidth is limited it is not possible for all ECUs to transmit data at the same time or with a predetermined frequency. For example, if two ECUs connected to a CAN bus start a transmission at the same there will be a collision and the message with the highest priority is allowed to be transmitted first. The lower priority message need to be resent later when the bus is free. A result of this is that the data transfer is not deterministic, if an ECU should register and transmit values from a specific sensor with a frequency of 10Hz, it may not be transmitted in 10Hz because of lack of bus access due to messages with higher priority.

3.2 Data streaming

Streaming data is an unbounded set of data, i.e a data set of unknown or infinite size and the data is continuously produced. Typical scenarios for this include network monitoring, financial and electronic trading systems, fraud detection[13] and the quite new Internet-of-Things area. Some characteristics that these areas have in common is that they generate large amounts of data and/or the data needs to be processed in real-time. A vehicle can generate large amount of data when it is used and could benefit from the stream processing paradigm. In a streaming context, a data entry is often called a *tuple* and contains data for a specific recording. In the automotive industry such tuples can take various forms depending on implementation, vendor and where in the chain of ECU's and flight recorders the tuple exist. A hypothetical schema for a tuple could contain the following: $tuple = [timestamp, sensor\_id, sensor\_value]$ The rate of the tuples can very depending on the type of sensor as some sensor requires higher rate than others for better granularity in the logging and the bus access.

Data streams can be consumed and processed by an SPE that operates on the tuples as they arrive. SPE's run operations on a flow of data more commonly known as streaming queries and can often be modeled as a Directed acyclic graph (DAG). The queries define the flow of the DAG from the SPE input to various operators and finally to one or several SPE outputs. The operators applies functions to the data,

such as filters, maps, aggregations, collect tuples in windows, joining several stream etc.

- Filter operators evicts tuples from the stream if they do not match a specific criteria.
- Map operations perform a function on each tuple. For example, a map could be used to convert temperature values in Fahrenheit to Celsius.
- Windows can be used to group tuples and operations can be performed on the different groups. Windows may be defined in terms of number of tuples or a time frame. If a window is defined with a time frame different time characteristics can be used; such as processing time or event time.
  As new tuples arrive the window must also move forward, this can be done in different ways, such as tumbling or sliding windows. A sliding window will move forward such that it still overlaps the old window. Take a sliding window with a 5 minute duration and a step size of 1 minute as example. Once the window moves forward it will discard the oldest minutes worth of data, keep the rest and add one minutes worth of new data. A tumbling window will move forward in such a way that it does not overlap with previous windows.
- Aggregations outputs an aggregated value over a set of tuples, such sum, min or max values.

3.3 Stream processing engines

Stream processing engines are frameworks developed to be used for processing data streams. During the last decade, multiple SPEs has been developed and introduced on the market. Some of the latest and today mostly adapted in the domain of stream data processing include Apache Spark[14], Apache Storm[15] and Apache Flink[16][17]. The latter is used in this thesis.

Apache Flink is an open source distributed fault tolerant SPE. It has support for both streaming and batch processing of data and guarantees exactly once processing of tuples. With Flinks terminology in the streaming context programs run in a streaming dataflow. In the dataflow, the incoming data arrives from a source, such as a socket or Apache Kafka, and leaves via a sink, such as a file output or Apache Kafka. Between the source and the sink Flink applies transformations to the tuples, such as a map or filter function. The dataflow model is very similar to a DAG with the exception that certain cycles is allowed. Flink can also operate on the data stream in parallel. When Flink processes a data stream it divides the different operations into *tasks*. A task can be split into parallel instances and processes where each instance will process a subset of the task's data. For example, if Flink is used to analyze streaming vehicle data in the cloud the analysis can be parallelized by splitting the data based on vehicle id. After the split, each subset would contain data for only a subset of vehicles and all data related to one specific vehicle will be located within the subset. Flink allows parallelism to be defined in various level, from the whole execution down to specific operator parallelism settings. When it comes to scheduling and work distribution, Flink is a distributed framework and uses the concept of workers. A worker is a TaskManager which is a JVM process.

A TaskManager uses task slots to limit the number of tasks it can accept. The TaskManager executes one or more subtasks in separate threads.

Apache Flink can process data in windows in two ways; incrementally aggregate when there is new input or buffer all the input and process it when all the data is available. The former approach is used for Reduce (combine two elements from the window) and Fold functions (combine one element in the window with output type). The latter approach is known as a WindowFunction and is the most flexible operation but it also requires more resources as all the data in the window has to be stored in memory. You can also define AggregateFunction to run incremental processing on tuples. The AggregateFunction works in the same manner as a Reduce function but allows more flexible processing. The AggregateFunction has an input type V, an intermediate Accumulator and an output O. As new tuples of type V is received they are added to the accumulator, finally when all data is received the result is extracted from the accumulator as the output of type O. Flink will also need to track the progress of time in order for the time windows to work. If the window is defined using processing time the regular system clock marks the progress of time. If the window is based on event time Flink cannot rely on the system clock as the event time progresses independently of the processing time. For example, in a scenario that receives real-time data with a small delay due to network transfers the event time progresses in the same pace as the processing time but with a small delay. Another scenario might replay historic data, e.g days or weeks worth of event time data, in a matter of minutes in processing time. In order to handle this Flink uses a mechanism called watermarks when handling event time windows. A watermark flows with the data stream inside Flink and holds a time stamp t. The watermark indicates that the event time has reach time t and no more tuples will arrive with a time stamp $t' <= t$. To "close", or finalize, a window Flink uses a Trigger. The default trigger mechanism for event time windows occur when the watermark passes the end of the current window. Once the window is finalized no more data will be added to the window and the final window operations, e.g retrieve the output from an Accumulator or run a WindowFunction, will execute.

3.4 Algorithms

3.4.1 Correlation coefficient

In statistics, correlation is a measure of the linear relationship between two continuous variables[18]. The correlation is measured as a coefficient in the range $(-1, +1)$. A correlation coefficient in the negative range indicates a negative relationship and a value on the positive range indicates a positive relationship. A coefficient value of 0 indicates that no linear relationship exist between the two variables while values closer to the edges of the range indicates stronger relationship. For example, a correlation coefficient value of -1 for the variables X and Y indicates that there is a very strong negative relationship while a value of $+1$ indicates a very strong positive relationship between X and Y.
Pearson's product moment correlation coefficient, possibly more known as just Pear-

son's correlation coefficient, is one method to measure the correlation between two variables. Pearson's correlation coefficient can be used to measure the correlation in both a population and a sample set. The calculation of the correlation coefficient for sample can be written as:

$$r = \frac{n\sum xy - \sum x \sum y}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}} \tag{3.1}$$

where n is the number of samples in the set.

3.4.2 Linear regression

Regression is a method to analyze and model the relationship between variables, with linear regression these relationships is linear in its parameters[19]. More specifically, the method is used to find relationship between a response variable and one or more predictor variables. Response variable are also known as dependent or predicted variables and often denoted as y, predictor variables are also known as independent or explanatory variables and often denoted as x. The approach is used to find a model that can explain the value of y depending on the value of x. With linear regression there are two types. There is a simple linear regression that model the relationship between two variables, the relationship can be expressed as:

$$y = \beta_0 + \beta_1 x + \epsilon \tag{3.2}$$

There is also a multiple linear regression where the dependent variable depends on multiple variables. The model can be expressed as the equation:

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \epsilon \tag{3.3}$$

In both cases, y is the dependent variable, $x_1, x_2, \ldots, x_n$ is the independent variables, $\beta_1, \beta_2, \ldots, \beta_n$ are coefficients for the independent variables and ϵ is a random error.

3.4.3 Stochastic gradient descent

The stochastic gradient descent algorithm is a stochastic approximation of the gradient descent algorithm. Both the gradient and stochastic gradient descent are minimization algorithms and can be used in linear regression to find the optimum coefficients for a model. L´eon Bottou explain both of the algorithms in detail in [20]. Consider a data set where each entry in the set consist of a pair of values (x, y) and we denote the pair z. In each pair, x is an independent variable and y is the dependent variable. To find the optimum solution we use a loss function $\ell(\hat{y}, y)$ to measure the error of predicting $\hat{y}$ when the real value is y. We also choose a set F of functions $f_w(x)$ parameterized with a weight (or coefficient) vector, similar to equation 3.2 above. The purpose is to to find the function $f \in F$ such that the loss $Q(z, w) = \ell(f_w(x), y)$ is minimized.
With gradient descent, we calculate gradient of the loss function over the whole data set for an initial weight vector and update the weights based on the gradient.

$$w_{t+1} = w_t - \gamma \frac{1}{n} \sum_{i=1}^{n} \bigtriangledown_w Q(z_i, w_i) \tag{3.4}$$

In this formula, γ is a chosen step size (or gain). The step size is set to take steps further based on the gradient and the cost of the loss function. Usually this requires multiple loops over the data set until algorithm achieves linear convergence.

Stochastic gradient descent is a simplification where the weights are updated more often. Each iteration calculate the new weights based on the gradient for a single randomly picked sample z_t of the data set.

$$w_{t+1} = w_t - \gamma_t \bigtriangledown_w Q(z_t, w_t) \tag{3.5}$$

4
Related work

This section present work related to the scope of the thesis. The thesis is touching several areas, such as stream processing on-board devices, edge/fog computing and distributed learning techniques. To show the importance of the subject of the thesis we will present work related to these topics in the following subsections.

4.1 Edge/fog computing

The terms edge and fog computing provides a service similar to cloud computing but instead of a central point in the cloud the computation take place on devices at the edges of the network, similar to the system developed during this thesis, possibly with the intent of performing a federated learning.

Bonomi et al. [21] presents the role of fog computing in the Internet of Things domain. They identify several examples related to this thesis, such as location awareness and low latency, geographical distribution, large number of nodes, federated services and on-line analytic. They specifically identify connected vehicles as one scenario of interest with potential services in infotainment, safety and traffic support. In the report they also bring up the aspect of the fog compared to the cloud; the fog provide localization and low latency while the cloud provides global centralization.

Yi et al. [22] identifies application scenarios for fog computing, one being mobile big data analytics. They believe that fog processing will be a key technique when it comes to analytics of large volumes of data originating from Internet of Things. Additionally, they identify a scenario where the fog and the cloud can be combined in a federated system where fog nodes process local data for emergency cases, while the cloud is responsible for tasks that require more computational resources.

L. M. Vaquero and L. Rodero-Merino proposes a definition of fog computing including challenges that fog computing need to overcome [23]. Specifically, they identify compute/storage capacity as one challenge, which highly depends on the fact that the cost of the devices is a major factor. This is very related to the scope of this thesis as there is a need to perform computationally heavy analysis on-board, but the on-board devices are to be kept at a minimum capacity to reduce the cost and size of the device.

4.2 Stream processing related to vehicles and IoT

Rho et al. [24] presents a framework for performing distributed stream processing on-board vehicles with the purpose of reducing the software complexity and development cost while the amount of data increases. The framework make use of a mixed setup of ECUs with single and multi-core processors and distribute the workload on the available cores. The evaluation of the framework shows that the end-to-end execution time was lowered in the distributed multi-core setup compared to one single core with tuples ranging from 4KB to 1MB.

Krishnaswamy et al. presents a system named SAWUR (Situation awareness with ubiquitous data monitor for road safety)[25]. The system is intended to run on-board a vehicle and detect potential dangerous situations in real time. The presented system uses existing crash related data to train models for predicting dangerous events. There is also an on-board system that uses context information, such as driver behaviour and road condition, and the trained prediction model to classify situations on the road and propose counter measures. Additionally, upon a a dangerous event, the data is recorded and fed back into the database of crash related data to strengthen the prediction model.

Shukla et al. [26] proposes a benchmarking suit for distributed stream processing of Internet-of-Things applications. The benchmark suite proposed XML parsing, various filters, statistical analysis and aggregation tasks and preditive analysis. The latter includes linear regression, multi-variate linear regression and decision tree classification. The linear regression trains online while the multi-variate linear regression and decision tree classifications are trained offline but used online for prediction. While the report do evaluate the benchmark suite on powerful virtual machines hosted on Microsoft Azure, the evaluation results show the importance of evaluating the performance of a stream processing task. The evaluation result yields large deviations among the different tasks, especially the peak throughput and end-to-end latency.

4.3 Distributed learning

One of the benefits of moving the analysis of vehicle data from centralized locations, such as the cloud, to the vehicles is the possibility of running learning algorithms distributed. Training a prediction model in the cloud would involve large computational cost and large sets of data, yielding one or more models depending on if and how the data set has been divided into smaller parts. Training such model on-board a vehicle will yield a model specific for that vehicle. The models can be combined together to one model representing a fleet of vehicles using federated learning.

Federated learning uses a server and worker devices. For example, the server can maintain a global model and the worker devices, in the scope of this thesis it would be a vehicle, have access to training data. The workers uses the training data to

compute and updated model and send the new model back to the server. Ulm et al.[27] presents a Federated Learning framework in Erlang. They present a fully realized framework using federated learning focusing on artificial neural networks. Zinkevich et al. [28] presents an algorithm for parallelized stochastic descent. They present an algorithm that uses k machines and m tuples, each machine is given m/k tuples and all k machines operate on the data in parallel with the regular stochastic gradient descent algorithm and combines the k results to an aggregated final result.

5
Architecture/Model

This chapter describes the proposed architecture for the analysis part of the chosen use case and describes operations that needs to be made on the data.

5.1 Assumptions

As described in section 3.2 the schema of tuples for a data stream of vehicle sensor readings can depend on various aspects. Because of this, the following assumption was made at the start of this thesis to layout the ground work:

1. The SPE will only receive the values from sensors it will use for the specific task.
2. The equipment used to output tuples (such as a flight recorder) can emit one tuple containing a time stamp and all sensor readings.

As mentioned in section 3.1, the timings of sensor readings is not deterministic, so a direct result of this and assumption 2 in the list above is that the log equipment (flight recorder) is expected to be able to hold-and-save known sensor values until the next tuple is sent. For example, if the flight recorder is setup to transmit a tuple every 5 seconds to an SPE, it should be able to keep any sensor value that is recorded before the next tuple is scheduled to be sent. If sensor values from multiple sensors is recorded they should be transmitted within the same tuple.

5.2 Pre-process existing test data

The data used for this thesis consist of existing log data from a real vehicle. The data is in a binary format that contain a time stamp and a value for each recording and sensor. The data set will be parsed and data for the relevant sensors is extracted with time stamp and sensor value for each recording. Very few sensor recordings will contain exactly the same time stamp due to different sampling rates and the non deterministic behaviour of the CAN bus. Therefore data will be re sampled in order to get all sensor recordings with a unified time vector to match assumption number 2 above.

5.3 Operations and data flow

As the data enter the SPE it is expected to arrive in tuples matching the assumptions mentioned above. The tuples should contain a time stamp representing the time of

the sensor recording (event time) and all sensors that have a recording from that time. Example:
Tuple containing all data:

$$tuple = [ts, sensor_1, sensor_2, \ldots, sensor_n]$$

Tuple missing $sensor_1$:

$$tuple = [ts, sensor_2, \ldots, sensor_n]$$

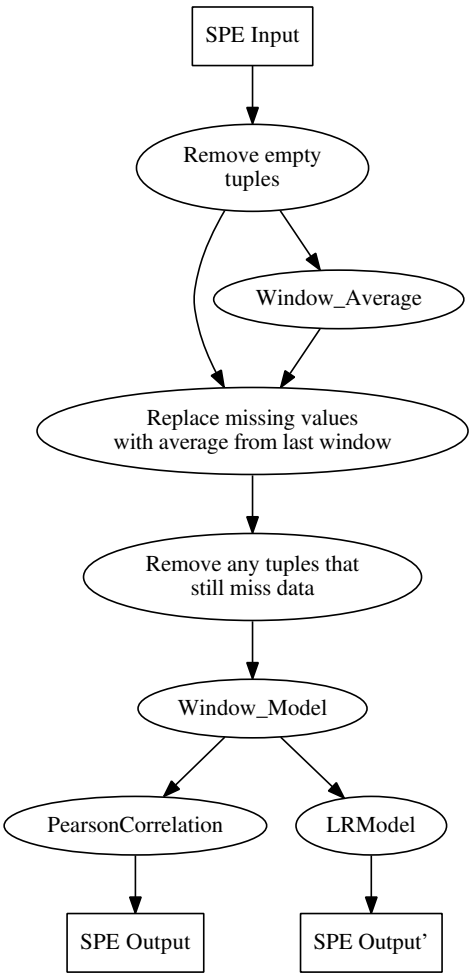


Figure 5.1: Data flow of implemented use case in an SPE

The complete flow of data and operators in the SPE can be seen in the figure 5.1, the details of each operation is described below.

1. **Remove empty tuples**
   The very first operator is a filter operator. It has one input and one output. The purpose of this filter is to discard all tuples that does not contain any sensor data readings at all. The system is supposed to handle sensor data that is missing one or a few sensor values and in such case interpolate a replacement value later, but if a completely empty tuple is discarded.

2. **Average value aggregation**
   In order to substitute missing values for tuples the output from the previous operator is used as input to a time window. The window will collect tuples over a fixed length of time and continuously slide the window forward. An aggregation operation is performed on the window. The aggregation calculates the average value for each sensor. The output from the aggregation is one single tuple that contains the average for each sensor.

3. **Replace missing values**
   The output from the previous operator is used to replace any missing values in the data stream. A map operation is performed on every tuple in the data stream and if any sensor value is missing from the tuple it is replaced with the average value from the previous calculation.

4. **Remove any tuples that is still missing data**
   This operation might seem redundant but it exist to avoid a scenario where the whole average window operation was missing values for a sensor. In such case the step of replacing missing values with the average would not succeed and the sensor value would still be missing. In that scenario it is not desired to have the tuple in the data set used for correlation coefficient calculation and training the linear model, because of this the tuple is discarded at this stage. All tuple outputs from this operator will contain a time stamp and values for all sensor as defined in section 2.2.

5. **Correlation coefficients calculation**
   The output tuples from the above filter is consumed by large sliding window of tuples. The content of this window is used to calculate the Pearson correlation coefficient between all pairs of sensor $(sensor_i, sensor_{j!=i})$ by calculating the sum of each sensor as well as each pair of sensors and then use the sums to calculate the correlation according to equation 3.1.

6. **Linear regression model training using stochastic gradient descent**
   The final operation performed is made using the same sliding window as the correlation coefficient mentioned above. The data in the window is used to train a linear regression model for the battery health prediction using stochastic gradient descent. For each new window when the time slides forward, the initial coefficients for the model is reset to zero and does not depend on the previous model. The reason for this is to not let older data influence the model. In each window we need to do multiple iterations over the data set and is due to this forced to buffer all the tuples in memory. For each iteration we shuffle the data before looping over each tuple in the data set to run the SGD algorithm as is defined in section 3.4.3. Once all iterations over the data set has been performed the result is sent to an output sink.

6
Implementation

This section presents the implementation of the data flow in the linear regression training as well as the platform tools used and the data needed for evaluating the performance.

6.1 Architecture

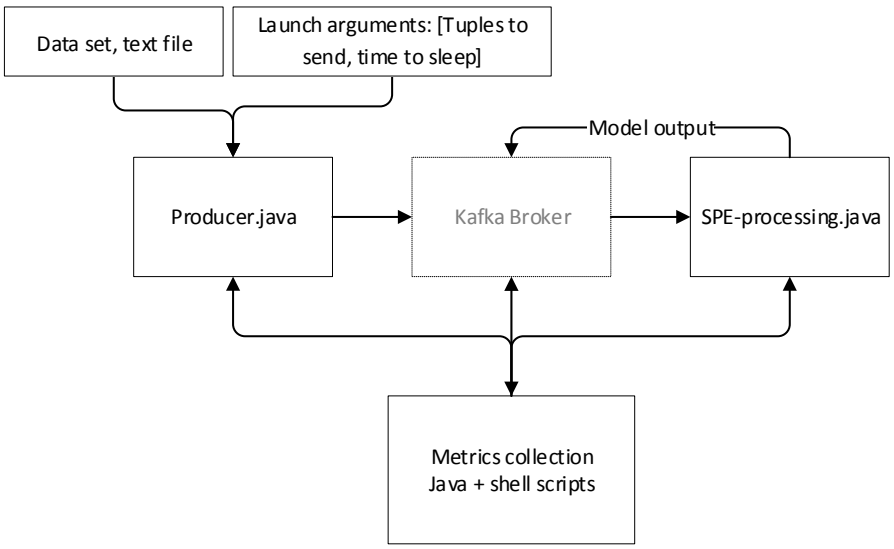


Figure 6.1: System architecture

The architecture of the implemented system if made up of three main parts; i) a producer that transmit a stream of tuples, ii) an SPE process that consumes tuple and perform various operations on the tuples as mentioned in the previous section and iii) a data broker in the middle between i and ii. Additionally there are a few more processes that collects metrics from the producer, SPE and the data broker to evaluate the performance.

The stream processing engine used for the developed system is Apache Flink and the data broker is Apache Kafka. Apache Kafka is a message passing system that can handle large amounts of data and is scalable and distributed. It might seem unnecessary complicated to run a scalable distributed message broker when the

processing takes places on-board a vehicle with on-board generated data. However, Apache Kafka is a good fit for this evaluation as there are good capabilities to monitor the traffic and performance metrics. Apache Flink also have built-in support to consume data from Kafka with automatic input rate limiting and relies on Kafka for its fault tolerance.

6.2 Producer

The producer is a java process that make use of Kafka's Producer API to send the tuples to the broker. The producer reads a text file that contain the data set to use. Every row in the data set file corresponds to one tuple and should contain a time stamp and the sensor data values. A row may contain only a time stamp if no data at all existed at that time and it may be missing values for some sensors if not all sensors existed at that time. In the first case it should be filtered in the SPE by a filter operator, in the second case any missing value should be replaced with an aggregated average value.
The producer process is executed with two start arguments; the number of tuples to send and a time to sleep. The producer will output the specified amount of tuples to the broker in one batch and then sleep for the specified amount of time, this is continued until all tuples has been sent. The two arguments can be used to limit the output rate, e.g. 700 tuples per second.
As the producer is using the Kafka Producer API the properties of the producer behaviour can be set in different ways.The producer is configured to require acknowledgement by the broker for every sent tuple and to re-send any un-acknowledged tuple a maximum of 10 times.
Besides the data present in each row in the data file, each tuple will also be given one more time stamp. As the tuple is sent to the broker the producer adds the current system time stamp to the tuple.

6.3 Apache Flink

Once the data flow presented in section 5.3 is invoked, Flink will start to consume tuples from the Kafka broker and process the tuples in the chain of operators. The implementation uses Flinks embedded Kafka consumer FlinkKafkaConsumer010. The consumer implementation will automatically adjust the tuple consumption rate, it will try to consume any tuples available from the Kafka Broker unless there is a back pressure problem somewhere among the operators in the SPE. If there is an operator in the system that cannot keep up with the rate of incoming tuples there will be a back pressure and operators earlier in the data flow will reduce the output rate which eventually can cause the Kafka consumer to reduce the consumption rate.

As the tuples is entering Flinks chain of operators via the Kafka consumer, the time stamp from each tuples is used to identify the original event time of the data in the tuple.

Next the tuples will flow through the operators presented in section 5.3.

In the Flink process, we would naturally want to normalize all the sensor data so that they all use the same scale. However, as the data is streaming and represent an infinite set of data, normalization is not straight forward. In a batch process the normalization can be made with the use of the max and min values of each sensor, but in the streaming context we don't always know the maximum or minimum value. The values could be normalized with domain knowledge about the sensor measurements, i.e the max and min of each of the selected sensors. The four sensors mentioned in section 2 that is used as features for the training uses different scales and does not fit well within the prediction model without normalization. Because of this we use an alternative approach known as feature engineering. We apply different calculations to the different sensors to give the signals a more common scale. This modification take place after the tuple has been consumed by Flink and the time stamp extracted, but before the tuple progresses further along the operators in the SPE.

The calculation of the average value is based on a sliding time window for grouping the tuples in the most recent time. For calculating the value we use an AggregateFunction that uses an accumulator to incrementally keep track of the sum of each sensor values and the count of tuples. Once all tuples has arrived and the window is finalized the average value is extracted from the accumulator.

The correlation calculation works in the same way as the average calculation, data is collected in a sliding window and an AggregateFunction is used to process the incoming data. As tuples is added to the window the AggregateFunction will incrementally add the sensor values to an accumulator. The accumulator keep track of the sum of each sensor, the sum of each pair of sensors $(sensor_i, sensor_{j!=i})$ and the total number of tuples that is received. Once the window is finalized and no more tuples will be added to the current window the output is extracted from the accumulator. The output is extracted by taking the sums for each pair of sensor and calculate the correlation according to equation 3.1.

The linear regression with SGD algorithm uses the same sliding window as the correlation calculation. However, with this algorithm we need to do multiple iterations over the whole data set and cannot use an AggregateFunction. Instead we will use an AllWindowFunction to buffer all the data in memory and iterate over the tuples when the window has been finalized. When the window has been finalized and all data is present the training will start. The training starts by setting an initial state for the weight vector to 0 and followed by multiple iteration over the set of tuples. For each iteration over the data set the data is first shuffled randomly before using the SGD algorithm as is defined in section 3.4.3. Once all iterations over the data set is finished the trained model properties and additional data is sent back to the Kafka broker. The additional data contain information such as the number of tuples in the window, the number of weight updates, average error among all updates, computation time and latency.

All time windows is using event time with the default Trigger mechanism to finalize windows. It also uses Flinks pre-defined watermark emittor AscendingTimestampExtractor that requires all time stamps to be in ascending order (e.g we will not have out of order tuples).

6.4 Metrics collection

To evaluate the performance of implementation several performance metrics is collected from the different processes to see how the the system performs with increasing load. The interesting metrics are the tuple throughput, the queue length and the latency for training the model.

- Throughput:
  The throughput is measured by collecting the producer output rate and the Flink Kafka consumer input rate. Both of these metrics are exposed as JMX entries from each process and are continuously updated by each process as the execution takes place.
- Queue length:
  The queue length is measured by checking the state at the Kafka broker. The broker keep track of the current offset for tuples and the consumer offset. The current offset increment with one for every new tuple the broker receives from the producer and is basically a counter of the total number of tuples received by the broker. The consumer offset indicates the number of tuples consumed by the consumer. The queue length, or lag, is the number of tuples that the consumer lags behind the current offset. An increasing lag indicates that the consumer have problems keeping up with the rate of produced tuples.
- Latency:
  The latency metrics is calculated when the model training is done and embedded in the model output to the Kafka broker. The latency in this case is the time between the finished fully trained model and the sent time stamp of the last tuple in the window.

The metrics for throughput and queue length are collected and saved in text files continuously by checking the status with even interval during each run. The latency is collected by consuming the model data output from the Kafka broker and retrieving the latency property from each message.

7
Evaluation

This section presents the evaluation of the proposed method. Section 7.1 present the setup of the evaluation environment with hardware and software resource as well as the data set used. Section 7.2 present the summary of the metrics collected for various different output rates.

7.1 Setup

7.1.1 Hardware

The resources used for this system is one laptop and two Raspberry Pi 3. These are connected with a Netgear GS105 network switch equipped with 1Gbps ethernet connections, the setup is illustrated in figure 7.1. The laptop is equipped with an Intel Core i7 processor and 8 GB of ram. The Raspberry Pies are equipped with an ARM Cortex A53 4 core processor running at 1200MHz, 1 GB of ram and 100Mbps ethernet connection.

While a Raspberry Pi is not used within vehicles the hardware of the Raspberry Pi is a valid representative to certain on-board computers, such as those used for flightloggers.

7.1.2 Software

Both Raspberry Pi devices is usgin Raspbian Strech as base operating system. For the SPE setup one Raspberry Pi is used where Apache Flink 1.3.2 is installed. Flink is configured to run in a standalone mode with one worker. The worker is given half of the available memory on the device, 512MB, and two task slots. As for parallelism in the workflow, some operations (some map and filter operations) is set to a parallelism of 2, while the window functions with correlation calculation and model training is set to a parallelism of 1 (no parallel tasks).
The laptop is running Xubuntu 16.04.3 LTS and is acting as the data producer, performance metrics collector and Network Time Protocol (NTP) server for clock synchronization with the device running Apache Flink. The performance metrics are collected from both the producer, the consumer and the Kafka broker. The data from the producer and consumer is collected via Java Management Extension (JMX) with 5 second intervals, the queue length is collected from the broker with 5

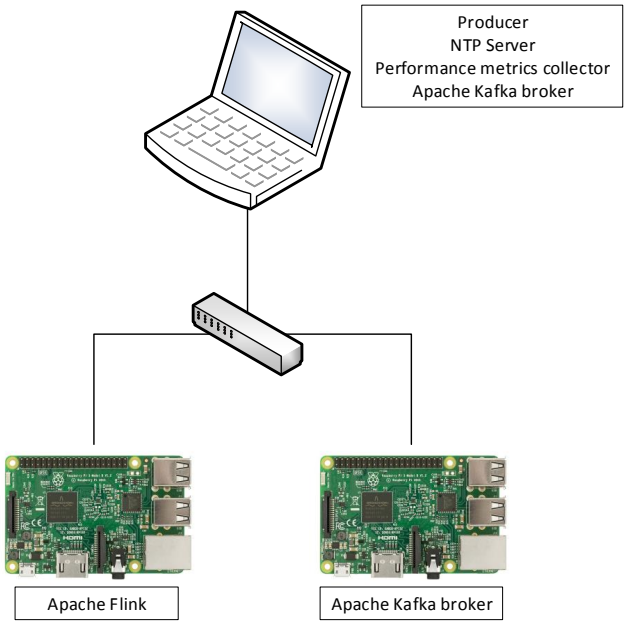


Figure 7.1: System setup

second interval and the model latency is collected by consuming the model output data from the broker. The clock synchronization is necessary for the latency calculation to be correct.

As for the data broker, we decided to run the evaluation on two different setups. One where the data broker is running on the second Raspberry Pi and one where the data broker is running on the laptop. The two setups will be referenced as system 1 and system 2 in the remainder. The evaluation process is performed against both setups separately. With this dual setup, we can easily distinguish the bottleneck upon system saturation. During the execution we noted a performance issues with Kafka on the Raspberry Pi, see appendix section A for details. Note that in both setups the SPE Apache Flink is running on a separate Raspberry Pi.

7.1.3 Data set

For this evaluation, a small subset of existing log data has been selected. The log data comes from a plugin hybrid vehicle of model year 2013 and the data was collected between 2014-05-08 and 2014-05-28. As mentioned in section 5.2, the data needs to be pre-processed to join the different sensors on a common time vector. The data is pre-processed by parsing the whole set of files, extracting the data and time vectors of the six interesting sensors mentioned in section 2.2 and joining all the data into one large set. The set is then re-sampled to a time vector with a sampling frequency of 1 second and written to a text file. The re-sampling method will create a new time vector ranging from the start of the first file to the end of the last file, with the time stamps corresponding to real world time. No sensor data is interpolated or in any way generated where the source files were missing data, sensor

values exist only where they exist in the source data with a slight modification to the time stamps in order to match the new time vector. However, the re-sampling process will add empty tuples in some places, e.g where one measurement file ends and there is time gap before the next file take place. These empty tuples correspond to times when the vehicle was not used and is expected when modeling time series data for a longer time period as vehicles is in parked mode most of the time. When the re-sampling method has been applied the complete data set consist of $\sim$ 1.7 million tuples, out of which $\sim$ 400 000 tuples contain at least one sensor value.

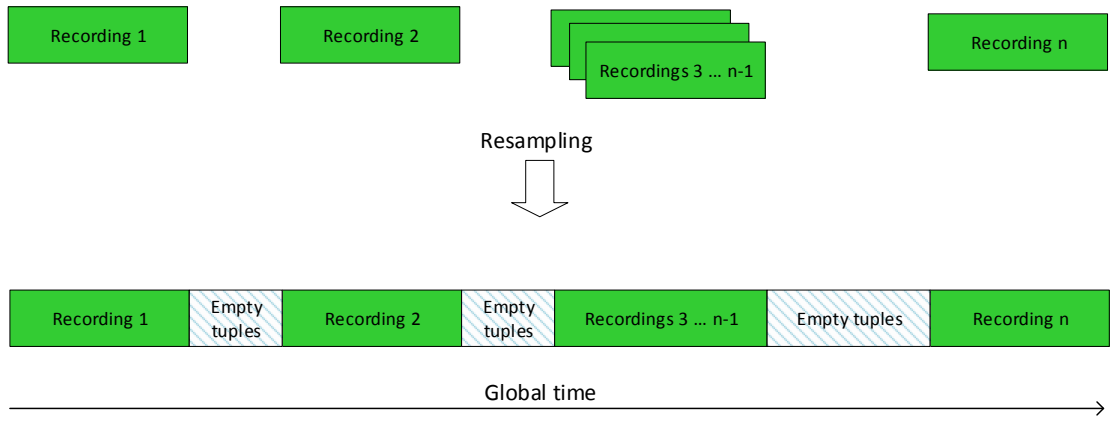


Figure 7.2: Example of re-sampling, result contains empty tuples between original recording tuples

7.2 Measurements

For the evaluation we executed the process with 10 different output rates for system 1 and 11 different rates for system 2. Among the executions several different output rates were used, ranging from 500 to 2000 tuplers per second. It's important to note that the output rate is the number of tuples sent by the producer each second but the event time of the tuples is not affected. No matter the output rate, the tuples is still sampled with a 1 second frequency. When the output rate is 500 tuples per second, the data leaving the producer during 1 second corresponds to 500 seconds of measurement data. Similar when the rate is 2000, 1 second of sent data corresponds to 2000 seconds of measurements. As the output rate increases, the historic data will replay faster and force the event time windows in Flink to be finalized faster.

The properties used for the evaluation is illustrated in table 7.1 and 7.2. The former show the sizes and the types of the windows used in Flink and the latter show the properties specific for the linear regression algorithm.

Window purpose	Type	Duration	Step size
Average value	Sliding window	1 hours	5 minutes
Correlation coefficient	Sliding window	3 days	8 hours
LR Model	Sliding window	3 days	8 hours

Table 7.1: Window properties based on their purpose

Property	Value
Iterations/epoch	30
Learning rate	0.0001

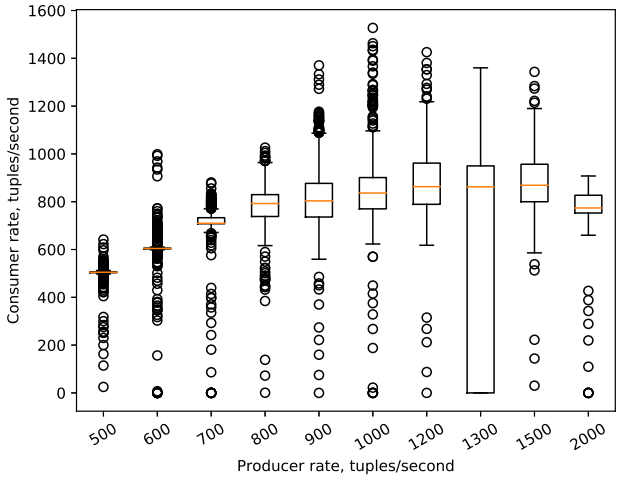
Table 7.2: Linear regression with SGD algorithm settings

7.2.1 Throughput

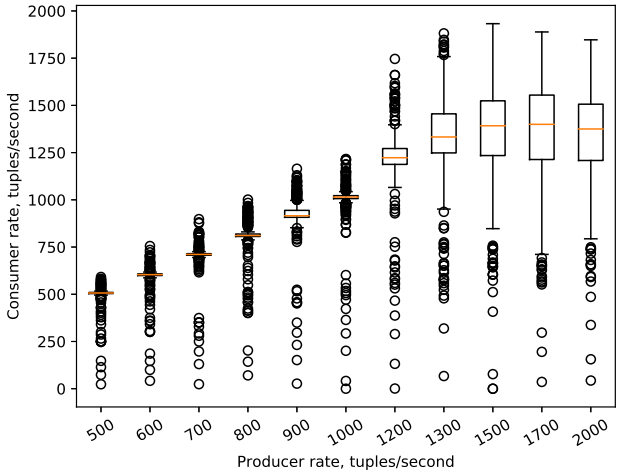
The throughput measures Apache Flinks performance of consuming tuples as the output rate increases. A higher input rate will fill up Flinks time windows faster and train a model more often. If Flink cannot keep up with the rate of the tuples the consumption rate will be adjusted by Flink automatically and reflected in this measurement. The throughput for both systems is illustrated in figure 7.3. The X-axis labels in the figures is the targeted output rate and the Y-axis is the reported consumption rate by Apache Flink.

Figure 7.3a show the throughput for system 1. Flink keeps up with the output rate up until 800 tuples per second where the system appears to be saturated as a constant consumption rate of $\sim$ 800 tuples is seen.
The metrics from system 2 is seen in figure 7.3b. Here we notice a higher consumption rate as Flink manages to keep up with the output rate until a rate of 1300 tuples per second. Once the output rate is greater than 1300 Flink will keep a pretty constant consumption with a median of $\sim$ 1300 tuples per second.

The lower consumption rate with system 1 is an effect of the mentioned performance problem with system 1. As noted above, the X-axis is the target output rate, i.e what the producer is expected to send. With system 1, the producer did not manage to reach the target when it was set to a higher value than 800 tuples per second. At higher rates the Kafka broker would not accept all sent tuples, leading eventually to lost tuples and a low *real successfull* output rate. For more details about the performance problem see section A.



(a) System 1



(b) System 2

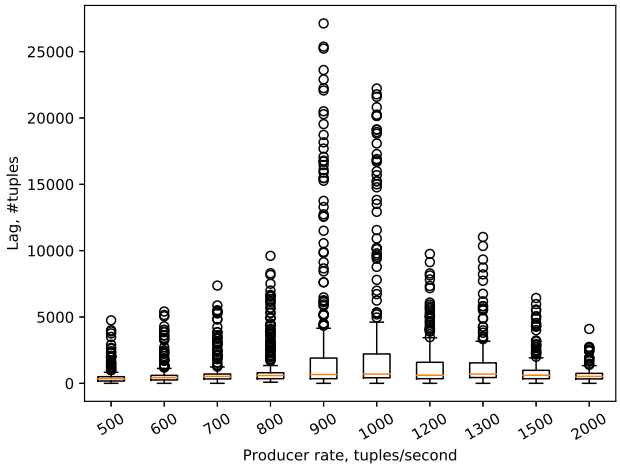
Figure 7.3: Throughput

7.2.2 Queue length/Lag

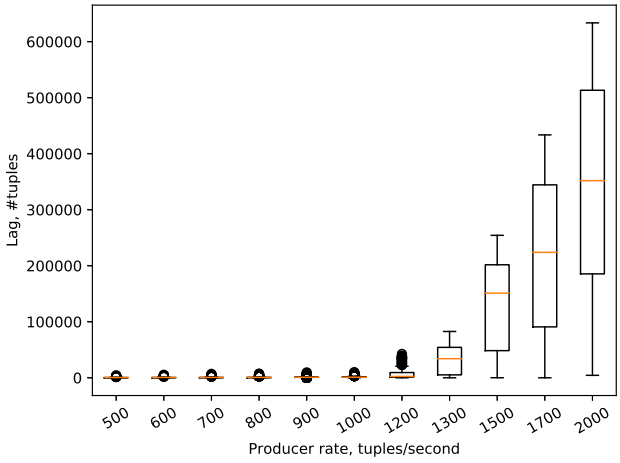
If Flink have problems consuming tuples in the same rate as the output rate it would reflect in the queue length. The queue length, also known as lag, is a metric representing the gap of tuples between the consumer and the producer. The lag should preferably be close to 0 as that would mean that Flink consumes tuples right after the producer sent them to the broker, ideal for real-time applications. However, if Flink cannot consume tuples in the same rate as the output rate from the producer, the queue length should be increasing. Similar to the throughput plot the X-axis represent the target output rate. The Y-axis is the lag value reported by the Kafka data broker.

Figure 7.4a show the queue length for system 1. It starts with a median lag of 350 tuples when the rate is 500 and grows to a median of 660 when the rate is set to 900. From the rate of 900 and forward the lag is kept quite constant with a median

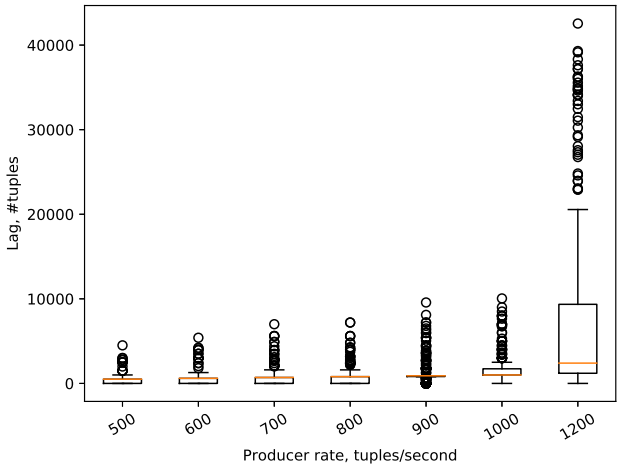
ranging between 600 and 700 tuples. Nonetheless, as explain in previous sections, the setup suffered from performance problems and the target rate was not achieved for rates exceeding 800 tuples per second. As seen in figure 7.3a, the consumption rate is almost constant once the target output is 800 or above which explains the consistent value for the lag.
In figure 7.4b we see the lag for system 2. With the broker on the laptop there is no problem achieving the target output rate from the producer which yields different results for the lag. For the rates between 500-1000 the median lag matches the rate, e.g when the rate is 500 the median lag is 500. When the rate is set to 1200 the median lag grows to 2400 tuples and continues to grow even more as the rate increases. The system seems to be saturated when the rate is $\sim 1000 - 1200$ tuples per second as the turning point for the lag is in that region. Figure 7.4c show the lag of the rates up until saturation, removing all rates that exceeds the turning point.



(a) System 1



(b) System 2



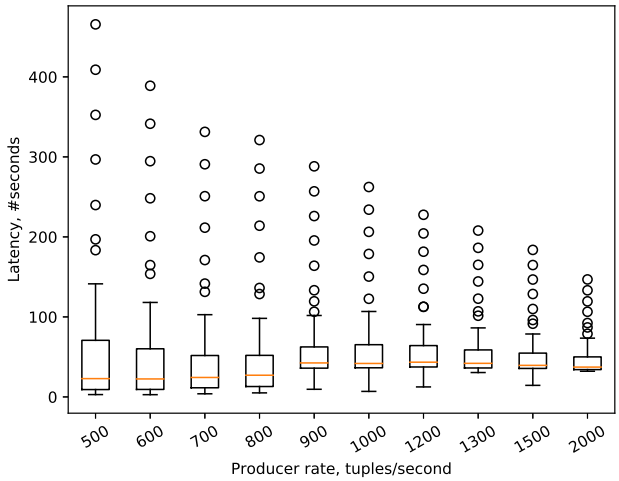
(c) System 2, output rate until 1200

Figure 7.4: Queue length/Lag

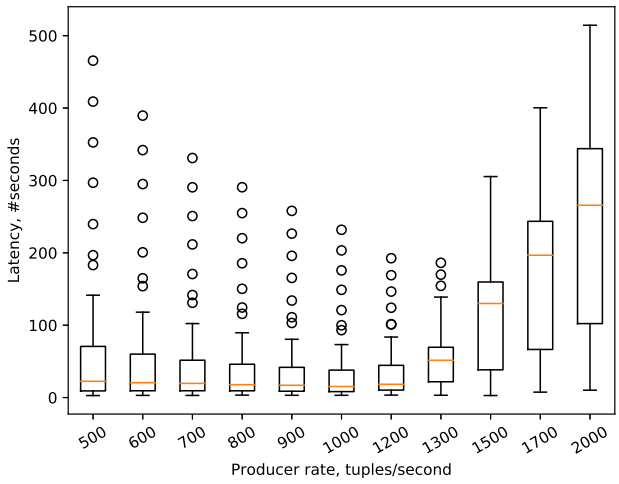
7.2.3 Latency

For the latency metric we use the total end-to-end processing time. Let's define two time stamps; i) ts_{model_done} and ii) ts_{sent}. The first is the current system time of the SPE host when the current model M_c has finished training. The second is extracted from one tuple in the current window and represent the time the tuple was sent from the producer. If W_c is the current window used for training model M_c and the window contains n tuples denoted $\{t_0, t_1, \ldots, t_n\}$ sorted by event time. The time stamp ts_{sent} is extracted from the last tuple of the window, t_n. We define the latency as $latency = ts_{model\_done} - ts_{sent}$. This represent the total time from sending the tuple from the producer to utilizing the tuple for model training.

The figure 7.5 show the measured latency for both systems. As in the two earlier sections, the X-axis represent the target output rate which for system 1 is only achieved for rates until 800, for system 2 the target output rate is always achieved. In figure 7.5a we see the latency for system 1. For target rates 500 to 800 there is a median latency of 20 to $\sim$ 27. When the target rate exceeds 800 the median latency increases to $\sim$ 40 seconds. We can also notice that the latency distribution, both with regards of first and third quartile but also the extreme values, reduces as the rate increases.
The latency for system 2 is illustrated in figure 7.5b. Here we notice a similar behaviour for the first half of the rates. With an output rate of 500 the median latency is $\sim$ 22 seconds, the latency reduces as the rate increases to median of $\sim$ 15 seconds when the rate is 1000. As the rate increases to 1200 we see a turning point and the median latency starts to increase as the rate grows. We also notice the same behaviour with reduced range of distribution until the turning point where the distribution grows instead.



(a) System 1

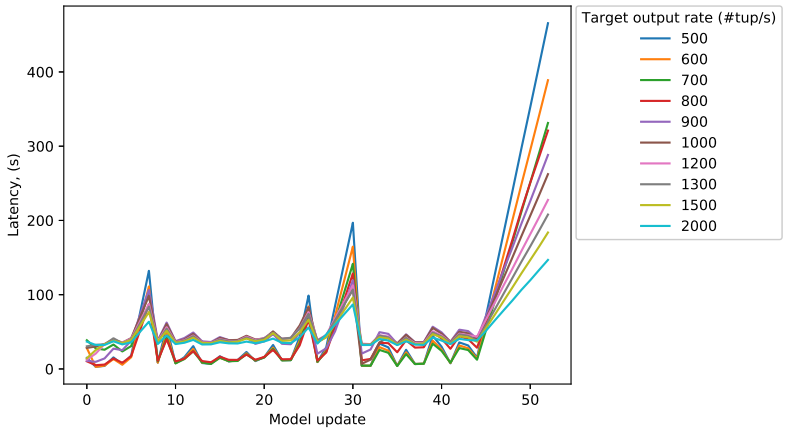


(b) System 2

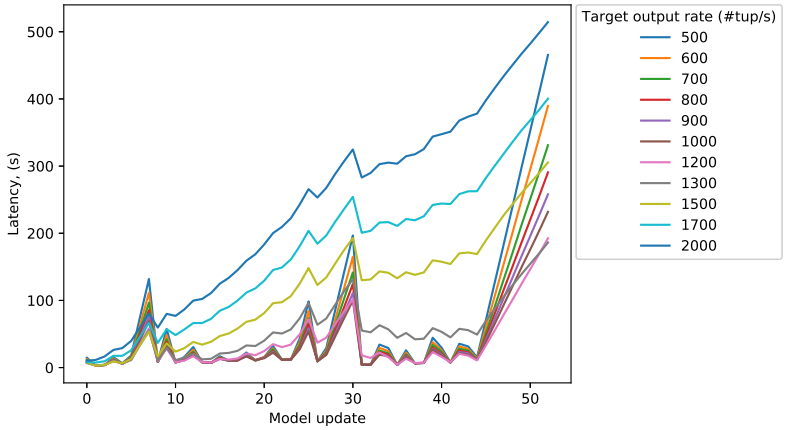
Figure 7.5: Latency per output rate

While the latency reduces as the output rate increases might seem odd, it gets clearer when looking at the latency for each model update. The latency for each individual model update and output rate is shown in figure 7.6a and 7.6b for system 1 respectively system 2. The X-axis of the graphs represent each new model that is trained, the Y-axis represent the latency for each update.
With system 1 we notice two distinct groups with overall different latency, the executions with rates exceeding 800 result yields an overall higher latency as we noticed in the figure 7.5. Nonetheless, for both groups we notice a few spikes in latency and a continuous increase at model update 44 and forward. For both the spikes and the increase at the end, the lower latency's yields the largest latency values.
Looking at system 2 we notice similar spikes in the same locations as system 1, but at larger rates we notice a different behaviour. First, we notice that for rates exceeding 1500 the latency increases for every new model that is trained, indicating that the Raspberry Pi with Flink is really saturated. The same behaviour can be

seen for the cases with a rate of 1200 and 1300, but it starts slightly later and yields a smaller increase. For rates of 1000 and below the latency per update is almost the same among all rates. We also see the spikes and large increase at model 44 as with system 1.



(a) System 1



(b) System 2

Figure 7.6: Latency per model update

7.3 Summary

With the results from these measurements, we have showed that a low powered device such as the Raspberry Pi can be used to continuously train a linear regression model on streaming data. Among the targeted output rates of up to 2000 tuples per second, the highest median for the input rate was noted at 1300 tuples per second, while a consumption of 1200 tuples per second was possible without causing any major increase in latency or lag. The optimal rate which the system seams to be able to handle is between 1000 and 1200 tuples per second as the turning point for lag and latency is in that region.

As for the latency part, we did notice some spikes for certain model computations, visible in figure 7.6. These spikes can possibly be explained by Apache Flinks mechanism of triggering windows for computation. As mentioned in section 3.3 an event window is finalized when there is a watermark with a time stamp t that does not belong to the current window. If there are large gaps in the stream of data, e.g there are large chunks of tuples that does not contain any data and will be removed in the empty-tuple filter, we might miss watermarks that indicates that the window should be finalized. The figure A.2b show the size of each window, and the moment of the latency spikes seem to correlate with the moments that the window size decrease. Apache Flinks documentation states that the watermarks flows within the data stream but it is not clear what happens when the tuples is filtered out. However, the fact that the spikes is reduced when the output rate increases (effectively sending the large chunks of empty tuples faster) and that the spikes seem to correlate with occasions when the data set size reduces, could be an indication that the spikes happen because of this. Nonetheless, this might need to be investigated further.

As the linear regression algorithm runs 30 iterations over each data set and updates the weights for all four features for every tuple we can assume that this part takes up most of the computation resources and time on the low-powered device. Simpler SPE work flows,such as simple aggregations and filters, should be able to handle larger input rates without any problems.

Regarding the dual system setup with the data broker, we noticed that the capacity of the SPE increased by circa 50% with system 2 compared to system 1. With a setup where the data broker is running on a Raspberry Pi and we use Apache Flink with the implemented work of this thesis, the major bottle neck is the data broker. Once the data broker is moved to a more powerful device the bottle neck will instead be the workload in Apache Flink on the Raspberry Pi.

8
Conclusion

8.1 Conclusion

The goal set for this thesis was to measure the performance of a modern stream processing engine on-board a diagnostic computer in a vehicle. While the performance varies depending on algorithm complexity, the data type and size (such as window size combined with input rate) among others, we have shown the performance when using linear regression for a prediction model of the hybrid high voltage battery health.

With the implementation made in this thesis it shows that a cheap low powered device equivalent to a diagnostic computer can process streaming data and run machine learning in form of linear regression with stochastic gradient descent on it. With a data set consisting of 1 tuple per second the system was able to handle an input rate of 1000-1200 tuples per second as a maximum without any major increase in lag and latency. The system was able to keep a median tuple consumption rate of 1300 tuples per second, however this caused larger impacts on the lag and latency.

The usage of an SPE on an on-board computer for analyzing data also raises concerns regarding how the data should be transferred from the vehicle network(s) to the on-board computer. The results of this thesis show a large difference in performance when running a common data broker on a low-powered device compared to a powerful laptop. In a scenario with an SPE on an on-board computer the system should have a reliable method of transferring tuples to the SPE, persist non-consumed data, all while not adding any bottle necks to the setup.

8.2 Future work

Several extensions can be made for the work of this thesis and some might even be required to further investigate the performance on an SPE on an on-board computer. One primary extension is to investigate the problem with the data broker. The problems noticed with the broker might be related to configuration specific details, however the same configuration was used on the laptop so that might not be a valid reason. Nonetheless, understanding the performance issues and possibly exploring other message brokers or protocols could be a valid extension to find the most suitable way to transfer signal data from the flight recorder to the SPE framework. Similarly, it might be beneficial to further investigate the latency spikes

that occurs to see if this really occurs due to the filter and watermarks or if there is another explanation. Apache Flink also have many different ways of creating watermarks and triggering windows for computation so there might be more efficient ways to handle the windows.

Another extension could be to parallelize the implementation of the stream processing. The current model training algorithm does not make use of any parallelism. Due to this, one interesting thing to explore would be to parallelize it locally in the SPE. As it is proven that stochastic gradient descent can be used in a parallelized task [28] such implementation might result in a higher maximum input rate for Flink.

Another interesting extension would be to run this implementation in a complete federated learning system, with a central node that manages worker nodes, in the scope of this thesis the central node could be a cloud server and the worker nodes the on-board computers. The central node would keep track of a global model for the battery health, or possibly multiple models depending on geographical locations, and push this model to the worker nodes. The worker nodes could use the implementation of this thesis to update the model based on local data generated in real-time on-board the vehicle, once the model is updated it is transmitted back to the central node.

Finally, another interesting work for the future would be to look at the performance of chips or processors that targets machine learning. As an example, the newly announced Snapdragon Hexagon 685 DSP[29] is a unit optimized for AI and machine learning algorithms.

Bibliography

[1] Amos Albert. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. Embedded World, 322:235–252, 2004.

[2] Evan Hirsh Dietmar Ahlemann Edward H. Baker David Crusius ... Trent Warnke Richard Viereckl, Alex Koster. Connected car report 2016. opportunities, risk, and turmoil on the road to autonomous vehicles. "https://www.strategyand.pwc.com/media/file/Connected-car-report-2016.pdf", 2016. Accessed: 2018-02-26.

[3] Hitachi Data Systems. The internet on wheels and hitachi, ltd. "https://www.hitachivantara.com/en-us/pdf/white-paper/hitachi-white-paper-internet-on-wheels.pdf", December 2015. Accessed: 2018-02-26.

[4] Inc. Gartner. Gartner says by 2020, a quarter billion connected vehicles will enable new in-vehicle services and automated driving capabilities. "https://www.gartner.com/newsroom/id/2970017", January 2015. Accessed: 2018-02-26.

[5] International Energy Agency. Global ev outlook 2017. 2017.

[6] Green Car Reports John Voelcker. Electric car battery warranties compared. "https://www.greencarreports.com/news/1107864_electric-car-battery-warranties-compared". Accessed: 2017-12-12.

[7] Yann Guezennec Giorgio Rizzoni Nullo Madella Vincenzo Marano, Simona Onori. Lithium-ion batteries life estimation for plug-in hybrid electric vehicles. pages 536–543, Sept 2009.

[8] Evelina Wikner. Lithium ion Battery Aging: Battery Lifetime Testing and Physics-based Modeling for Electric Vehicle Applications. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2007. Licentiate thesis.

[9] Jens Groot. State-of-Health Estimation of Li-ion Batteries: Cycle Life Test Methods. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2012. Licentiate thesis.

[10] Kandler Smith, Matthew Earleywine, Eric Wood, Jeremy Neubauer, and Ahmad Pesaran. Comparison of plug-in hybrid electric vehicle battery life across geographies and drive cycles. In SAE Technical Paper. SAE International, 04 2012.

[11] Françoise Simonot-Lion Nicolas Navet. In-vehicle communication networks - a historical perspective and review. 2013.

[12] Broadcom Corporation. Broadr-reach physical layer transceiver specification for automotive applications. "http://www.ieee802.org/3/1TPCESG/public/

BroadR_Reach_Automotive_Spec_V3.0.pdf", may 2014. Accessed: 2017-12-06.

[13] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. SIGMOD Rec., 34(4):42–47, December 2005.

[14] Apache software foundation. Apache spark™- lightning-fast cluster computing. "https://spark.apache.org/". Accessed: 2017-12-06.

[15] Apache software foundation. Apache storm. "https://storm.apache.org/". Accessed: 2017-12-06.

[16] Apache software foundation. Apache flink: Scalable stream and batch processing. "https://flink.apache.org/". Accessed: 2017-12-06.

[17] Stephan Ewen Volker Markl Seif Haridi Kostas Tzoumas Paris Carbone, Asterios Katsifodimos. Apache flink™: Stream and batch processing in a single engine. IEEE Data Eng. Bull., 38:28–38, 2015.

[18] MM Mukaka. A guide to appropriate use of correlation coefficient in medical research. Malawi Medical Journal : The Journal of Medical Association of Malawi, 24(3):69–71, 2012.

[19] Xin Yan and Xiao Gang Su. Linear Regression Analysis, Theory and Computing. World Scientific, 2009.

[20] Léon Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent, pages 177–186. Physica-Verlag HD, Heidelberg, 2010.

[21] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[22] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: Concepts, applications and issues. In Proceedings of the 2015 Workshop on Mobile Big Data, Mobidata '15, pages 37–42, New York, NY, USA, 2015. ACM.

[23] Luis M. Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. SIGCOMM Comput. Commun. Rev., 44(5):27–32, October 2014.

[24] Jaeyong Rho, Takuya Azumi, Hiroshi Oyama, Kenya Sato, and Nobuhiko Nishio. Distributed processing for automotive data stream management system on mixed single- and multi-core processors. SIGBED Rev., 13(3):15–22, August 2016.

[25] S. Krishnaswamy, S. Loke, A. Rakotonirainy, O. Horovitz, and M. Gaber. Towards situation-awareness and ubiquitous data mining for road safety: rationale and architecture for a compelling application. 2005.

[26] Anshu Shukla and Yogesh Simmhan. Benchmarking Distributed Stream Processing Platforms for IoT Applications, pages 90–106. Springer International Publishing, Cham, 2017.

[27] M. Jirstrand G. Ulm, E. Gustavsson. Functional federated learning in erlang. 2017.

[28] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J. Smola. Parallelized stochastic gradient descent. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, Advances in Neural Information Processing Systems 23, pages 2595–2603. Curran Associates, Inc., 2010.

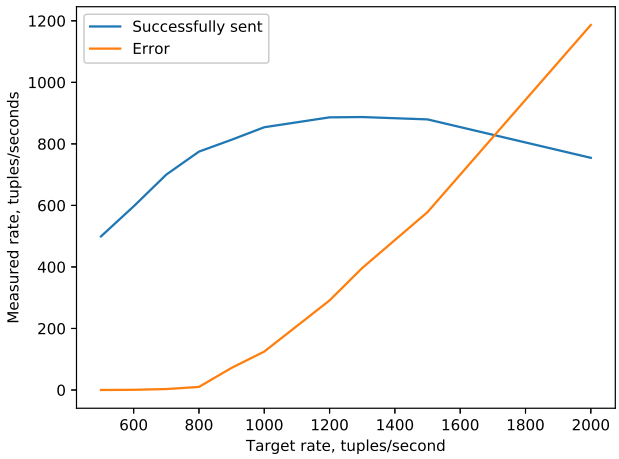
[29] XDA Developers Kyle Wiggers. Qualcomm's hexagon 685 dsp is a machine learning powerhouse. "https://www.xda-developers.com/qualcomm-snapdragon-845-hexagon-685-dsp/". Accessed: 2017-12-21.

A

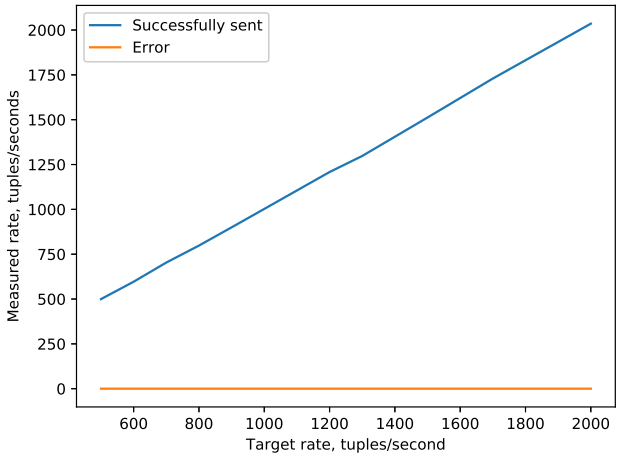
Practical limitations in connection to using the Kafka system

As mentioned in section 7.1.2 it was noted during the executions with various output rates that the Kafka broker hosted on a Raspberry Pi became a bottle neck.
Figure A.1a and A.1b show the number of reported errors as the producer output rate increases. The reported errors represent the number of tuples that was sent by the producer but not properly received by the broker. We can see that when the broker runs on the Raspberry Pi and the target output rate is higher than 800 tuples per second the number of reported errors increases while the number of successfully sent tuples does not match the targeted output. After an output rate of 1500 tuples per second the number of successfully sent tuples even decreases. When the broker is running on the laptop the number of reported errors stay constantly at 0 and the successfully sent is matching the target output rate.

The number of successfully sent tuples is also reflected in the size of the windows used for the model training. As the implementation uses event time windows the windows should contain the same amount of tuples every time we run the process with the same data set. When the process run the size of window w_i and w_{i+1} may be of different size as the windows represent different event times, but when several runs with the same data set is made the size of each window should be constant. Figure A.2a show the window sizes for each model update. Each model update is based on one event time window of data and the data size should be constant no matter what the output rate is. As the figure shows, the data set size decreases as the output rate increases. Figure A.2b shows the window sizes with the broker on the laptop instead, here we can see a steady window size that does not change with the output rate.
It was also noted, but not captured in the metric collections, that the successfully sent messages was higher and error was lower when the producer was active but not the consumer. If both the producer and the consumer was active at the same time the broker would not maintain the rate of successful message transfers and the errors would instead increase.

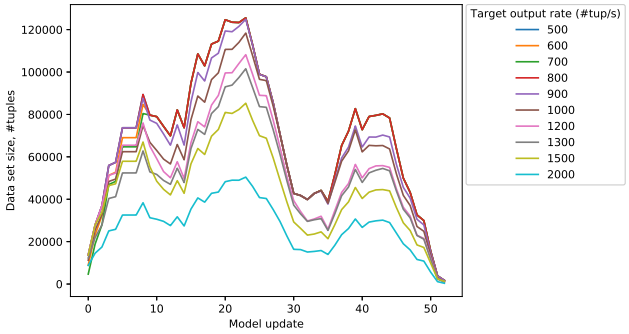


(a) Kafka on pi

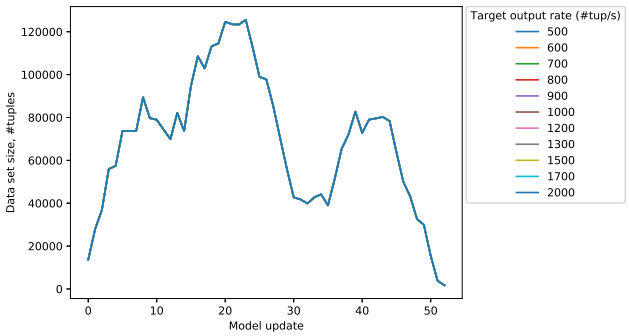


(b) Kafka on laptop

Figure A.1: Error rate



(a) Kafka on pi



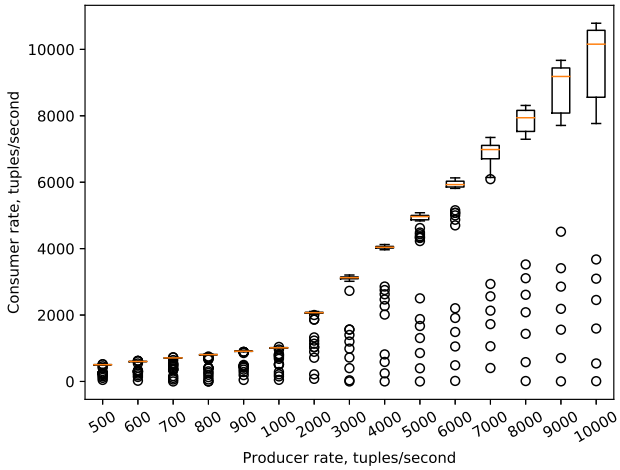
(b) Kafka on laptop

Figure A.2: Window size(s)

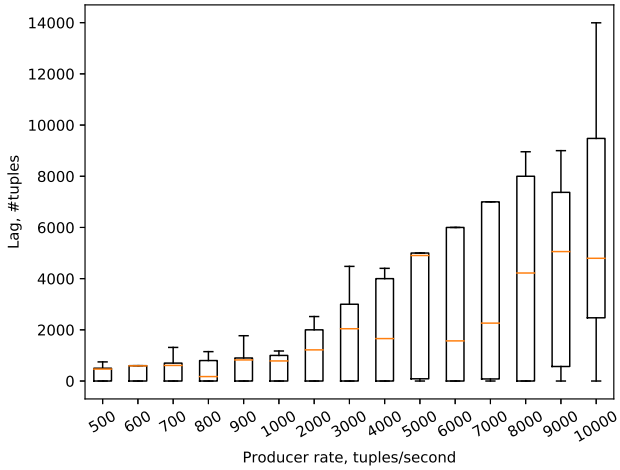
B

Performance of analysis on x64 hardware

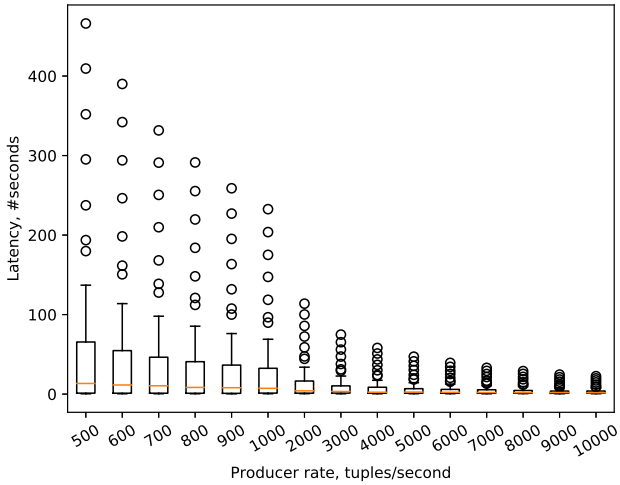
Out of curiosity the stream processing workload was executed on Apache Flink on the laptop in the evaluation setup too. This was made to have a quick understanding of what a modern laptop could handle compared to an ARM device such as the Raspberry Pi. When doing this execution all programs mentioned in the implementation was running on the laptop. This includes the producer, the Kafka broker, Apache Flink and the streaming analysis with linear regression training and the performance metric collection processes. The Apache Flink setup was the same as on the Raspberry Pi in terms of memory and allocated task slots.
Figure B.1 show the three performance properties that was in the scope of the evaluation. Figure B.2 show properties related to each model update for every output rate used. As in the Evaluation chapter, the X axis on the type of plots in figure B.2 represent each new model update. This reveals the latency, size of data set and the training time for all 54 models trained for every output rate.



(a) Throughput

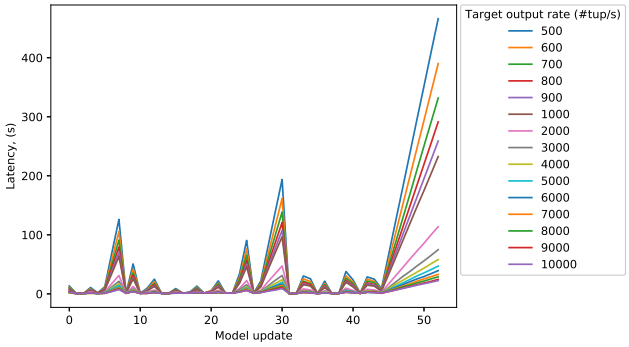


(b) Queue length/lag

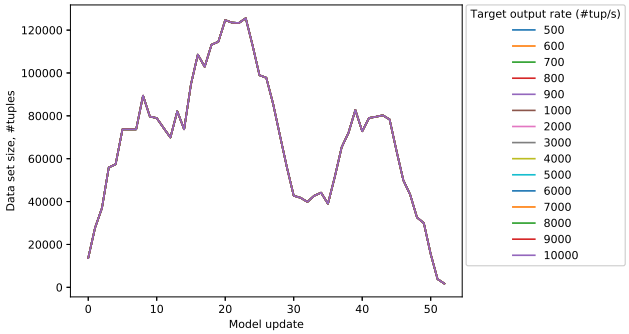


(c) Latency

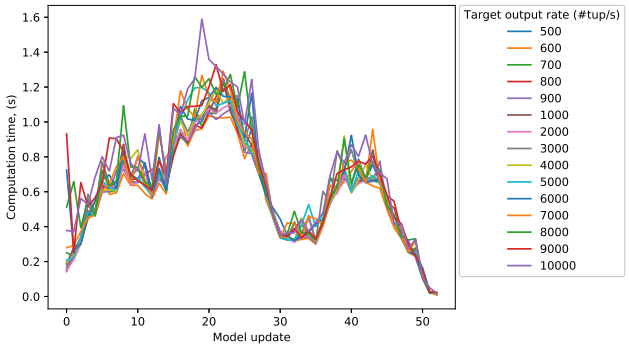
Figure B.1: Throughput, lag and latency when running the analysis on x64 hardware



(a) Latency



(b) Window size (number of tuples)



(c) Training time

Figure B.2: Detailed data per model update and rate