CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Vehicle data analysis using cloud-based stream processing

Master's thesis in Computer Systems and Networks

MORHAF ALARAJ

PHILIP BOGDANFFY

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

Master's thesis 2018

Vehicle data analysis using cloud-based stream processing

MORHAF ALARAJ
PHILIP BOGDANFFY

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Networks and Systems
Distributed Computing and Systems Research Group
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Vehicle data analysis using cloud-based stream processing
MORHAF ALARAJ
PHILIP BOGDANFFY

Master's Thesis 2018
Supervisor: Vladimir Savic, Department of Computer Science and Engineering
Supervisor: Philippas Tsigas, Department of Computer Science and Engineering
Supervisor: Jonas Thorsell, Volvo Group Trucks Technology
Examiner: Marina Papatriantafilou, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Division of Networks and Systems
Distributed Computing and Systems Research Group
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Vehicle data analysis using cloud-based stream processing
MORHAF ALARAJ
PHILIP BOGDANFFY
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In the automotive industry, *maintenance planning* can be seen as a set of strategies that aims to increase the uptime of vehicles. Unexpected events such as if a vehicle suddenly stops due to internal component failures or due to a flat tire are in the context of maintenance planning referred to as *unplanned stops.* For transport vehicles, unplanned stops are critical and will in most cases lead to late deliveries and possibly damaged goods and can in the worst case imply dangerous consequences for the driver of the vehicle. As data become more and more available and the connectivity in vehicles get better, more advanced techniques can be applied when it comes to maintenance planning.

The demand of creating self-learning or automated systems that can predict unplanned stops is increasing due to the big amounts of data that are generated by today's systems. Manually analyzing vehicle data is becoming an unsustainable approach and it is hard for humans to keep up with the digitised systems due to increased complexity and big data amounts in the vehicles.

In this thesis, we propose a concept that enables stream processing on vehicle data on a remote machine. The implementation of the proposed concept is built using state-of-the-art streaming components, namely Apache Spark Streaming and Apache Kafka. This thesis focuses more on the design of the system architecture and the components of the concept.

The results show that it is possible to create machine learning models that continuously evolves and learns from data streams. Implementations of the proposed concept can for example be used to detect anomalies in vehicle components remotely without re-configuring any software inside the vehicles. The machine learning models that were trained with a Volvo data set did not deliver the desired prediction accuracy for the area of maintenance planning. Future work in this area would require further research in which online machine learning algorithms that best fits this vehicle data and also how features should be chosen to be able to predict anomalies in vehicle data.

Keywords: Anomaly detection, Stream processing, Machine learning, Maintenance planning

Acknowledgements

We would like to thank our supervisors Philippas Tsigas and Vladimir Savic and examiner Marina Papatriantafilou at Chalmers University of Technology for their advice and guidance during this thesis. We would also like to thank Jonas Thorsell and Magnus Svensson at Volvo Group Trucks Technology for their invaluable insights and support during this thesis.

Morhaf Alaraj & Philip Bogdanffy, Gothenburg, December 2018

Contents

List of Figures

List of Tables

1 Introduction

Data streams are unbounded flows of data that have been created from one or many sources. The concept of processing data streams is called *stream processing* and is an alternative to the processing of data that can be found in a database-management system. Today, stream processing is widely used in applications that require analysis on continuous data as for example monitoring Twitter data in order to discover trending topics.

In this thesis, the concept of stream processing will be investigated in the automotive industry in order to propose a new concept that includes processing of vehicle data using stream processing. An implementation of this concept could for example be used to monitor vehicles health remotely. It could also be used in order to predict component failures in vehicles using machine learning.

In the automotive industry, *maintenance planning* can be seen as a set of strategies that aims to increase the uptime of vehicles. In contrast to uptime, we have downtime which is a result of unexpected events such as a flat tire or a sudden component failure. These events are referred to as *unplanned stops* and for transport vehicles, unplanned stops are critical and will in most cases lead to late deliveries and possibly damaged goods, which is the motivation for preventing unplanned stops [17]. The most common strategy today for increasing the uptime of transport vehicles requires human expert knowledge and includes scheduled workshop visits according to vehicle component life-time statistics. The increasing amount of data generated by the digitised systems in the vehicle makes it difficult for humans to keep up with the increased amount of data. Analyzing vehicle data manually is not a viable approach due to the massive amounts of data that are generated, therefore the digitised systems in vehicles needs to implement efficient data processing and self-learning models. [8]

Recent studies [25] shows that it is possible to predict failures for specific components in transport vehicles using machine learning techniques. However, the fact that each component in the vehicle is unique in its characteristics makes it difficult to implement a generic model that will predict faults for every component in the vehicle. Therefore, the aim of this work is to propose a concept that enables cloud-based processing of on-board vehicle data in order to predict unexpected behaviour. The motivation for using cloud-based stream processing is that powerful remote machines have more computing power than what is available on-board and analyzing the data off-board also enables finding correlations between different vehicles in order to predict anomalies. Moreover, in case of software changes, only the software in the remote machines needs to be re-configured, instead of each vehicle. Volvo Trucks have provided a data set containing vehicle data that will be used in this work. By enabling communication between vehicles and a remote machine, we believe that it will be valuable for future maintenance planning strategies. To demonstrate this, a proof of concept will be presented where we apply machine learning techniques on data streams with the goal to show that it is possible to perform anomaly detection on vehicle data.

1.1 Context and Motivation

Unplanned stops for transport vehicles are very costly for both the vehicle manufacturers and also their customers [24]. In addition to this, an unplanned stop can in the worst case imply dangerous consequences for the driver. Damaged goods, dissatisfied customers and the cost of towing the vehicle to a safe place are also consequences of unplanned stops. Mitigating work in the area of maintenance planning today is performed manually [16]. Even though research in the area has been successful and proved that the cost reduction will increase significantly using advanced predictive maintenance methods [25], the market is way far behind today's research. The ability to send vehicle data to external systems is limited mostly due to limitations in connectivity since transport vehicles cross border regularly. Huge amounts of collected vehicle data are manually analyzed by experts in the field and therefore the need for automation and learning in this process is necessary since the number of connected devices and the amount of transmitted data will continue to increase [11]. Manually analyzing vehicle sensor data requires human expert knowledge and as the digitised systems inside the vehicles become more complex, this becomes an unsustainable approach.

This research is necessary in order to evaluate the possibilities of implementing anomaly detection in vehicle data using stream processing and machine learning techniques. It is also necessary in order to evaluate the performance of evolving machine learning models that uses stream processing. By enabling efficient computation of vehicle data on a remote machine, i.e a back-office system, we believe that future maintenance planning methods can benefit from this approach. One reason for this is that software and hardware in vehicles would not have to be patched as new models are implemented. Instead, new models can be implemented on a powerful remote machine which also can scale up in order to handle much more data from multiple vehicles simultaneously.

1.2 Goals and Challenges

The main goal in this thesis is to propose a new concept of cloud-based processing of vehicle data. An implementation of the concept will be performed in order to evaluate the performance and capabilities of the proposed concept. The concept includes two-way communication between a vehicle and a remote machine, vehicle data collection and cloud-based stream processing.

The implementation will evaluate how the proposed concept performs in production as well as providing a use case for vehicle maintenance purposes including machine learning techniques. The goal with the machine learning is to show that it is possible to predict anomalies in vehicle data by implementing the proposed concept. A successful implementation of the proposed concept should be able to process vehicle data with high throughput and low latency. The stream processing engine Apache Spark Streaming will be used in the implementation in order to process data streams generated by vehicles. Apache Spark Streaming was preferred over other stream processing engines due to the fact that it supports online machine learning and provides an intuitive API. However, the only streaming machine learning algorithms that are available for Apache Spark Streaming are Streaming K-Means and Streaming Logistic regression. These algorithms will be used for the second part of the implementation.

Our contribution is a concept that implements cloud-based stream processing of on-board vehicle data that enables researchers and engineers within the area to implement models that can be used for analyzing a vehicles health. For example, new machine learning techniques could be tested and evaluated in order to detect whether a component in a vehicle is likely to fail or not.

1.3 Stream processing engine

Two stream processing engines were evaluated in an early experimental stage of this thesis, namely Apache Storm and Apache Spark Streaming. The requirement for the stream processing engine was that it should provide many integrations out of the box, such as with data pipelines and machine learning frameworks. This was a requirement because the concept and the system architecture was not specified and we needed to be able to quickly evaluate and test different system architectures. The first candidate, Apache Storm, was used with the extension Trident-ML in order to try using online machine learning with stream processing. Unfortunately Trident-ML was not compatible with newer versions of Apache Storm and the code contribution on Github for the project had decreased. Trident-ML was then integrated with an older version of Storm, version 0.8.3. However, trying to integrate Trident-ML with the data pipeline Apache Kafka in this version was not successful. The next step was to evaluate Apache Spark Streaming. The installation and setup was made quickly due to the rich documentation. In order to try out the online machine learning functionality, MLlib was installed and could easily be integrated with Apache Kafka. Since Spark Streaming met the requirements, it was the obvious choice for this thesis.

1.4 Thesis Structure

Chapter 2 will present the background of the main topics in this thesis, namely stream processing, machine learning and maintenance planning. The proposed concept is described in Section 2.5. We also present work related to the fields *maintenance planning* and *anomaly detection* in Section 2.6. Moreover, Chapter 3 presents our implementation of the proposed concept. The experimental results and evaluations are presented in Section 4. Finally, the Discussion and Conclusion is presented in Chapter 5.

2 Background

This Chapter introduces the main topics in this thesis, namely stream processing, machine learning and maintenance planning. The first sections present the paradigms and techniques that are later used in the proposed concept which is described in Section 2.5.

2.1 Stream Processing

Data streams are unbounded flows of data, i.e. unbounded sequences of tuples that share the same schema. Examples of data streams are network traffic and sensor data. In stream processing, a tuple can be seen as a stream element and a schema is composed by attributes *A1,...,An* [9]. The concept of data streams processing is called *stream processing* and is an alternative to the processing of data that can be found in a database-management system (DBMS). Stream processing is distinguished by that the rate of arriving items is not under control by the system, while the rate of arriving items are limited by the disc speed in a DBMS. In stream processing, the following assumptions are made [15]:

1. If the arriving data from a stream or streams is not stored or processed immediately, it is lost forever.
2. Data arrives so rapidly that it is not viable to store it in a conventional database.

2.1.1 Definitions

Figure 2.1 presents a Data Stream Management System and a stream processing engine that processes the arriving tuples and output a new stream. The data that the stream processing engine is working with is temporarily stored in the working storage. In stream processing, queries are continuously executing on a data stream by a *stream processing engine*. These queries are called continuous queries. Continuous queries in the context of data streaming are defined as Directed Acyclic Graphs (DAGs) of internally connected operators and are executed by the stream processing engine in order to perform computations on the data. [9] A *sliding window* is a bounded part of the data stream, e.g. the most recent n elements or the items that arrived the last x minutes. Sliding windows are used due to the unbounded nature of data streams [9].

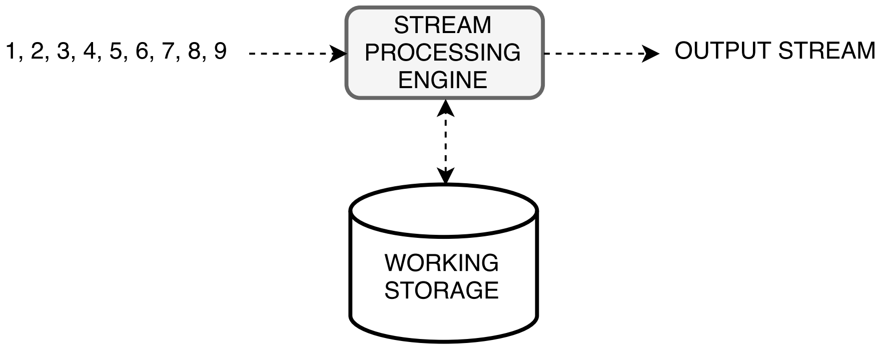

Figure 2.1: Data Stream Management System

A *stream operator* is a function that processes tuples in order to output new modified tuples. An example of a stream operator is a filtering function that removes unnecessary tuple elements in an incoming stream. A stream operator that perform this kind of filtering is called a *stateless operator*, which means that it does not maintain any state in order to perform the filtering. There exist another type of stream operator, namely a *stateful operator*, which maintains a state of the incoming tuples and uses this in order to compute the result. An example of a stateful operator is the *count* function that counts the number of occurrences for elements in the incoming stream. In order to count the elements, we need to maintain a state for each element so that we know how many times it has appeared. Stateful operators are also used when applying machine learning techniques on data streams in order to achieve evolving machine learning models that learn over time.

2.1.2 Micro-batch processing

Stream processing is often compared to *batch processing.* In batch processing, the entire data that will be processed is available at launch time and grouped into different *batches.* This means that no new data will considered in the processing when the batch processing starts. In stream processing, the goal is to process incoming data as soon as it arrives. *Micro-batch processing*, i.e. micro batching, is considered as the intersection between stream processing and batch processing. In micro batching, the incoming data that arrived within the explicitly defined *batch interval* is grouped into small batches. In micro batching, we can also have a sliding window in order to specify how much of the historical data that should be included in each batch. Figure 2.2 shows how micro batching relates to stream processing and batch processing. The stream processing engine that is used in this work, namely Apache Spark Streaming, implements the concept of micro-batch processing.

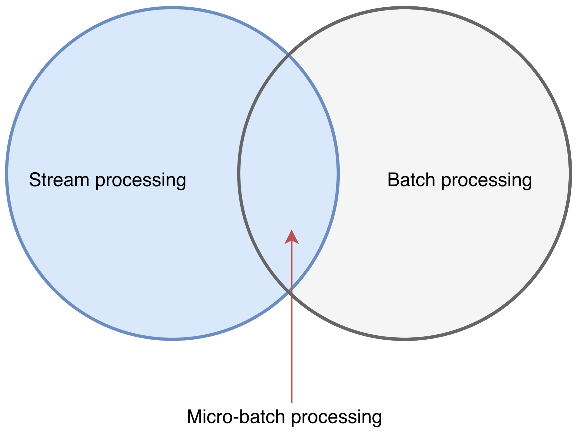

Figure 2.2: Intersection of stream processing and batch processing, micro batching

2.1.3 Apache Spark Streaming

Spark Streaming is a stream processing engine that enables scalable processing of data streams. It is scalable in that sense that it is possible to add more computational power using additional machines, i.e. computational nodes. Spark Streaming processes tuples in a micro-batching manner as described in 2.1.2. Spark Streaming is an extension of the Spark Core API. It can be used together with machine learning libraries in order to perform machine learning techniques on data streams. Spark Streaming API's can be used in the programming languages Java, Scala and Python. Spark Streaming applications can be submitted to a Spark Streaming cluster in order to start processing data streams. Different *cluster managers* exist which aims to allocate resources across the deployed Spark Streaming applications. One example of a cluster manager is Spark's built in standalone cluster manager. When the application is submitted to the cluster, Spark Streaming obtains *executors* on the computational nodes in the cluster. Finally, the packaged code is internally distributed to all executors by the cluster manager.

Receivers in Spark Streaming receives data from the data pipeline in parallell and buffers it to the memory in the computational nodes. A data stream in Spark Streaming is called a *Discretized Stream* and is implemented as a continuous sequence of Resilient Distributed Datasets (RDDs) objects. RDDs are immutable, distributed and partitioned collections of objects in Spark Core API. RDDs can be operated in parallell on different clusters which implies scalability. A RDD contains all the data that belongs to one micro-batch, in other words, each one of the RDDs contains parts of the data that were received during a specific period of time.

In the case of using multiple computational nodes, it is important to consider the order of data and determinism in the output since the order of the output can be critical in some application areas. Using two computational nodes that are responsible for counting and aggregating the number of visitors in two different locations, one of these nodes can fall behind which means that a snapshot of their states indicates

inconsistency.
Using a Discretized Stream, the output RDDs for each interval includes all of the data received in the same interval or previous intervals, therefore this eliminates the issue [2].

RDDs are fault tolerant and can recover fault tolerant data without the use of replication. The lineage graph that consists of the deterministic operations that were used to create the RDD is stored. In other words, the data can be recomputed deterministically. When data is not consumed from a fault tolerant data storage, such as HDFS or S3, the data is replicated in executors in the cluster. Spark Streaming also offers *write ahead logs* which stores data to a fault tolerant data storage in order to recover data.

Spark Streaming offers a web user interface named Spark UI that is used for visualizing different streaming metrics such as processing throughput and read rates for different data streams. Spark Streaming integrates with a machine library called MLlib. MLlib is built on Spark and is used for performing machine learning techniques on incoming data streams. It provides supervised and unsupervised machine learning techniques and can be used for online machine learning. The algorithms *Streaming K-Means* and Streaming logistic regression are available in MLlib.

2.1.4 Apache Kafka

Apache Kafka is a distributed streaming platform [3] that is used for establishing a reliable data streaming pipeline. Kafka uses a publish-subscribe architecture and is fault tolerant. Figure 2.3 shows how messages are exchanged in Kafka. A producer tags a message with a topic while the consumer subscribes to that specific topic in order to consume it.

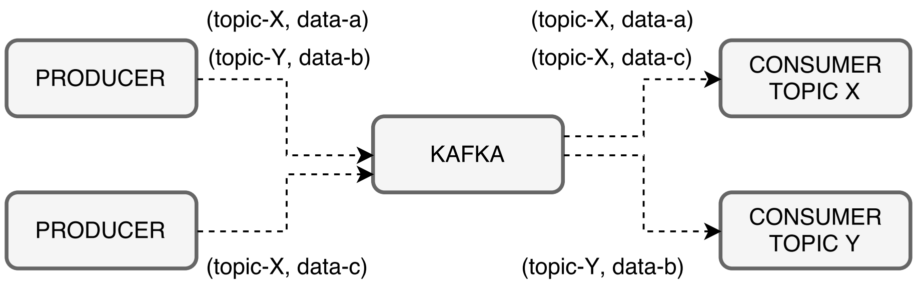

Figure 2.3: Kafka message passing

A topic is where the data is exchanged by the producer and consumer. A topic can be split into partitions and it can be used in order to scale up the storage. Each one of the partitions is ordered and contains immutable sequences of data. Records of data is appended to the partitions in a time ordered manner. In order to identify each record in a partition, an offset value is set for every record. Kafka *brokers* are systems that are responsible for handling the data published to the topics. Spark Streaming provides a Kafka Direct API that enables *exactly once processing* of records when used.

2.2 Machine Learning

Machine learning is about giving machines the ability to learn without being explicitly programmed to have a certain behaviour [19]. In practice, machine learning is used for extracting information from data from which decisions are made. Machine learning algorithms not only extracts and summarizes data, they also build a model based on the data in order to learn something from it and discover something that can be observed in the future [15]. *Online machine learning* is a form of machine learning where new data is used to dynamically update the machine learning model in an online fashion, while the traditional batch learning uses a static data set to learn from the data.

Feature selection is a method that can be used for selecting the most important features in a data set. Before submitting input data to a machine learning algorithm it is often necessary to modify the data in order to have a compatible data set. As an example, consider a classifier that will act as an e-mail spam detector. There are multiple ways of classifying an e-mail as spam. One approach is to analyze each word in the e-mail. Another approach is to look at other useful information, e.g. from which host the e-mail was sent. Spam is often sent from certain hosts that could be spammers, or hosts that are included in a botnet [15]. Adding the host in the feature selection could be a better approach to build the classification model than to analyze each word in the e-mail.

Two major different paradigms exist in machine learning, namely *supervised* and *unsupervised* learning. These paradigms will be described further in the following sections. Semi-supervised learning is a combination of the supervised and unsupervised learning, this paradigm is not covered in the following sections.

2.2.1 Supervised learning

Supervised learning is used when we want to classify data into a predefined class or classes based on parameter values of the data. It is possible to train a supervised algorithm with the use of *training data*. The training data contains labels so that the algorithm will learn which class similar data will belong to. In other words, the supervised learning algorithm is trained with the correct results and expect the classifier to predict the class of new data. For example, a training set could consist of a list of fruits with different properties such as color, weight and shape, but also the actual fruit name, i.e the class. In this case, the supervised algorithm can determine if new data has similar properties as the data in the training set in order to classify new data. *Cross validation* can be used in order to get the prediction accuracy of the trained model. For example, the training set can be split into two data sets, one containing 90% of the data, and the other containing 10% of the data. The first data set can be used in order to train the supervised machine learning model, while the other data, i.e. *validation set*, can be used in order to observe the prediction accuracy of the trained model. Since the labels of the validation set are known, it is possible to calculate the accuracy of the predictions. With cross validation this procedure can be applied n times with a different training and validation set each time and then the average prediction accuracy can be extracted from the n observations.

The training data contains a set of pairs (x,y) and is used to build the model that will be used for future predictions. The value x is called the *feature vector* and can be numerical or categorical. The value y is the label, which is the class of the data. The goal with classification is to predict the y-value associated with values of x, namely a function $y = f(x)$.

Decision tree classifiers is a family of classification algorithms. A decision tree contains a set of nodes that are arranged as a binary tree where each node contains a condition that needs to be fulfilled in order to traverse the tree in a specific order. The classification starts with visiting the root node and evaluating its condition. If the expression is evaluated true, the left child of the root is visited, otherwise the right child of the root is visited. This process is repeated until a leaf is reached, which contains the result of the classification. Figure 2.4 presents an example of a binary decision tree.

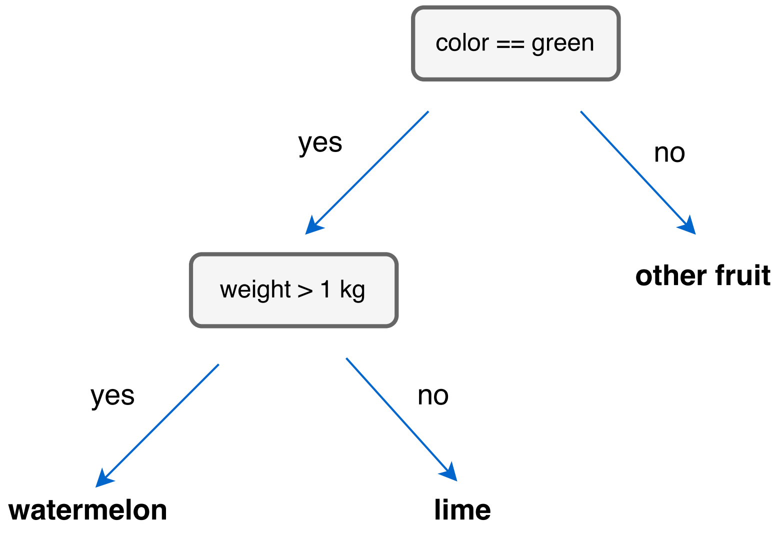

Figure 2.4: Binary decision tree

There exist different types of supervised learning techniques, e.g. classification and regression. Examples of supervised learning algorithms are Support Vector Machines (SVM), K-Nearest Neighbor (KNN) and logistic regression. Logistic regression is a statistical regression model with binomial output which is more of a probabilistic model compared to decision trees. Since the output is binomial, it is used for classification. In logistic regression, the aim is to fit a model to the feature space in order to predict the output of new data points based on this model [4]. The logistic(*sigmoid*) function is a S-shaped curve with the output values between 0 and 1. The sigmoid function is displayed in Equation 2.1.

$$p = \frac{1}{1+e^{-z}} \tag{2.1}$$

The learning of the logistic regression model is about finding the coefficients for the sigmoid function that minimizes the squared mean distance between the sigmoid function and the data points in a data set. This distance can for example be minimized using gradient descent [13] where we apply the gradient descent optimization algorithm to recursively minimize the error for each feature in a sample. The gradient descent algorithm is applied until convergence to a local minimum. The convergence time of gradient descent is determined by a learning rate α which affects the *step size* when converging to a local minimum.

Streaming logistic regression with stochastic gradient descent is the Spark Streaming version of logistic regression. Here, the algorithm does not converge for every training iteration. Instead, a variable *numIterations* specifies how many times the gradient descent algorithm should iterate when the model is trained.

2.2.2 Unsupervised learning

In unsupervised learning we do not know the output of a machine learning function. In other words, our training set does not contain a label for each observation. Unsupervised training is used when we know little about the data and want to discover patterns in the data.

There exist different unsupervised learning techniques, e.g. Clustering, Neural networks and Principal Component Analysis (PCA). *Clustering* can be used for grouping data into different clusters based on the features of the data. For example, the data set $[1, 2, 3, 10, 20, 30]$ could be grouped into two clusters, one cluster that contains data points smaller or equal to 3, and one cluster that contains data points larger or equal to 10. Points in the same cluster will in the ideal case have a short distance between them. A common distance measure that can be used is the n-dimensional Euclidean space. In this distance measure, points are vectors of n real numbers.

An example of an unsupervised learning algorithm is called K-Means clustering. The idea of K-Means clustering is to assign each data point in a data set to the appropriate cluster. Initially, k number of cluster centroids are assigned to the model. The variable k is application specific and defined by the model creator. The position of the clusters could either be randomly chosen or chosen by a mathematical function. Then, each data point in the data set is assigned to the cluster with the shortest euclidean distance to the respective data point. When all data points have been assigned to a cluster, the positions of the cluster centroids are re-calculated using the mean value for all data points in each cluster, until convergence [20]. *Streaming*

K-Means is an online version of the K-Means algorithm. When using this in the context of data streaming, the clusters are updated dynamically when new data arrives. The difference between the Streaming K-Means algorithm and the K-Means algorithm is that clusters centers are calculated continuously as data arrive due to the nature of data streaming. Due to the streaming behaviour, the streaming algorithm does not converge when only trained once. A new cluster is updated according to Equation 2.2 [10].

$$c_{t+1} = \frac{c_t n_t + x_t m_t}{n_t + m_t} \tag{2.2}$$

The previous center for the cluster is defined as c_t and n_t is updated dynamically and contains the number of points assigned to the cluster so far. The new cluster center calculated from the new data stream is defined as x_t while m_t is the number of points that has been assigned to the cluster in the current data stream.

2.2.3 Anomaly detection

Anomaly detection in the context of data streams is the problem of finding deviating patterns in a data stream, i.e. patterns that does not conform with the expected behaviour [6]. Today, anomaly detection is widely used in many areas such as network intrusion detection systems and safety critical systems. Anomalies in data streams can have a different impact on different application areas. In the area of cyber-security, an anomaly can translate to a non authorized attempt to access confidential data. In safety critical systems in the automotive industry, an anomaly could indicate that an underlying electronic vehicle component is faulty.

Machine learning can be used in order to detect anomalies. Machine learning paradigms such as supervised learning, unsupervised learning and semi-supervised learning can be used in order to perform anomaly detection.

For example, a supervised classification algorithm can be used in order to train a machine learning model that finds anomalous behaviours. In this case, a training set that contains both anomalous and normal data is used as input to the algorithm. Unsupervised machine learning can also be used in order to find anomalies. For example, clustering can be used for grouping points that differ from the majority into small clusters. An assumption in this scenario is that small clusters are representing anomalous data, since they differ from the majority. Figure 2.5 presents different clusters in a 2-dimensional data set. Assuming that anomalous behavior is not observed as frequent as normal behaviour, groups G1 and G2 contains points with normal behaviour.

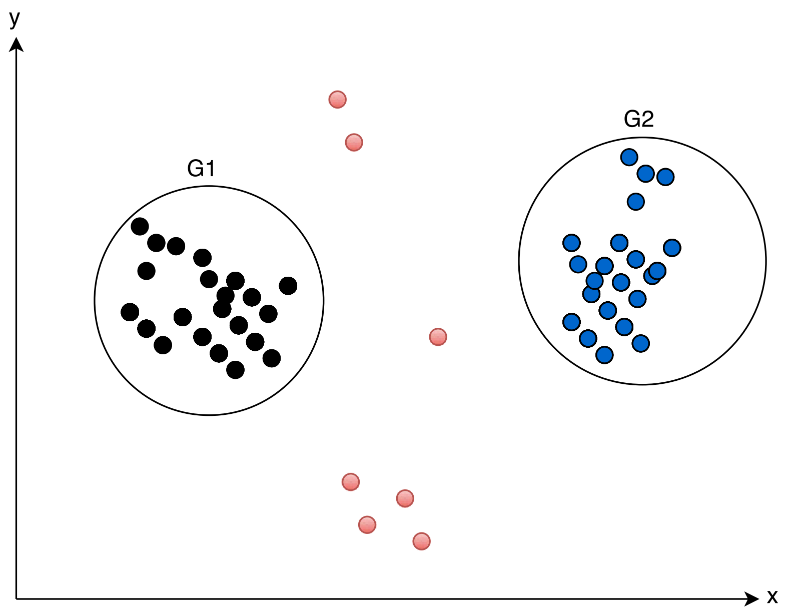

Figure 2.5: Anomalies in a 2-dimensional data set

2.3 Volvo Trucks

This Section describe different maintenance strategies that are relevant for the area of predictive maintenance. It also provides background information for the vehicle components used in this work.

2.3.1 Maintenance strategies

In the automotive industry, *maintenance planning* can be seen as a set of strategies that aims to increase the uptime of vehicles. Uptime is defined as the time period that the vehicle is fully operational, i.e. the vehicle is healthy. In contrast to uptime there is downtime which is a result of different unexpected events such as if a vehicle suddenly stops due to internal component failures or due to a flat tire. These kind of events are referred to as *unplanned stops* and for transport vehicles, unplanned stops are critical and will in most cases lead to late deliveries and possibly damaged goods [17].

Today, Volvo Truck vehicles are scheduled to visit workshops based on maintenance plans designed by Original Equipment Manufacturers. These plans are often designed according to basic vehicle observations as for example a vehicles mileage [25]. As data become more and more available and the connectivity in vehicles get better, more advanced techniques can be applied when it comes to maintenance planning. For example, data mining on big data was not a suitable maintenance prediction strategy only a few years ago. In [25], the authors state that due to limited connectivity in the vehicles, the collected data would vary too much from vehicle to vehicle, making it difficult to extract appropriate data. The accessibility of data is therefore directly related to the quality of maintenance prevention.

There are multiple strategies for the area of maintenance planning. Among these, we can find corrective, preventive and *predictive* maintenance. Corrective and preventive maintenance are widely seen in many applications today and are considered with human actions, i.e repairs or replacements. The difference between the two methods are that in corrective maintenance, a repair or replacement is applied after a fault has occurred. The latter method can be viewed as a countermeasure to an unplanned stop. In preventive maintenance, a component is for example replaced when a vehicle visits a workshop at a scheduled time, before the vehicle has signaled for any faults, i.e a planned stop. Predictive maintenance is similar to preventive maintenance in the sense that both of principles aim to prevent unplanned stops. On the other hand, predictive maintenance does not include any unnecessary replacements of components and therefore also prevents unnecessary planned stops. This makes predictive maintenance the most profitable maintenance planning method in terms of cost, assuming that the predictions are reliable.

2.3.2 Instrument cluster

An instrument cluster is an electronic panel located inside the vehicle. The Human Machine Interface (HMI) is located in the instrument cluster and displays vehicle information such as for example current speed, mileage and notifications. The instrument cluster that is programmed and used in this work runs a Linux operating system. Notifications to the driver can be triggered programmatically and can contain messages to the driver. This is further described in section 3.1

2.3.3 APX framework

APX [12] is a client-server architecture software for sending AUTOSAR [7] signal data to non-AUTOSAR applications. Communication between APX clients and APX Server occurs via sockets. Figure 2.6 illustrates the client-server architecture in APX, where the sockets are represented by squares.

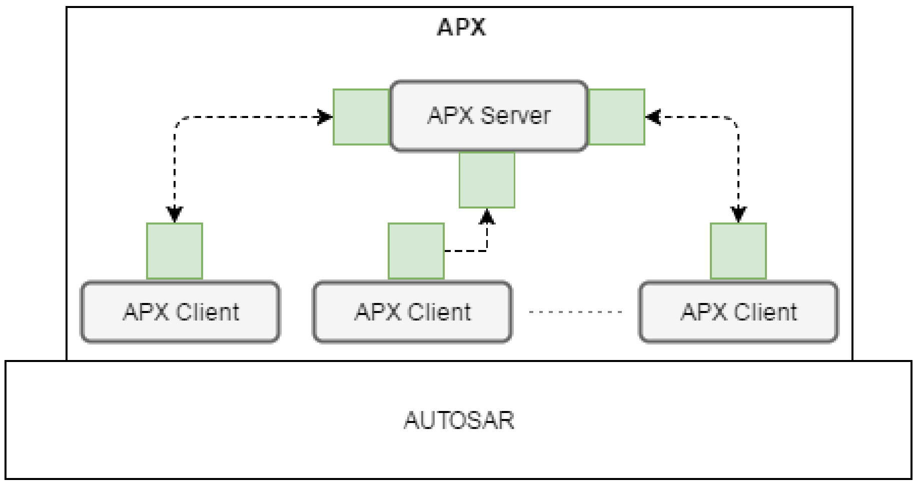

Figure 2.6: Block diagram for the APX framework.

The APX API enables APX clients to provide and subscribe to signals. For example, client A can subscribe to signal X and provide signal Y and client B can subscribe to signal Y and provide signal Z. If client A sends data from signal Y on the socket between itself and the APX Server, client B will receive this data from the socket between itself and APX Server. The VS client uses APX Client in order to set up the socket between itself and the APX Server.

The server part of VS is responsible for receiving data from RSS and to communicate with the HMI-interface which is a part of the instrument cluster. The implementation details of VS is described in Section 3.1.

2.4 Web components

Apache Tomcat is an open source web server and Java Servlet container that can be used for deploying Java web applications based on servlets. A Java servlet is a program that runs on a Servlet container. The servlet acts as a middle layer and handles the requests between HTTP clients and the applications on the server side. The Tomcat element Connector is responsible for the communication with the client, e.g. a HTTP Connector is responsible for handling HTTP traffic. Furthermore, the element Context refers to a web application and is based on a web application archive (WAR). A web application is deployed on Tomcat by simply placing the generated WAR file that contains the web application in the webapps directory in Tomcat.

The Grails web framework is an open source Groovy-based web framework for the Java platform. Groovy is a programming language for the Java platform developed by the Apache foundation. Grails is built for the Java Virtual Machine (JVM) and Grails web applications can be deployed on Tomcat. Java libraries can be imported which enables building extensive web applications.

1. **Controller**
   Grails controllers handles the HTTP requests.

2. **Service**
   Grails services handles business logic and can be called directly from a controller.
3. **Urlmappings.groovy**
   Maps URL's to controller functions.
4. **Build.gradle**
   External dependencies, such as Java libraries, are defined in this file.
5. **Bootstrap.groovy**
   This file is launched upon application startup and can be used for starting services.

   Grails applications can be packaged to a WAR file using command line scripts proviced by Grails, i.e. grails war, which will generate a WAR file that can be deployed on Tomcat.

2.5 Concept

In this section we will present the data processing concept that this work is based on. The main idea in the concept is to process on-board vehicle data efficiently on a remote machine using stream processing. In order to enable processing of on-board data on a remote machine, two software components are required, namely *Vehicle Software* (VS) and *Remote Server Software* (RSS). VS runs inside a vehicle's instrument cluster while RSS runs on a machine in the cloud. The purpose with using VS is to collect vehicle data and transmit the data to the remote machine while that is further processed by RSS. VS is also responsible for vehicle signal collection and displaying pop-up messages in the instrument cluster display.

The concept comes with two different architectures for the data flow; *full architecture* and *bypassed architecture* that are provided in Figure 2.7 and Figure 2.8 respectively. In the full architecture, VS interacts with RSS by sending HTTP POST messages to the RSS web server. The web server then produces the collected vehicle data to a data pipeline. In the bypassed architecture, VS produces collected data directly to the data pipeline in RSS in order to avoid the latency from the web server. The main difference between the two architectures is therefore how data is transferred to the data pipeline.

In both of the architectures, a stream processing engine consumes from the same data pipeline that the web server produced to in the full data architecture and that VS produced to in the bypassed data architecture. Moreover, when the stream processing engine has calculated a result, the result is produced to another topic of the data pipeline. The web server consumes from the pipeline and sends the results back to each vehicle if required.

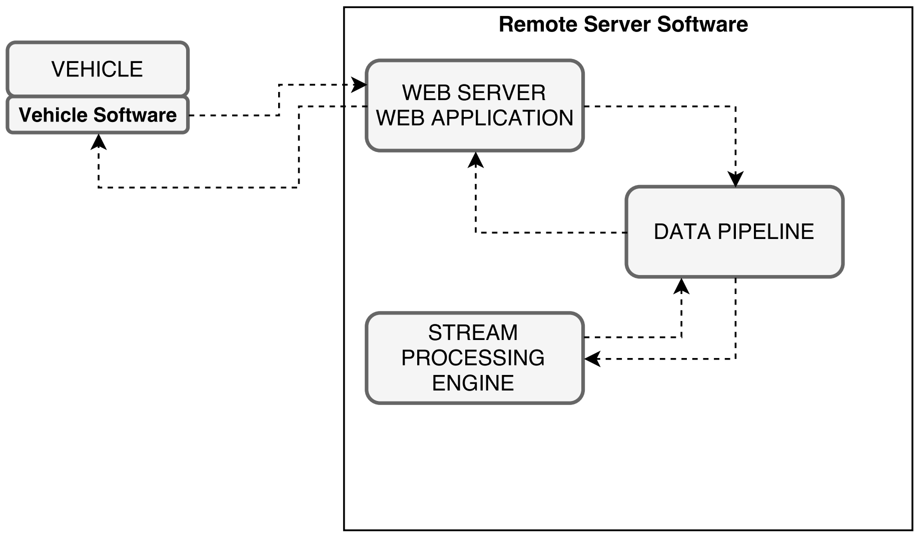

Figure 2.7: High level full architecture of the proposed concept.

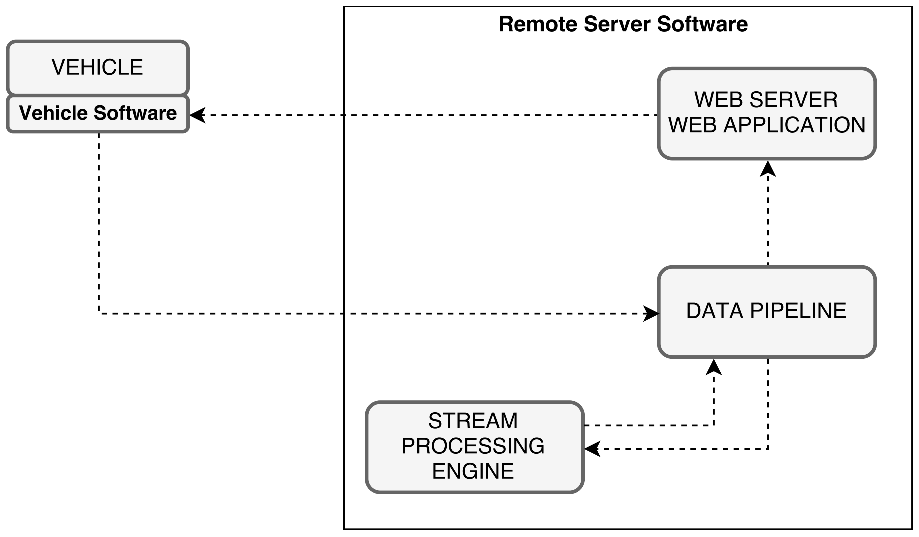

Figure 2.8: High level bypassed architecture of proposed concept.

2.6 Related Work

2.6.1 Maintenance planning

The past decade, the department of Advanced Technology & Research (ATR) at Volvo Group Trucks Technology has proposed different machine learning methods aimed at preventing unplanned stops. Prytz et. al has proposed a predictive maintenance method based on supervised classification algorithms, namely KNN, C5.0 and Random Forest [17]. This method is designed to be used as a complement to scheduled workshop visits and aims to predict air compressor and turbocharger failures based on historical data. The classification is performed before a workshop visit while in our implementation, we detect anomalies on-the-fly in a data streaming fashion using machine learning techniques. The results showed that the proposed predictive maintenance method reduced the maintenance cost, but the classification quality was not great. The authors argue for that the classification quality is highly related to the amount of data that is available. In our work, we will evaluate how the classification quality relates to the amount of training samples and available data.

Another predictive maintenance approach is Consensus Self-Organising Models (COSMO), which is an unsupervised method for performing on-board anomaly detection [8]. COSMO transmits compressed on-board data to a remote machine in order to find deviations among different vehicles. It simply points out a vehicle that deviates from the majority and classifies that one as faulty, i.e finding consensus among vehicle signals. Faulty vehicles are also matched against a database containing vehicle service records in order to increase the prediction accuracy. In this predictive maintenance method, the authors aim to transmit the relationship between signals to the remote machine rather than transmitting the signal value itself. This means that it is up to the on-board software to find the relationships between the signals and decide which signals are interesting. Thus, the feature selection will filter out only signals that are related to each other and consider the other ones as not interesting. The COSMO method were able to predict fifty percent of air compressor faults in a case study with the population of 19 vehicles.

The above mentioned predictive maintenance methods implements a fixed approach or model in order to detect anomalies, while the goal in our work is to provide a concept which enables implementation of different models using a generic and modular approach. Relating to our contribution, our contribution is a concept of remote processing of on-board vehicle data that enables researchers and engineers to implement models that can be used for analyzing a vehicles health in an efficient manner on a remote machine. In Section 3.3.2, we demonstrate the use of both supervised and unsupervised machine learning techniques.

2.6.2 Anomaly detection

In the past decade, anomaly detection in data streams has engaged many researchers in academia. The motivation for using stream processing is the big amounts of data that are generated by today's digitised systems and applications. There exist different approaches for performing anomaly detection on data streams. Ahmad et al. [1] states that detecting anomalies in data streams using machine learning techniques is a difficult task since the detector needs to process data in real time in order to predict and simultaneously learn from the data streams. In our work, this is achieved by Spark Streaming and MLlib, where the learning is performed online.

Rettig et. al [18] proposed an online anomaly detection pipeline. Their approach is to combine two metrics in order to dynamically detect anomalies in large-scale telecommunication networks. The two metrics are *relative entropy* and *Pearsons correlation*. State-of-the art streaming components were used in their work. Apache Kafka is used as a data pipeline while Apache Spark Streaming is used as a micro-batching framework. Kafka and Spark Streaming are used in order to meet the requirements generality and scalability and these components are also used in the implementation of the proposed concept in this work. Compared to our work, Spark Streaming runs on top of YARN cluster manager.

Soni et al. [22] performs anomaly detection on telepresence robots using supervised machine learning techniques, namely KNN, SVM and Random Forest. Sensor data and log files are analyzed in the cloud in order to lower the computational load on the robots. The authors mention that it is a tedious process to identify all component failure types, therefore cloud based data analysis would be a viable option. Moreover, for future work they mention that as the amount of data that needs to be analyzed increases, a distributed machine learning platform would be needed.

3
Design and Implementation

In this chapter we will present an implementation of the proposed concept in order to provide a practical demonstration of what is possible by implementing this concept. Also, the implementation allows us to perform evaluations in order to identify potential bottle necks in the concept architecture.

3.1 Vehicle Software

VS is a client-server architecture software and is built using the programming language Python. The reason for using Python is because the instrument cluster widely supports Python. In VS we have two components; VS-Server and VS-Client, where VS-Server is a web server that listens for incoming requests. Once VS-Server receives data, it creates a VS-Client that is responsible for either:

1. Collecting real-time signal data and transmit the data to the RSS.
2. Displaying pop-ups in the instrument cluster display.

The signal collection is performed using APX, which is described in Section 2.3.3. The reason for implementing VS with client-server architecture is because it is the paradigm used in APX and also that we both want to send data from vehicles as well as receiving data. The block diagram of VS is illustrated in Figure 3.1.

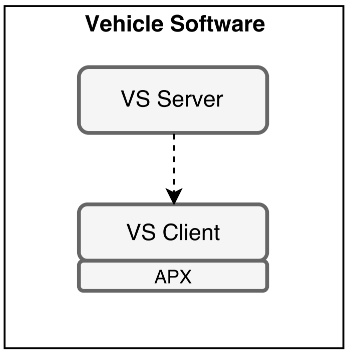

Figure 3.1: Block diagram for the Vehicle Software.

In VS server, we implemented two application entry points, i.e endpoints, that is used to trigger different functions in the software. An endpoint in this context is an URL that can be accessed by HTTP messages from RSS. One of the endpoints is used for creating a VS client that collects signal values and the other endpoint is used for sending notifications to the vehicle. The following two endpoints are implemented in VS:

1. */client*, that expects data that contains the attribute *signals_require*, which value is an array of signals. Each signal has four properties as provided in Table 3.1. An example of data that is sent to the /client endpoint is available in Appendix A.1.
2. */popup*, that expects data that contains the attribute *signals_provide*. This attribute contains the same data properties as for the *client* endpoint. One endpoint triggers the functionality of providing signals, while the other endpoint enables the functionality of requiring or listening to signals. The HTTP messages that triggers these endpoints contain JSON data. An example of data that is sent to the /popup endpoint is available in Appendix A.1.

Table 3.1: Data properties for signals when creating clients in Vehicle Software

Attribute	Type
name	String
apx_signature	String
value	Integer
num_samples	Integer
sample_rate_ms	Integer

In order to prevent blocking from the VS server main thread, each VS client runs in a separate thread. A VS client thread will be terminated as soon as it has sent the same number of signal samples as specified in the attribute *num_samples*. We do not want to keep states for each VS client in RSS and therefore each sample will be stored locally in a list of samples for each VS client until it has received *num_samples* samples. Before client termination, the list of samples is sent either as JSON to the RSS web server using HTTP POST (full architecture) or directly to the Spark Streaming application using Kafka (bypassed architecture). JSON was used because it is convenient to work with in the web application.

We created two different classes in VS called *PopupManager* and *VSClient*. These classes are provided in Appendix A.4 and Appendix A.5. PopupManager creates an APX client, connects to APX server and sends data on the socket between the APX client and APX server. Finally it closes the connection to APX server and terminates the current thread. The VSClient also creates an APX client and then connects to APX server. The difference between the two classes is that VSClient implements a data listener provided by APX and for each received signal it stores the value and timestamp in memory. By sending all samples from VS to RSS in the same request we also reduce network communication overhead.

3.2 Remote Server Software

RSS is responsible for receiving data streams from the vehicles and process these as well as sending data back to vehicles when necessary. It is composed by the following components:

1. **Apache Tomcat and Grails Web framework** [23]
   Tomcat is used as web server and the Grails web application is deployed on Tomcat.

2. **Apache Spark Streaming** [21]
   Spark Streaming is used as stream processing engine.

3. **Apache Kafka** [3]
   Kafka is used as a data pipeline for message passing.

The web components in RSS were chosen over others due to previous experience with these components. As illustrated in Figure 2.7, the web application in RSS is responsible for receiving tuples and sending them to the data pipeline, i.e. to Kafka. The stream processing engine Spark Streaming consumes from Kafka and processes the data and performs machine learning techniques on the stream using MLlib. The result is sent back to the data pipeline and is read by the web application in order to send data back to the vehicle that initially generated the data stream. An example of data that is sent to RSS from a VS-Client is available in Appendix A.2.

3.3 Experiment

We implemented an experiment that consisted of two different parts; *system performance for simulated data* (part I) and *system performance for production data* (part II). In part I we studied how the different components in RSS contributes to the stream processing throughput and the overall latency. The goal with part I was to measure how the stream processing throughput depends on different streaming parameters. For this purpose data was simulated in VS using a Python script A.6 which made it easier to tweak the parameters such as tuple dimension and tuple size. Table 3.2 show the different streaming parameters that we included in the evaluation. The evaluation was performed by selecting different values for the included parameters to observe the processing throughput. The remote machine that RSS was deployed on is a laptop running the operating system Linux Ubuntu 16.04 on a dual core (2 physical/4 logical) Intel i5-6200U CPU at a speed of 2,3 GHz per core with 8 GB of RAM. The vehicle software was deployed on the same laptop, simulating real hardware.

Table 3.2: Parameters that affects the stream processing throughput

Parameter
Stream input rate
Number of tuples
Tuple dimensionality
Number of processing cores
Number of Kafka partitions
Batch interval

The goal with part II was to show that with our implementation of the proposed concept, supervised and unsupervised machine learning techniques can be used in order to perform anomaly detection on vehicle data. For this, we used a data set provided by Volvo Trucks in order to show a valuable use case for the concept. The data set contains vehicle data and a label that indicates if a vehicle has an air compressor fault. Examples of features in this data set are *pumped air volume since last compressor change*, *cruising time mean* and *fuel consumed in Drive*.

3.3.1 System performance for simulated data

Here we created different evaluations based on the parameters presented in Table 3.2. The different data sets were simulated with different combinations of number of samples and dimensions. The different number of sample sizes was 100, 1.000, 10.000, 100.000 and 1.000.000, and for each sample size we generated a data set with one of the following data dimensions: 2, 10, 50, 100, 500. Table 3.3 shows the different data sets used for the different evaluations. Note that each sample size for the same dimension is evaluated separately, meaning that we evaluated 25 different data sets. We also performed an evaluation that measured how the throughput relates to the batch interval for batch interval values between 1 and 10. The Spark Streaming application code is available in A.8.

Table 3.3: Data sets used for *System performance for simulated data.*

Sample size	Dimensions
[100, ..., 1.000.000]	2
[100, ..., 1.000.000]	5
[100, ..., 1.000.000]	10
[100, ..., 1.000.000]	100
[100, ..., 1.000.000]	500

The evaluations was performed using two different Kafka topics, with two different configurations. One configuration where the topics were created with the same number of partitions as the number of available cores on the remote machine and one configuration with only one partition per topic. The reason for this was to evaluate how the Kafka topic configurations affect the stream processing throughput. Table 3.4 shows the different Kafka topics used for part I of the experiment. The different configurations were tested independently, meaning that we first used the topic training_1, collected the results and then performed the same routine with training_4.

Table 3.4: Kafka topics and partition configuration used for part I of the experiment.

Topic	Partitions
training_1	1
training_4	4

We initialized these evaluations by deploying the web application on Tomcat. The Spark application was deployed in standalone mode. Moreover, we created two different Kafka topics as specified in Table 3.4. This procedure was performed both when using one and four Kafka topic partition(s). For simplicity, the two different topics are referred to as *training*. Secondly, we created two VS clients; one that provided the data sets from Table 3.3 one that required the data. The providing VS client was only created in order to be able to simulate data. The code for the VS client creation is provided in Appendix A.7 and the VS code for generating simulated data is provided in Appendix A.6. Once the VS client that required data receives data, it sends the data to the RSS web server over HTTP. The RSS web server then produces the data to the data pipeline which the Spark Streaming application was configured to consume from.

For all of the evaluations, the Spark Streaming application was programmed to loop through each incoming tuple and count the occurrence of each value in the tuple. The code for the Spark Streaming application is provided in Appendix A.8. Finally, the processing throughput for the *full* architecture was observed in Spark UI. The results are presented in section 4.2.1.

Recalling the system architecture from Figure 2.7 we see that data is sent from VS to the RSS web server and thereafter produced to the data pipeline. Here, we studied how the overall streaming throughput was affected by bypassing the RSS web server. This alternative architecture means that VS produces data directly to the data pipeline on RSS. Figure 3.2 and 3.3 shows how data flows when using the full architecture and the bypassed architecture respectively.

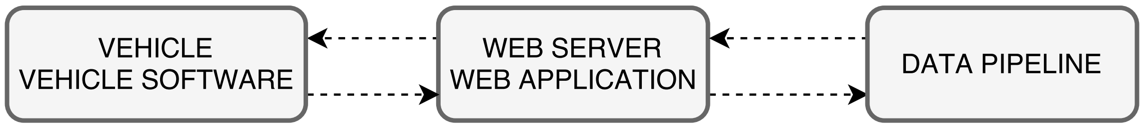

Figure 3.2: Full architecture

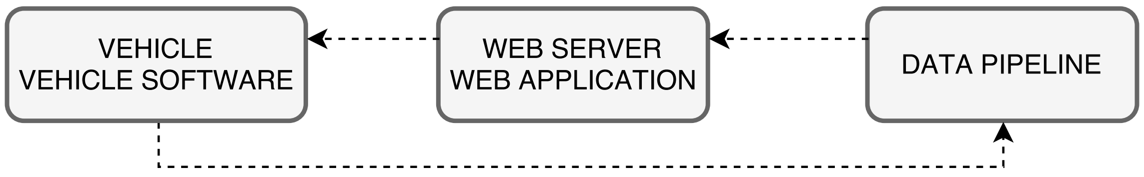

Figure 3.3: Bypassed architecture

3.3.2 System performance for production data

This evaluation was performed by training two different machine learning algorithms; one supervised learning algorithm and one unsupervised. As mentioned in Section 3.3, the goal here was to show a valuable use case for the proposed concept. We trained the two machine learning models based on an existing data set from a Volvo database with the aim to train a model with the characteristics of an air compressor fault. We used the algorithm Streaming logistic regression for the supervised learning and streaming K-Means clustering for the unsupervised learning. The Spark Streaming application code for the Streaming K-Means implementation is available in A.9 and the code for the streaming logistic regression is available in A.10.

In order to have vehicles with similar conditions in the training data set we extracted vehicles that operates in the same country and with the same chassis type and ended up with a data set of 54 features and 2271 samples. Examples of features in this data set are *pumped air volume since last compressor change*, *cruising time mean* and *fuel consumed in Drive*. Finally, we removed samples that had missing values for any of the features and ended up with 800 samples; 400 samples with an air compressor fault and 400 samples without an air compressor fault. An important point here regarding the validation data set is that we were not able to receive all features from this data set in VS using APX since not all features are available in APX. Therefore, we created two VS clients; one that provides simulated signals for 10% of the data set and one that subscribes to these signals. Therefore, we initialized two VS clients; VS-C1 and VS-C2 by sending a HTTP POST message from RSS to VS server for each client.

We initialized part II of the experiment in the same way as for part I, i.e by deploying the RSS web application on Tomcat and deploying Spark Streaming in standalone mode. We split the data set into a training set and a validation set. The training set represents 90% of the data set and the validation set represents 10%.

For the clustering model, two clusters were created when initializing an instance of the class *StreamingKMeans* with the argument *k* equal to two using the MLlib API. One cluster represents vehicles with air compressor faults and one cluster represents healthy vehicles. Here we also specify the dimensionality of the data set and the source of the training and validation data. The training set was specified to come from a text file while the validation set was sent using Kafka. Upon a consumed message from the specified Kafka topic, the model runs the method *predictOnValues* on the stateful K-Means clustering model object that returns which cluster the tuple was assigned to along with the label of the cluster.

For the logistic regression model we used the MLlib API's *StreamingLogisticRegressionWithSGD* class in order to create a model and train it with the training set. The data sources here was the same as for the clustering model. The difference was that the training set in this model was labeled while the validation set was not. Upon a consumed message from the specified Kafka topic, the model runs the method *predictOnValues* on the stateful logistic regression model object that returns the prediction along with the label for the incoming data. This output was later used in order to find the accuracy for the classification algorithm.

When RSS receives samples from VS-C2 it produces the data to a Kafka topic named *validation*, which the stream processing engine is configured to consume from.

If the sample from VS-C2 is predicted as faulty, i.e. an air compressor fault, the stream processing engine produces this data to a Kafka topic called *detection* which the RSS Grails service is configured to consume from. Upon a consumed message from *detection*, Grails sends a HTTP POST message to VS server in the vehicle on the endpoint *popup*. This will launch a VS client using the class PopupManager which will send data on the socket provided by the "name" attribute and a pop-up will be shown in the driver's display. Since this is a signal that we provide and not subscribe to, we only need to send one sample in order to trigger a pop-up.

The machine learning implementation was validated by substituting the Volvo provided data set against the Statlog (German Credit Data) data set [5]. First we used the Matlab classification learner to observe the prediction accuracy for different machine learning algorithms and then we used the same data set with our implementation in order to validate the correctness.

4 Results

This chapter presents the results for the different evaluations presented in Section 3.3. Section 4.1 presents the resulting implementation and the capabilities with the proposed concept. Moreover, section 4.2 presents the results of the experiment where we include metrics for the stream processing. The evaluations within the experiment were performed ten times and the metrics were averaged.

4.1 Concept

The proposed concept of cloud-based stream processing of vehicle data was successfully implemented in a prototype software. Real-time signal collection was successfully performed by utilizing the APX framework. By utilizing the data pipeline Apache Kafka we could either pass vehicle data directly from vehicles to the data pipeline, i.e the *bypassed architecture* or through the RSS web server and then to the Apache Spark Streaming application using the *full architecture*. The pros and cons with the two different architectures are discussed in Chapter 5. Our implementation of the proposed concept is capable of doing the following:

1. Collect vehicle signal data and send it to RSS
2. Process vehicle data in a scalable and fault-tolerant way using Spark Streaming
3. Perform machine learning techniques on vehicle data
4. Trigger HMI-popups from RSS based on the machine learning results

The following Sections presents the results from the different evaluations for our implementation of the proposed concept. Other application areas are further discussed in Chapter 5.

4.2 Experiment

The evaluation of the processing throughput was as mentioned in 3.3.1, performed by simulating different streams of data from VS which represents a fleet of vehicles that continuously transmits data to RSS. In these results we observe how the stream processing throughput depends on the stream input rate, tuple dimensionality, number of processing cores, number of Kafka partitions and batch interval. In all of the observations we consider processing time as the total time it takes to process a fixed amount of tuples. Moreover, the streaming operation in all of the observations was a function that counts the occurrence for each value in a tuple. The time complexity for the used streaming operation is linear.

In part II of the experiment, we show a use case of how our implementation of the proposed concept could be used for future predictive maintenance strategies by using machine learning.

4.2.1 System performance for simulated data

Figure 4.1 shows the throughput for the Spark Streaming application for the bypassed and the full architecture. The number of tuples represents the whole simulated stream.

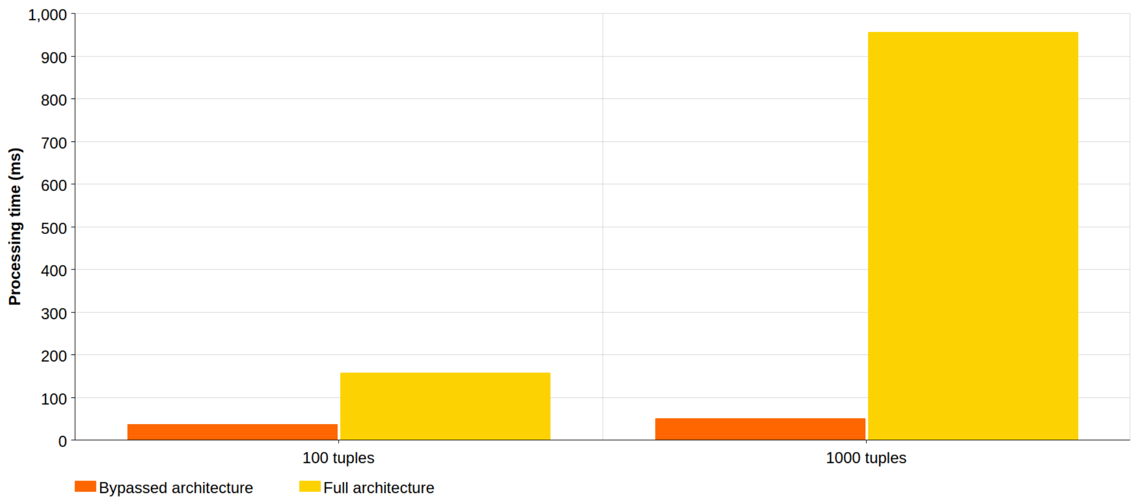

Figure 4.1: Processing time evaluation for the two different architectures.

Figure 4.2 shows the throughput for the Spark Streaming application with respect to tuple dimension when performing a function that counts the occurrence of each value in a tuple on the simulated data stream that was sent from VS. The throughput here was measured using 4 partitions for the Kafka topic, 4 cores for processing data and a batch interval of 5 seconds. The increase in processing time is linear with the number of tuples and dimensions since the streaming operation used in this observation is a linear function.

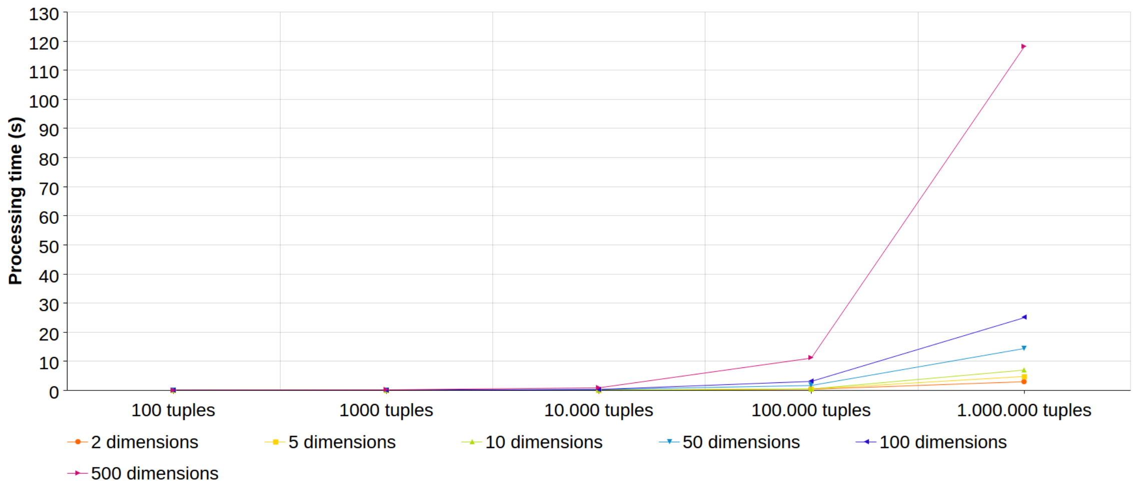

Figure 4.2: Processing time for the different data sets with respect to tuple dimension.

In Figure 4.3, we performed a similar evaluation but instead observed how the throughput is affected by changing the number of Kafka partitions and processing cores. The tuple dimensionality in this evaluation was 10. As mentioned in Section 2.1.3, it is important to consider the order of data and determinism in the output since the order of the output can be critical in some application areas. By using multiple Kafka partitions we increase the throughput of the processing by parallelizing the consumption of messages. However the order of the messages is not preserved and does therefore not guarantee consistency when using multiple Kafka partitions. This is an issue related to the interleaving of the incoming input tuples from different sources.

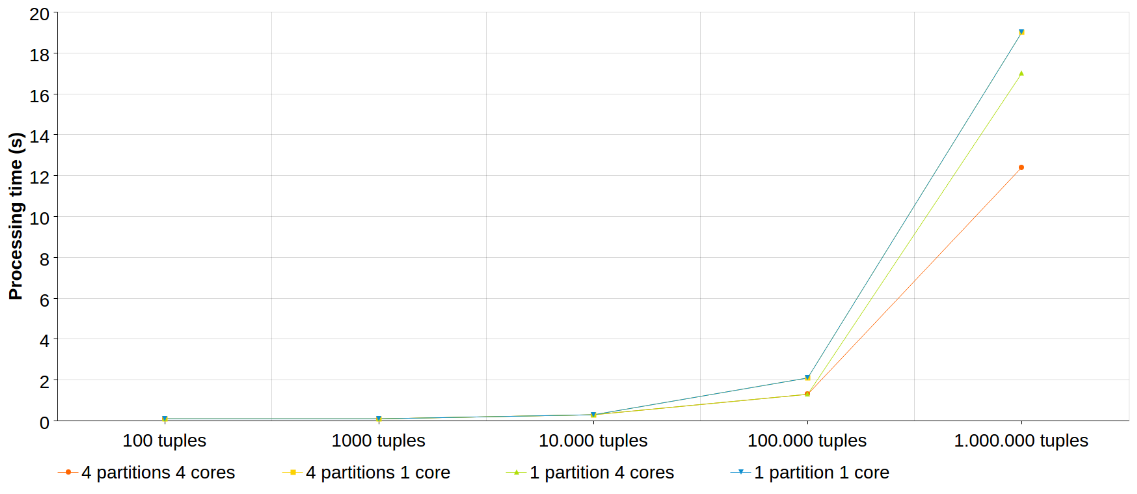

Figure 4.3: Processing time for the different data sets with respect to the number of Kafka partitions and the number of used cores.

A notable point for this part of the experiment is that the total throughput is a function that depends on the processing throughput and also the Kafka throughput, where the Kafka throughput is the rate of how many tuples the stream processing engine can consume from a Kafka topic. If we convert the total throughput to latency, we get the following latency for RSS:

$$L(RSS) = L(Kafka) + L(Processing) \tag{4.1}$$

where

$$L(Processing) = L(Computing) + batch\_interval \tag{4.2}$$

Here, $L(Computing)$ is a function that depends on parameters in Table 3.2, where the parameter $batch\_interval$ is a constant value.

In Figure 4.4 we show the Kafka latency for different tuple input sizes. Here, we observe that the throughput is constant at 7500 tuples/s, making the Kafka latency linear with the amount of tuples. Recalling Figure 4.3 we can see that for any of the configurations regarding number of used cores and Kafka partitions the processing throughput is over 13.000 tuples/s.

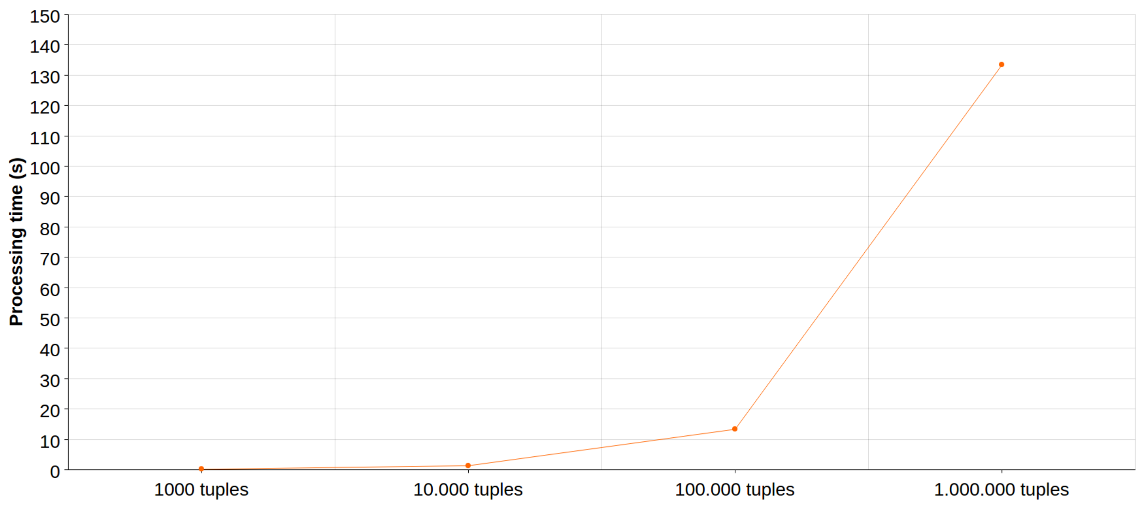

Figure 4.4: Kafka consuming throughput for different data sets.

In Figure 4.5 we can see how the processing time decreases with higher values for the batch interval. We can also see that this only applies for large data sets when using this specific hardware and scaling architecture. In this evaluation, we used a tuple dimension of 5.

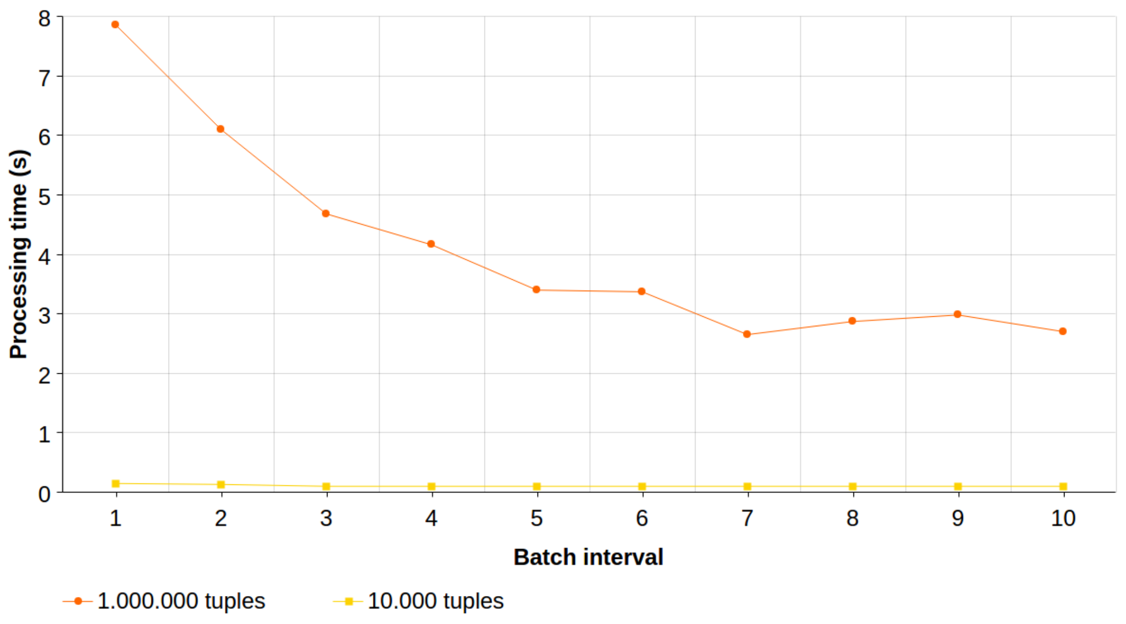

Figure 4.5: Processing time for two different data sets with respect to the batch interval.

4.2.2 System performance for production data

We performed 10-fold cross validation on the data set provided by Volvo Trucks in order to validate the prediction accuracy for the two different online machine learning algorithms that are available in Spark MLlib. The goal here was to show that the machine learning models are getting higher prediction accuracy as we train the algorithms with more and more data. The validation set contained 80 samples of equally distributed classes, i.e 40 with label 0 and 40 with label 1. As mentioned in Section 3.3.2 the training set covered 90% of the complete data set.

The two different algorithms used in this experiment were streaming logistic regression and streaming K-Means Clustering. For the logistic regression model we used the parameters specified in Table 4.1 and here we elaborated with different parameters in order to find the best configuration among those that were tested. For the K-Means clustering model, we set the k value to 2 due to the characteristics of the data set. For both of the algorithms we submitted the complete training set, validated the model with the validation set and observed the prediction accuracy for different model parameters.

Table 4.1: Prediction accuracy in relation to number of training samples

Number of iterations	Gradiant descent step size	Prediction accuracy
1	0.1	50%
10	0.1	50%
100	0.1	50%
1000	0.1	50%
1000	0.01	53%
1000	0.001	50%
10000	0.01	50%

We wanted to observe if the trained model was evolving as it received more training samples and therefore the training set was divided into 5 different subsets according to Table 4.2 where we iteratively trained and validated the model for each subset.

Table 4.2: Subsets of the Volvo training set

Number of samples	% of training set
20	3%
40	5.5%
80	11%
160	22%
620	58%

For the Volvo data set the results matched the desired behaviour with both of the two different algorithms, i.e the prediction accuracy increased as the model was iteratively trained. The prediction accuracy with respect to number of training samples for the streaming K-Means Clustering model is presented in Table 4.3.

Table 4.3: Prediction accuracy for the logistic regression model for the Volvo data set.

Training samples	Average prediction accuracy
20	50%
40	50%
80	50%
160	50%
620	57.5%

The prediction accuracy with respect to number of training samples for the streaming K-Means Clustering model is presented in Table 4.4.

Table 4.4: Prediction accuracy for the streaming K-Means Clustering model for the Volvo data set.

Training samples	Average prediction accuracy
20	52.5%
40	52.5%
80	55%
160	55%
620	57.5%

Figure 4.6 shows a comparison between the two different algorithms.

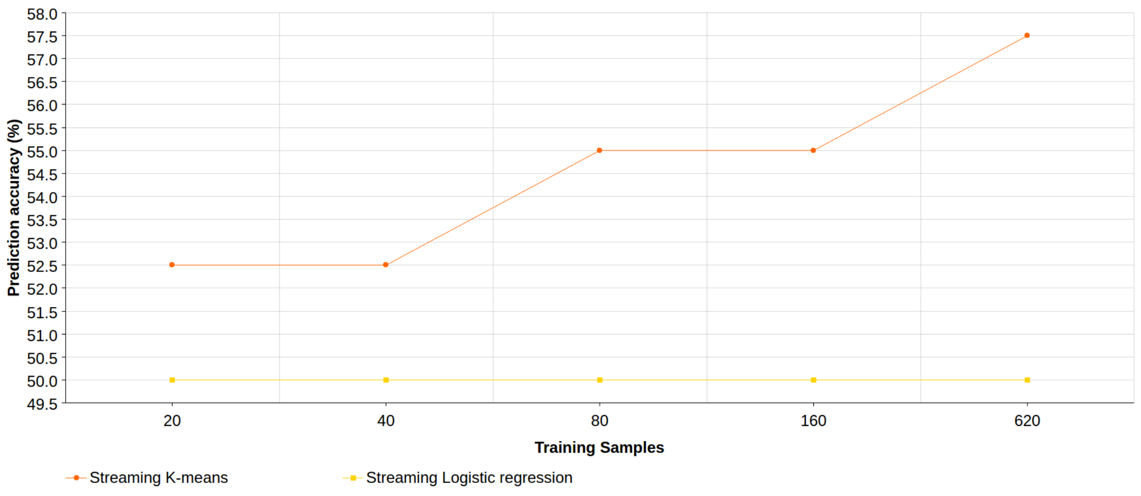

Figure 4.6: Prediction accuracy in relation to number of training samples for the Volvo data set.

Due to the low prediction accuracy for the Volvo data, we performed the same evaluation on the Statlog (German Credit Data) data set [5] in order to find out whether the problem was with the implementation or the initial data set. This data set contains 24 features and 1000 samples and we performed the same test as for the Volvo data set, i.e 10-fold cross validation with increasing amount of training data for each training iteration and 10% validation data. This data set was only tested with the logistic regression algorithm with the same model parameters as for the Volvo data set. The results when using the Statlog data set are presented in Table 4.5 and visualized in Figure 4.7.

Table 4.5: Prediction accuracy in relation to number of training samples for the Statlog data set.

Training samples	Average prediction accuracy
10	59%
60	74%
260	76.66%
460	77.33%

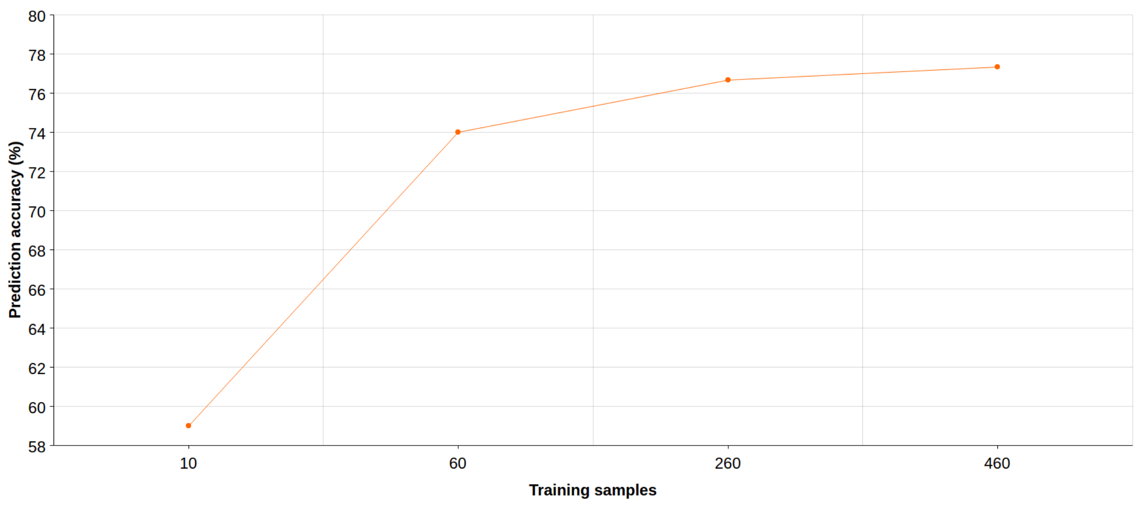

Figure 4.7: Prediction accuracy in relation to number of training samples for the Statlog data set.

5
Discussion and Conclusion

The main goal in this thesis was to propose a new concept for cloud-based processing of vehicle data on a remote machine. The motivation is that an implementation of this concept does not require any vehicle software re-configuration once the software is installed. The idea is that each implementation is unique in its behaviour while the concept components are not modified. We evaluated an implementation of the proposed concept-architecture would perform with respect to throughput and latency. The implementation was made using state-of-the-art streaming components, namely Apache Spark Streaming and Apache Kafka, and our implementation shows that it is possible to collect vehicle data that is further processed on a remote machine using stream processing. We have also showed that it is possible to implement the proposed concept using machine learning to perform anomaly detection on data streams. As mentioned in the related work, Section 2.6, the prediction accuracy of the machine learning techniques is highly related to the amount of data that is available and also the quality of the data. The more data we can use for learning the characteristics of unplanned stops, the better predictions. The proposed concept enables two-way communication between a vehicle and a remote machine, which opens up many possibilities and new ways of working with predictive maintenance. The driver could for example engage in the process by following instructions sent from RSS, like pumping up the air break pressure to a certain level. The signals that represents the air break pressure signals would then be sampled and sent to RSS.

We presented a use case where the goal was to predict air compressor faults with both unsupervised and supervised machine learning techniques using the same data set. We also saw how two different data sets gave completely different results as the models were more and more trained. Not only are the model parameters important to get as high prediction accuracy as possible, but also the data itself. The resulting prediction accuracy for the Volvo production data set was in the best case slightly above 50%, which is not considered as reasonable for vehicles in production. A possible reason for this result is that the Volvo data set was imbalanced, i.e. the rows in the training data with the label "faulty" were not as many as the rows labeled "non-faulty". This is a natural behaviour for vehicles since they are more likely to be in a healty state than in a faulty state. In a complex mechatronic system such as a transport vehicle, identifying and mapping vehicle parameter values with an actual labeled fault is not a trivial task. There are many parameters that can be associated with an air compressor fault and in order to identify these, humans with expert knowledge are required. However, it is hard for humans to keep up with the digitised systems due to increased complexity and big data amounts in the vehicles. With our concept, these experts would instead implement their models, let the concept components do the processing and learning and then the output could be observed.

Recalling Equation 4.1 and Equation 4.2, we know that the total RSS latency is strictly related to the processing throughput of the stream processing engine and since our implementation operates with micro batching, each Spark Streaming application needs to be initialized with a batch interval which adds a constant value to the processing latency. Therefore the ideal case would be if a test were processed in only one micro batch so that the batch interval latency only gets added to the total latency once. However, if we increase the batch interval we also increase the total latency.

For one batch, Spark Streaming can only consume $7500 * batch\_interval(s)$ for the hardware setup used in this work. This means that if the batch interval is for example 1 second, the stream processing engine would need to two micro-batches for 13.000 tuples. For every new micro batch, overhead is added in terms of scheduling and memory mapping and therefore Kafka is the bottleneck for total latency.

For our implementation, we evaluated the processing throughput for two different architectures; the bypassed and the full architecture. The results showed clear advantages in terms of throughput and latency with the bypassed architecture, i.e by sending data directly from VS to the data pipeline. The main reason for this is the memory overhead that is created for each connection between VS and the RSS web server. The bypassed architecture did not meet our goals with low latency and high throughput processing. This issue is of course strictly related to the hardware of the remote machine but the bypassed architecture would in any case outperform the full architecture. However, we believe that the remote web server is a valuable concept component with other purposes than to pass data. For example, the remote web server application can be used as a supervisor between VS and the Spark Streaming application.

The streaming components Spark Streaming and Kafka both provides fault tolerant mechanisms for handling loss of data. Our experiment did not cover the fault tolerance aspect, but we observed that given the concept components, fault tolerance and data loss prevention can be achieved, if the application area requires it.

We observed that guaranteeing consistency when parallelizing the data processing using Kafka was challenging when we used four partitions and four cores. This needs to be further investigated in future work since it is vital to parallelize the computations when working with distributed systems.

5.1 Future work

In production, an implementation of the proposed concept would need an administrative web interface that can monitor the different components. Moreover, we would also need a database that keeps track of which Kafka topics are used for a certain Spark Streaming application. This database would be used mostly by the RSS web application in order to look up which topics and Spark Streaming application that belongs to a certain machine learning model. Assuming that we have an administrative interface where all the vehicles for a specific population are displayed, we would also want to retrieve data from this database regarding available signals and already created tests in order to re-use them.

Supervised and unsupervised machine learning techniques can be used to find potential anomalies in vehicle data. When the vehicle data is suspected to be anomalous, different in-vehicle diagnostics tests can be initialized from RSS since the concept enables two-way communication. We think that these detailed and in-depth diagnostics tests that are executed in the vehicle will have a better accuracy than what would be possible using classification.

The prediction accuracy was not satisfying for the area of predictive maintenance. Future work in this area would require further research in balancing the data set and how features should be chosen to be able to predict anomalies in vehicle data. Moreover, future research would need to study how the machine learning algorithms respond to for example up-sampling the data set in order to balance the data set or maybe using algorithms that are not that sensitive to unbalanced data sets.

We tested the implementation of VS on a simulated hardware and the software was never integrated with a testing vehicle environment or a real vehicle. It would be interesting to further research in the vehicles capabilities of sending data and how to compress that data efficiently in order to reduce network latency. Moreover, the work in this thesis did not include the security aspect. Messages exchanged by VS and RSS are not encrypted and neither the VS or RSS validates the source of incoming messages. Also, internal errors in the VS must not affect any other software in a vehicle and would therefore be preferred to run in a sand-boxed environment. This thesis also leaves room for further research in how to parallelize the stream processing engine and how to distribute the components on a powerful cluster.

5.2 Conclusion

The proposed concept shows that it is possible to collect vehicle data that is further processed on a remote machine using stream processing. Our results show that it is possible to create machine learning models that continuously evolves and learns from data streams. However, a system that detects anomalies in vehicle data would require at least 99% prediction accuracy, which we were not able to show with the Volvo data set. The Volvo data set used in this work was not suitable for real-time anomaly detection. With implementations that provides at least 99% prediction accuracy, the proposed concept can be used to detect anomalies in vehicle components remotely without re-configuring any software inside the vehicles.

Bibliography

[1] Ahmad, S., Purdy, S. (2016). Real-Time Anomaly Detection for Streaming Analytics. CoRR, abs/1607.02480

[2] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I. (2013). Discretized Streams: Fault-Tolerant Streaming Computation at Scale. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles pp. 423-438

[3] Apache Kafka. (2017). Apache Kafka. [online] Available at: https://kafka.apache.org/ [Accessed 28 Aug. 2017].

[4] Appstate.edu. (2017). [online] Available at: http://www.appstate.edu/ whiteheadjc/service/logit/intro.htm#logitmodel [Accessed 4 Jun. 2017].

[5] Archive.ics.uci.edu. (2017). UCI Machine Learning Repository: Statlog (German Credit Data) Data Set. [online] Available at: https://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data) [Accessed 18 Jun. 2017].

[6] Arindam, B., Chandola, V., and Vipin, K. "Anomaly Detection". ACM Computing Surveys 41.3 (2009): pp. 1-58. Web. 18 June 2017.

[7] Autosar.org. (2017). AUTOSAR: Home. [online] Available at: https://www.autosar.org/ [Accessed 16 Mar. 2017].

[8] Byttner, S., Rögnvaldsson, T., Svensson, M. (2011). Consensus self-organized models for fault detection (COSMO). Engineering Applications of Artificial Intelligence, 24(5), pp. 833-839

[9] Cederman, D., Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P. (2016) Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types. CoRR, abs/1606.04746

[10] Spark.apache.org. (2017). Clustering - RDD-based API - Spark 2.2.0 Documentation. [online] Available at: https://spark.apache.org/docs/latest/mllib-clustering.html [Accessed 28 Aug. 2017].

[11] "Ericsson Mobility Report: 70 Percent Of World's Population Using Smartphones By 2020". Ericsson.com. N.p., 2017. Web. 18 June 2017.

[12] GitHub. (2017). cogu/apx. [online] Available at: https://github.com/cogu/apx [Accessed 30 Mar. 2017].

[13] Holehouse.org. (2017). 06_Logistic_Regression. [online] Available at: http://www.holehouse.org/mlclass/06_Logistic_Regression.html [Accessed 18 Jun. 2017].

[14] Iris Data Set. [online] Available at: https://archive.ics.uci.edu/ml/datasets/iris [Accessed 28 May 2017].

[15] Leskovec, J., Rajaraman, A. and Ullman, J. (2014). Mining of massive datasets. Cambridge University Press.

[16] Prytz, R. (2014). Machine learning methods for vehicle predictive maintenance using off-board and on-board data. CoRR, abs/1710.06839

[17] Prytz, R., Nowaczyk, S., Rögnvaldsson, T. and Byttner, S. (2013) Analysis of truck compressor failures based on logged vehicle data. International Conference on Data Mining (DMIN13)

[18] Rettig, L., Khayati, M., Cudré-Mauroux, P., Piórkowski, M. "Online Anomaly Detection Over Big Data Streams". 2015 IEEE International Conference on Big Data (Big Data) (2015): np. Web. 18 June 2017.

[19] Samuel, A. (1959). Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of Research and Development, 3(3), pp. 210-229

[20] Sites.google.com. (2017). k-means clustering algorithm - Data Clustering Algorithms. [online] Available at: https://sites.google.com/site/dataclusteringalgorithms/k-means-clustering-algorithm [Accessed 28 Aug. 2017].

[21] Spark.apache.org. (2017). Apache Spark. [online] Available at: http://spark.apache.org/ [Accessed 30 Mar. 2017].

[22] Soni, J., Prabakar, N. and Kim, J-H. (2017) Prediction of Component Failures of Telepresence Robot with Temporal Data. 30th Florida Conference on Recent Advances in Robotics

[23] Tomcat.apache.org. (2017). Apache Tomcat. [online] Available at: http://tomcat.apache.org/ [Accessed 31 Jul. 2017].

[24] Prytz, R., Nowaczyk, S., Rögnvaldsson, T. and Byttner, S. (2013). Towards a Machine Learning Algorithm for Predicting Truck Compressor Failures Using Logged Vehicle Data. Twelfth Scandinavian Conference on Artificial Intelligence pp. 205–214

[25] Prytz, R., Nowaczyk, S., Rögnvaldsson, T. and Byttner, S. (2015). Predicting the need for vehicle compressor repairs using maintenance records and logged vehicle data. Engineering Applications of Artificial Intelligence, 41, pp. 139-150

A

Appendix 1

A.1 Vehicle Software data

Example JSON data for the VS-Server /popup endpoint:

```
{
    "signals_provide": [
        {
        "name": "Signal1",
        "value": "4",
        "num_samples": "1",
        "sample_rate_ms": 300
        }
        .
        .
        .
        {
        "name": "Signal2",
        "value": "1",
        "num_samples": "1",
        "sample_rate_ms": 300
        }
    ]
}
```

Example JSON data for the VS-Server /client endpoint:

```
{
    "signals_require": [
        {
        "name": "Signal1",
        "num_samples": "5",
        "sample_rate_ms": 300
        }
        .
        .
        .
        {
        "name": "Signal2",
        "num_samples": "5",
        "sample_rate_ms": 500
        }
    ]
}
```

A.2 Remote Server Software data

Example JSON data that is sent to RSS from a VS-Client after being sampled:

```
{
    "data": [
        {
            "signal_name": "Signal1",
            "samples": [
                {
                    "timestamp": "2017-05-05 10:00:00",
                    "value": 32
                },
                {
                    "timestamp": "2017-05-05 10:00:20",
                    "value": 50
                }
            ]
        },
        {
            "signal_name": "Signal2",
            "samples": [
                {
                    "timestamp": "2017-05-05 11:10:00",
                    "value": 10
                },
                {
                    "timestamp": "2017-05-05 11:11:00",
                    "value": 50000
                }
            ]
        }
    ],
    "vehicle_ip": "192.0.0.1"
}
```

A.3 Vehicle Software Server endpoint functions

```
@require_http_methods(["POST"])
@csrf_exempt
```

```
def spawn_vsclient_popup(request):
    body_unicode = request.body.decode('utf-8')
    body = json.loads(body_unicode)
    popupmanager.PopupManager(body)

    return HttpResponse("OK")

@require_http_methods(["POST"])
@csrf_exempt
def spawn_vsclient(request):
    body_unicode = request.body.decode('utf-8')
    body = json.loads(body_unicode)
    content = body

    rss_ip_port = content.get('rss_ip')
    signals_provide = content.get('apx_signals_provide') or []
    signals_require = content.get('apx_signals_require') or []
    apx_node_name = content.get('apx_node_name')

    apx_signals_provide = []
    apx_signals_require = []

    #These provide signals are only used for simulating signals.
    for signal in signals_provide:
        apx_signal = {}
        apx_signal['name'] = signal['name']
        apx_signal['name_signature'] = '"'+signal['name']+'"'+signal['signature']
        apx_signal['sample_rate_ms'] = signal.get('sample_rate_ms') or 0
        apx_signal['num_samples'] = signal.get('num_samples') or 0
        apx_signals_provide.append(apx_signal)

        vsc = vsclientprovide.VSClient("Node"+str(randint(1,10000)), apx_signal)
        t = Thread(target = vsc.run, args = ())
        t.start()

    for signal in signals_require:
        apx_signal = {}
        apx_signal['name'] = signal['name']
        apx_signal['name_signature'] = '"'+signal['name']+'"'+signal['signature']
        apx_signal['sample_rate_ms'] = signal.get('sample_rate_ms') or 0
        apx_signal['num_samples'] = signal.get('num_samples') or 0
        apx_signals_require.append(apx_signal)

    if len(signals_require) > 0:
        vsc = vsclientreq.VSClient(apx_node_name, apx_signals_require, rss_ip_port)

    return HttpResponse("OK")
```

A.4 Vehicle Software PopupManager class

```python

class PopupManager():

  def __init__(self, content):
    self.node = apx.Node(content['node_name'])
    node_name = content['node_name']
    signal_name = content['name']
    signal_signature = content['signature']
    signal_default_value = "=0"
    signal_value = content['value']

    self.node.append(apx.RequirePort('require','C'))
    self.node.append(apx.ProvidePort(signal_name,
    signal_signature, signal_default_value))

    self.apx = apx.Client(self.node)
    if self.apx.connectTcp('127.0.0.1', 5000):
      print("Successful connect")
      self.apx.write_port(signal_name, signal_value)
      self.apx.stop()
      print("Successful popup")
```

A.5 Vehicle Software VSClient class

```python
@apx.DataListener.register
class VSClient(apx.DataListener):
  def __init__(self, node_name, signals_require, rss_ip_port):
    self.name = node_name
    self.has_listener = True
    self.samples = []
    self.samples_counter = 0
    self.node = apx.Node(node_name)
    self.send_rate_ms = None
    self.signals_require = signals_require
    self.rss_ip_port = rss_ip_port

    for signal_require in self.signals_require:
        signal_require['samples_counter'] = 0
        self.node.append('R' + signal_require.get('name_signature'))

    self.apx = apx.Client(self.node)
    self.lock = threading.Lock()

    if len(signals_require) > 0 and self.has_listener:
        print("Adding listener")
        self.apx.set_listener(self)

    if self.apx.connectTcp('127.0.0.1', 5000):
      print("Successful connect")

  def is_sample_limit(self, signal_counter, signal_num_samples):
      if signal_counter >= signal_num_samples:
          return True
      else:
          return False

  def all_samples_done(self):
      for signal_require in self.signals_require:
          if signal_require.get('num_samples') >
           signal_require.get('samples_counter'):
             return False
      return True


  def add_sample(self, data, signal_name):
      sample = {}
      sample['time_stamp'] = time.strftime("%Y-%m-%d %H:%M:%S")
      sample['value'] = data
      sample['name'] = signal_name
      self.samples.append(sample)
      print("Adding %s to samples with value: %s"%(signal_name, str(data)))

  def on_data(self, port_id, port_name, data):
      print("Recieved data from: %s with value: %s"%(port_name, str(data)))

      for signal_require in self.signals_require:
         if signal_require.get('name') == port_name:
            current_signal = signal_require
            break
      if self.all_samples_done():
         print("Test complete!")
         #self.apx.stop()
         remote_obj = {}
         remote_obj['data'] = self.samples
         remote_obj['vehicle_ip'] = socket.gethostbyname(socket.gethostname())
         self.send_data_remote(remote_obj)

         raise SystemExit
      else:
         current_signal_is_limit = self.is_sample_limit(
           current_signal.get('samples_counter'), current_signal.get('num_samples'))
         if current_signal_is_limit is False:
            if current_signal.get('timer') is not None and
               current_signal[’timer’].isAlive():
              print("Timer alive, waiting...")
              #self.timer.join()
            else:
              print("Starting timer...")
              current_signal[’samples_counter’] = current_signal.get(’samples_counter’) +
              current_signal[’timer’] = Timer(int(current_signal[’sample_rate_ms’])/1000,
              self.add_sample, (data, port_name))
              current_signal[’timer’].start()


  def send_data_remote(self, data):
      r = requests.post("http://"+self.rss_ip_port+"/ProcessIncoming", json=data)
      print(r.status_code, r.reason)
```

A.6 Vehicle Software Data Generation

```
    def generate_and_send(self):
        for _ in range(self.num_samples):
            data = []
            for i in range(dimensions):
                data.extend([randint(1, 255)])
            data_to_string = ’, ’.join([str(x) for x in data])
            data_to_string_array = ’[’ + data_to_string + ’]’
            try:
                time.sleep(self.signal_provide.get(’sample_rate_ms’)/1000)
                self.apx.write_port(self.signal_provide.get(’name’), data_to_string_array)
```

A.7 Vehicle Software Client creation

```
    def __init__(self, node_name, signal_provide):
        self.name = node_name
        self.node = apx.Node(node_name)
        self.send_rate_ms = None
        self.signal_provide = signal_provide
        self.num_samples = signal_provide[’num_samples’]
        self.num_dimensions = signal_provide[’num_dimensions’]
        self.apx = apx.Client(self.node)

        self.apx.connectTcp(’127.0.0.1’, 5000)

        if signal_provide:
            self.node.append(’P’ + signal_provide.get(’name_signature’))
            self.node.append(’R’ + ’"require"C’)
        else:
            self.node.append('R' + signal_provide.get('name_signature'))
            self.apx.set_listener(self)
```

A.8 Spark Streaming Application Code - Word occurrences

```
import kafka.serializer.StringDecoder
import org.apache.spark.{SparkConf}
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.kafka._
import java.util.Properties


object Streamer {

  def Streamer(kafkaTopic: String, batchInterval: Int, nrCores: String, appName: String)
    val conf = new SparkConf().setAppName(appName)
    conf.setMaster(nrCores)
    val ssc = new StreamingContext(conf, Seconds(batchInterval))

    val kafkaTopicSet = Set(kafkaTopic)

    val kafkaBrokers = "localhost:9092"
    val kafkaParams = Map[String, String]("metadata.broker.list" -> kafkaBrokers)

    val kafkaProperties = new Properties()
    kafkaProperties.put("bootstrap.servers", kafkaBrokers)
    kafkaProperties.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer")
    kafkaProperties.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer")

    val kafkaMessages = KafkaUtils.createDirectStream[String, String, StringDecoder,
    StringDecoder](
      ssc, kafkaParams, kafkaTopicSet)

    kafkaMessages.foreachRDD( rdd => {
      rdd.collect().foreach(tuple => {
        var occurenceMap:scala.collection.mutable.Map[String,Int] = scala.collection
        .mutable.Map()

        val featureArray = tuple._2.split(",")
        featureArray.foreach(v => {

          if (occurenceMap.get(v).isEmpty) {
            occurenceMap += v -> 1
          } else {
            occurenceMap(v) = occurenceMap(v) + 1
          }
        })
      })
    })

    ssc.start()
    ssc.awaitTermination()
  }

  def main(args: Array[String]): Unit = {
    Streamer(args{0}, Integer.parseInt(args{1}), args{2}, args{3})
  }
}
```

A.9 Spark Streaming Application Code - Streaming K-Means

```
import kafka.serializer.StringDecoder
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.kafka._
import java.util.Properties

import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}
import org.apache.spark.mllib.clustering.StreamingKMeans
import org.apache.spark.mllib.regression.LabeledPoint

object Streamer {

  def Streamer(kafkaTopicTraining: String,
               kafkaTopicTesting: String,
               kafkaTopicVisualization: String,
               slidingWindow: Int,
               nrDimensions: Int,
               nrCores: String,
               appName: String,
               stepSize: String,
               k: Integer) {

    val conf = new SparkConf().setAppName(appName)
    conf.setMaster(nrCores)
    val ssc = new StreamingContext(conf, Seconds(slidingWindow))
    var vehicleIp = ""

    val trainingTopic = Set(kafkaTopicTraining)
    val testingTopic = Set(kafkaTopicTesting)

    val kafkaBrokers = "localhost:9092"
    val kafkaParams = Map[String, String]("metadata.broker.list" -> kafkaBrokers)

    val kafkaProperties = new Properties()
    kafkaProperties.put("bootstrap.servers", kafkaBrokers)
    kafkaProperties.put("value.serializer",
      "org.apache.kafka.common.serialization.StringSerializer")
    kafkaProperties.put("key.serializer",
      "org.apache.kafka.common.serialization.StringSerializer")

    val kafkaProducer = new KafkaProducer[String, String](kafkaProperties)

    val kafkaMessagesTesting = KafkaUtils.createDirectStream[String, String,
      StringDecoder,
      StringDecoder](
      ssc, kafkaParams, testingTopic)

    val kafkaMessagesTraining = KafkaUtils.createDirectStream[String, String,
      StringDecoder,
      StringDecoder](
      ssc, kafkaParams, trainingTopic)

    val testData = kafkaMessagesTesting.map(_._2).map(x => LabeledPoint.parse(x))
    val trainingData = kafkaMessagesTraining.map(_._2).map( x => LabeledPoint.parse(x))

    kafkaMessagesTesting.foreachRDD(x => {
      if (x.count() > 0) {
        vehicleIp = (x.collect()(0)._2 split "-" take 2)(1)
      }
    })

    val model = new StreamingKMeans()
      .setK(k)
      .setDecayFactor(1.0)
      .setRandomCenters(nrDimensions, 0.0)

    model.trainOn(trainingData.map(p => p.features))
    val mdl = model.predictOnValues(testData.map(lp => (lp.label, lp.features)))

    mdl.foreachRDD { rdd =>

      if (rdd.count() > 0) {
        val prediction = rdd.collect()(0)._1
        if (prediction == 1.0) {
          val record = new ProducerRecord("anomaly", "key", vehicleIp)
          kafkaProducer.send(record)
        }
      }
    }

    kafkaProducer.close()

    ssc.start()
    ssc.awaitTermination()
  }

  def main(args: Array[String]): Unit = {
    Streamer(
      args{0},
      args{1},
      args{2},
      Integer.parseInt(args{3}),
      Integer.parseInt(args{4}),
      args{5},
      args{6},
      args{7},
      Integer.parseInt(args{8})
    )
  }
}
```

A.10 Spark Streaming Application Code - Streaming Logistic Regression

```
import kafka.serializer.StringDecoder
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.kafka._
import java.util.Properties

import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}
import org.apache.spark.mllib.classification.StreamingLogisticRegressionWithSGD
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint

object Streamer {

  def Streamer(kafkaTopicTraining: String,
               kafkaTopicTesting: String,
               kafkaTopicVisualization: String,
               slidingWindow: Int,
               nrDimensions: Int,
          nrCores: String,
          appName: String,
          numIterations: Integer,
          stepSize: Double) {

val conf = new SparkConf().setAppName(appName)
conf.setMaster(nrCores)
val ssc = new StreamingContext(conf, Seconds(slidingWindow))
var vehicleIp = ""
val initialWeightsVector = Vectors.zeros(nrDimensions)

val trainingTopic = Set(kafkaTopicTraining)
val testingTopic = Set(kafkaTopicTesting)

val kafkaBrokers = "localhost:9092"
val kafkaParams = Map[String, String]("metadata.broker.list" -> kafkaBrokers)

val kafkaProperties = new Properties()
kafkaProperties.put("bootstrap.servers", kafkaBrokers)
kafkaProperties.put("value.serializer",
  "org.apache.kafka.common.serialization.StringSerializer")
kafkaProperties.put("key.serializer",
  "org.apache.kafka.common.serialization.StringSerializer")

val kafkaProducer = new KafkaProducer[String, String](kafkaProperties)

val kafkaMessagesTesting = KafkaUtils.createDirectStream[String, String,
  StringDecoder,
  StringDecoder](
  ssc, kafkaParams, testingTopic)

val kafkaMessagesTraining = KafkaUtils.createDirectStream[String, String,
  StringDecoder,
  StringDecoder](
  ssc, kafkaParams, trainingTopic)

val testData = kafkaMessagesTesting.map(_._2).map(x => LabeledPoint.parse(x))
val trainingData = kafkaMessagesTraining.map(_._2).map( x => LabeledPoint.parse(x))

kafkaMessagesTesting.foreachRDD(x => {
  if (x.count() > 0) {
    vehicleIp = (x.collect()(0)._2 split "-" take 2)(1)
  }
})

val model = new StreamingLogisticRegressionWithSGD()
  .setInitialWeights(initialWeightsVector)
  .setNumIterations(numIterations)
  .setStepSize(stepSize)
```

```

    model.trainOn(trainingData)
    val mdl = model.predictOnValues(testData.map(lp => (lp.label, lp.features)))

    mdl.foreachRDD { rdd =>

      if (rdd.count() > 0) {
        val prediction = rdd.collect()(0)._1
        if (prediction == 1.0) {
          val record = new ProducerRecord("anomaly", "key", vehicleIp)
          kafkaProducer.send(record)
        }
      }
    }

    kafkaProducer.close()

    ssc.start()
    ssc.awaitTermination()
  }

  def main(args: Array[String]): Unit = {
    Streamer(
      args{0},
      args{1},
      args{2},
      Integer.parseInt(args{3}),
      Integer.parseInt(args{4}),
      args{5},
      args{6},
      Integer.parseInt(args{7}),
      args{8}.toDouble
    )
  }
}
```