CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# SSPOC: Smart Stream Processing Operator Classification

Master's thesis in Computer Systems and Networks

Victor Gustafsson, Hampus Nilsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

# SSPOC: Smart Stream Processing Operator Classification

Victor Gustafsson
Hampus Nilsson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

SSPOC: Smart Stream Processing Operator Classification
VICTOR GUSTAFSSON
HAMPUS NILSSON

Supervisor: Vincenzo Gulisano, Department of Computer Science and Engineering
Advisor: Karl Bäckström, Department of Computer Science and Engineering
Examiner: Marina Papatriantafilou, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

SSPOC: Smart Stream Processing Operator Classification
VICTOR GUSTAFSSON
HAMPUS NILSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

## Abstract

Stream Processing is a rapidly growing field. Efficiently handling a stream processing query often requires knowing what type each operator is, as knowing its behaviour allows for tailored solutions. Today, each framework handles the identification of operators in its own way, often using semantics and compile-time info for this purpose. Having a more general way of classification could be an interesting way to simplify the creation of such framework. Creating such a general way requires a change from semantic info, as different frameworks use different semantics, to more general information. We pioneer a first step in this direction by using metrics available at runtime to classify a basic set of operators.

In this thesis, we present a machine learning model for classification of stream processing operators. The model is a densely connected multi-layer feed-forward neural network. The operators that are classified are limited to a subset of the standard set of operators available in the stream processing framework Apache Flink. The training, validation and test datasets are also a contribution of this thesis. These were collected from public queries using our collection method. We also propose a set of features for our classifier, that aid in differentiating operators; we suggest that other machine-learning based solutions can use them.The model is optimized for prediction accuracy while training on data collected from 9 different queries. It reaches a prediction accuracy of 97.51% on the validation dataset and 99.796% on the test dataset.

Keywords: Computer, science, computer science, engineering, project, thesis, machine learning, neural networks, stream processing

# Acknowledgements

We would like to express our gratitude for the expertise and assitance given to us by Karl Bäckström and Marina Papatriantafilou during the course of this thesis. We would also like to extend a special thanks to Vincenzo Gulisano, our supervisor who has been on our side all the way through and has provided us with valuable feedback and direction. It would not have been possible without you.

Hampus Nilsson & Victor Gustafsson, Gothenburg, March 2019

# Contents

# List of Figures

# List of Tables

w

# 1
# Introduction

As more data is generated and becoming available to us every day, tools are required to process and analyze this data. Today, the business logic of many applications require them to provide critical information in real-time, as is the case with autonomous cars and pricing of flight tickets, among others things. When combining these requirements with the large volumes of data that have to be analyzed, solely relying on offline data processing is no longer an option. This trend has given rise to a new paradigm in computer science, which is called Stream Processing. Stream Processing is a *big data* technology, that allows for querying streams of data and processing it before it is stored. The demand for this type of processing has lead to the introduction of multiple Stream Processing frameworks, such as Apache Storm [1] and Apache Flink [2]. A Stream Processing program, also known as a query, consists of an acyclic graph of processing nodes, that perform transformations on a stream of data and outputs a result. These processing nodes are commonly known as *operators*. With frameworks, much of the complexity of Stream Processing is abstracted away, simplifying the process of creating these analyzing queries. They offer automatic features for aspects such as parallelizing and scaling, they also provide strong flexibility for the developers. For instance, they allow for extension of operators that are pre-defined in the framework, and also the creation of custom ones, which opens up a wide range of possible use-cases. This flexibility, especially in the case of custom operators, may however prevent the framework from knowing the exact semantics of an operator, how it behaves and how it should be handled. This can be an issue, since knowing what type of transformation an operator performs allows for the possibility of more efficient execution, as the framework can then make informed decisions on how to manage these operators. For instance, some operators can be scaled and have multiple copies of itself each taking care of a part of the stream, executing in parallel. Others require the entire stream to pass through a single instance in order for a query to return a correct result.

Knowing what type of transformation an operator performs is not always straightforward, as while there is a reoccurring set of "basic" operators in Stream Processing, a user constructing their own operators might combine the semantics of several operators inside a single operator.There is no standardized way of deciding the type of an operator between frameworks, meaning that each framework uses its own method of inferring the operator type. As Stream Processing frameworks exist in different programming languages and for different platforms, any method aspiring to find the type of an operator, and also work for different frameworks, needs to do so using information that is not specific to any one framework. We hypothesize that it is possible to determine the type of an operator using run-time information that should be available while executing any Stream Processing query, such as metrics and connections between operators. In order to facilitate such a run-time classification, we propose a method of finding the type of an operator using general run-time information that is fed to a classifier created using Machine Learning (ML) techniques.

## 1.1 Project Aim

The objective of this thesis is to investigate whether it is possible to use Machine Learning to classify Stream Processing operators, based only on general run-time metrics that keep the anonymity of the data being processed. The lack of available data to use for this purpose means this thesis also deals with choosing and extracting metrics from public stream processing queries. Data will also be extracted from personally created queries, these queries are designed and implemented by us, the authors, rather than someone else.

## 1.2 Limitations

This thesis is limited to exploring the use of the Machine Learning concept Neural Networks to classify Stream Processing operators, based on metrics these operators generate at run-time. The main focus is to optimize the classification accuracy of the model. We do not focus on other aspects, such as performance, although they may still be noted and discussed. Only classification of operators present in the queries utilized in order to gather training data are considered in the thesis.

# 2 Background

The following sections provide background information necessary in order to follow the discussions and reasoning in later chapters.

## 2.1 Stream Processing

Stream Processing is a paradigm that is suited for application in areas where large amounts of data need to be analyzed efficiently. The term was popularized by Apache Storm [3], and although similar ideas had existed under other terms, such as *event processing* or streaming analytics, they have now mostly converged under the name of Stream Processing. The paradigm is well suited to near real-time fields such as sensor data analysis and financial analysis. The strength of Stream Processing lies in the fact that instead of storing data and then processing it, like in traditional databases, SPE:s instead process data on-the-fly. This removes the need to handle storage of the massive data amounts, and also removes the necessity of writing the data to disk, which is magnitudes slower than analyzing it in memory.

In Stream Processing, a data stream is defined as a potentially infinite sequence of tuples, all structured according to the same schema. A tuple schema is denoted as $(A_1, A_2, A_3, ..., A_n)$, where $A_i$ is a generic attribute of the tuple which for tuple $t$ is denoted as $t.A_i$. The schema defines the types of the attributes and often also a name. The types are generally common datatypes such as *string*, *double*, *time*, *integer* etc.. Most tuples have an external timestamp, or one of the generic attributes $A_i$ can be interpreted as one. In cases which this is not true SPEs generally assign tuples a timestamp based on the "time of arrival" into the system.

*Queries* that are used in SPEs are defined as "continuous" as they are performed on a continuous stream and also continuously push results to the user as the query predicates are fulfilled. A query, in this case, is a directed acyclic graph where each edge is a data stream, and each node is an operator. Operators transform an incoming stream into a new outgoing stream(s). The typical operators can be seen as analogous to the normal relational algebra operators, with for instance *filter* being analogous to *select*. Operators will be described in 2.1.1. The operators are distringuished as either stateful or stateless. The stateless operators keep, as the name suggests, no state, which means the output is solely based on the input tuple. Stateful operators keep information from tuple to tuple, in essence performing their computations on a sequence of tuples. Because a stream is theoretically unbounded, stateful operators perform their computations on windows of tuples. These windows can either be sliding count-based (e.g. the last 100 tuples) or sliding time based (e.g. all tuples received the last minute). Specific versions of these windows are tumbling, where the window size is equal to the window sliding interval, and jumping windows, where the sliding interval is larger than the size of the window. The numbers in the examples, namely the time or the amount of tuples, are parameters for the window. A very simple example query with some common operators can be seen in Figure 2.1.

**Batch Processing**

While Stream Processing is necessary for real-time analysis of large continuously incoming volumes of data, it does not replace offline data analysis (analyzing data stored on disk). Analyzing data collected over longer periods of time is often an additional requirement to analyzing it in real-time, therefore storing and analyzing it later is still necessary sometimes. For this purpose one can use what is known as *batch processing*, which has higher performance when analyzing data that is already available. Batch processing is essentially analyzing a lot of data that is already in one's possession, such as data stored on disk, in one batch. This is in contrast to Stream Processing, which is analyzing a continuous stream of data. Methods for analyzing data that are different from methods used in Stream Processing may be used, as they can be more effective in this case.

### 2.1.1 Operators

An operator is as mentioned previously an entity that performs a transformation on a data stream. According to IBM there are three overarching types of operators [4]:

- Source Operator - An operator that takes external data and creates a stream that it then feeds to following operators
- Processing Operator - An operator that takes in a data stream, transforms it and then feeds it downstream. These operators are the ones mentioned previously that can be likened to relational algebraic operators.
- Sink Operator - An operator that writes its input data to external systems.

There are many different operators, and although there are some operators considered *basic*, available operators vary depending on which SPE is utilized. The frameworks that act as SPEs have their own set of operators, and also their own implementation. The ability for users to create own operators, also known as *custom* operators, is also present in most SPEs to handle cases the set of operators it provides is ill-suited for. The basic operators are usually enough, as they are all based on applying a function that the user has defined, meaning they are already able to handle most cases.

**Basic Operators**

There are a few operators that in general are present in most, if not all, SPE:s. We will here summarize which operators we consider *basic* and how they work. This is important as the basic operators are what we base the classifier classes on, and the way the operators work is the basis on any discussion regarding if features can distinguish between them. Here we provide intuitive descriptions for the operators, but a more in-depth description can be found in Appendix 2.

***Source***

The *Source* operator is just what the name suggest, an operator that somehow gets the data into the query. This could be read from a socket, a file or any other medium. *Source* generally convert the data it reads into an appropriate tuple schema for the query, removing extraneous things such as metadata. A formal function definition for the *Source* could not be found, however as its workings is intuitive and heavily based on implementation it is not a necessity.

***Sink***

The *Sink* operator is an endpoint to a query. *Sinks* generally connect the query to the "outside" world. This could mean for instance writing to databases, posting to websites/dashboards, directly sending results to APIs, connecting to elasticsearch etc.. This means that a *Sink* operator also needs to convert data from the tuple schema it receives, to some format that match the system it is interconnected to. Just like *Sources*, *Sinks* also lack a formal description due to its heavy reliance on implementation and also since it is not part of the actual computation.

***Map***

The *Map* operator is a generalized projection operator. It outputs one tuple for every incoming tuple, usually performing some transformation of the elements in the tuple. The incoming and outgoing schema may differ, however the outgoing tuple preserves the timestamp of the incoming one.

***FlatMap***

The *FlatMap* operator is an extended version of the Map operator. While the two are almost the same, the main difference is that a *FlatMap* outputs an arbitrary number of tuples for each incoming tuple. In practice, this means that *Map* operators can be considered a subset of the *FlatMaps. FlatMaps* vary somewhat based on implementation, but the usual behaviour is that 0, 1 or more output tuples are created based on each input tuple. Note that this could be 0 for the first tuple, then 4 for the next, meaning the *FlatMap* decides the number of output tuples based on the incoming one, rather than it being a chosen constant number.

***Filter***

The *Filter* operator is an operator that is used to either discard tuples based on some predicate, or split them into different outgoing streams. Each outgoing stream will be associated with one predicate, except a potential default one. Every incoming tuple is forwarded to the first stream whose associated predicate it satisfies. If a tuple matches none of the predicates, it is either routed to the default stream if it has been provided, or discarded entirely. A filter never alters the data in the tuples.

***Aggregate***

The *Aggregate* operator, as the name suggests, is used on windows of tuples to compute aggregate functions such as sums or averages. Tuples on the incoming stream are stored in the window until it is full. The window can be either time-based, which means the window is based on timestamps, or count-based, which means the window is based on tuple count instead. A time-based windows it is considered full if the difference in time between the incoming tuple and the first one in the window exceeds the configured values. When it is instead count-based a window is instead considered full if it contains the configured amount of tuples.

An Aggregate only produces output when the window is full (regardless of the type of window). The output tuples are sent over the output stream and use the timestamp of the **earliest** tuple in the window. The schema of the output tuple represents the set of user-defined functions (e.g. average, count, sum, etc.) that are computed over all the tuples in the current window.

The update of the window happens each time an output tuple is propagated. In this step all stale tuples are discarded from the window according to the parameter. For time-based windows a tuple is considered stale if its timestamp differs from a newly receieved timestamp by more than the size of the window. If the window is count-based, a configurable number of the earliest tuples will instead be considered stale.

***Join***
The *Join* operator is used to, as the name suggests, join multiple streams together. It has an output stream, and two input streams which are generally referred to as respectively *left* and *right*. The join operators also utilizes windows, however it keeps track of two separate windows, one for each of input streams. Tuples arriving on the *left* side are stored in the *left* window but are used to slide the *right* window, and vice versa for the *right* side. For time-based windows this means that on arrival of a tuple in the left window, the right window is updated by removing all tuples that are considered stale compared to the incoming one.

After the window has been updated, each tuple in the right window is concatenated with the incoming tuple which produces output tuples which, provided that a predicate is fullfilled, results in a positive outcome.

The updating of windows, evaluation of the predicate and propagation to the output for input tuples of the *right* stream are done in the same fashion.

***Reduce***
The *Reduce* operator continuously combine sequential tuples with the current result. It utilizes the same sort of windows (count-/time-based) as the other stateful operators. The reduce operator performs a set of functions that combine an incoming tuple with the current output value to produce a new output value. This is simple in theory, however it still utilizes windows, which means that the "current output" needs to be recalculated once tuples used to calculate it grow stale.

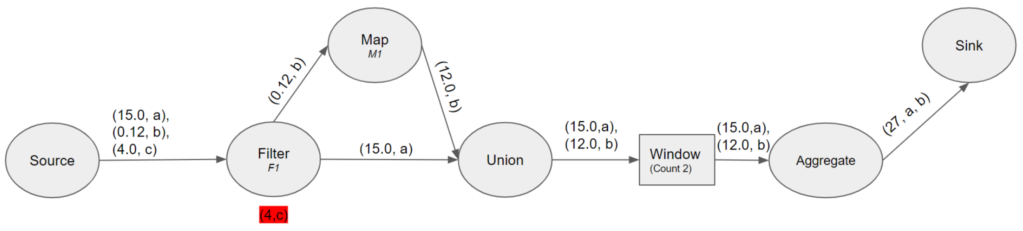


**Figure 2.1:** An example Stream Processing *query.* The source produces a stream with a schema of (value:double, system:char). $F1$ filters values from system b to $M1$, those from system a to $M1$, and drops data from system c. $M1$ converts the value from system b to match the way system a prints its values. A Union then merges the streams again. A window with a count of 2 puts both values in one window upon which the Aggregate is then applied. The aggregate sums the values, and also shows which systems these values originate from.

## Operator Properties

It can be seen in the description that operators have properties that depend on their type, they also have some that depend on their implementation, the query they are a part of, the environment they are run in etc.. The properties consist of more common ones like latency, processing time, parallelism etc. that could all be discussed in regards to most types of distributed programs, and but also some more defining ones. These properties can be used to derive the operator's type and are therefore important to the thesis. One such property which is of interest to this project is *selectivity*. *Selectivity* is a measurement on the output to input ratio of an operator. For example, if for every incoming tuple two tuples are output, the selectivity would be 2.

$$S(out, in) = out/in \tag{2.1}$$

It can be inferred that some operators, such as filters, do not have a constant selectivity, since it is based on the operators interaction with data, and data varies. For instance, in the case of filter, an entry could either be passed on, resulting in a selectivity of 1, or thrown away, resulting in a selectivity of 0. Therefore selectivity is often calculated using a statistical measure, such as an average over a period of time.

$$S(out, in, t) = \frac{1}{t}\int_0^t out/in, \mathrm{d}t \tag{2.2}$$

Another heuristic is to instead use an average over a set amount of tuples.

Additional properties to the ones just mentioned, that can be considered as more defining ones according to the operator definitions above, are:

1. Number of input streams - some operators only handle one stream
2. Number of output streams - some operators output to only a single stream(e.g. a Map), some to multiple (e.g. a Filter)
3. Memory usage - Stateful operators generally use more memory than stateless ones in order to keep their state

4. Tuple sizes - Differing input tuple and output tuple sizes indicate that an operator changed the data, which some operators never do (e.g. Filter). The size could be considered to be two different properties, one based on the size of the schema (the amount of entries in the tuple) and another based on the actual byte sizes of the tuple.

### 2.1.2 Steam Processing Engines

Stream Processing systems often use frameworks, or Stream Processing Engines (SPE:s) as they are also known as, such as Apache Flink [2] or Apache Storm [1] to ease development of these complex applications. These engines provide API:s for among other things, creating queries, streaming data and storing responses to queries without writing custom code. These frameworks automate the communication between operators, and also aid with different aspects including parallelization, fault-tolerance and state management. They are generally intended to work for any streaming application and therefore provide extensive customization options for the developers. Extension of the basic operators and also the creation of custom ones are possible, which, for the framework, complicates knowing the internal semantics, in turn making automatic optimization a hard task.

#### Apache Flink

Apache Flink [2] is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. [2] Although it has support for both batch processing, which is the standard way of processing where data is first stored and then later processed, and also Stream Processing, it can be considered as a Stream Processing Engine when run using its DataStream API rather than its DataSet API. Flink programs can be created using either Java or the script language *Scala*. More information regarding *Scala* can be found here [5]. Queries written using this framework are primarily intended to be placed on a networked cluster, but can also be run on a single machine.

Queries run using Flink are linked to a dashboard. This dashboard fetches data using a REST API built into the Flink queries, and this API is also available to external applications meaning an arbitrary script can obtain exposed information regarding the query execution. Some metrics are present for any Flink query and users can also create custom metrics of different types that are also accessible through the same API. The metrics in the REST API exists in different scopes, or levels as it is also known, such as at "Task" level. The metrics at this level has its own values for each chain of operators in the query.

One feature that Flink has, that is of importance to this thesis, is automatic operator chaining. This allows Flink to automatically create chains of operators where it thinks it could improve performance. These chains are considered a single entity when being assigned as tasks, meaning they all end up at a single task node. A task can in this way be either an operator, or a chain of operators. This becomes a problem when extracting native Flink metrics, since values such as bytes in or bytes out only exist at the task level, resulting in a chain of operators only having a single entry for these. This prevents using the data to classify an operator, as it is not possible to separate these values into those applying to each operator in the chain. There is however an option to disable operator chaining entirely, allowing one to bypass this issue entirely. For more information on this or other Flink features we refer the reader to the Apache Flink web page [2].

## 2.2 Machine Learning

In this section we explain aspects of Machine Learning (*ML*) relevant to this thesis. We introduce a machine learning algorithm called "K-means Clustering", which will be utilized as a baseline comparison. We also more thoroughly explain a family of ML algorithms known as artificial neural networks (*(A)NNs*), which is the core of the method proposed by this thesis.
Machine Learning is a paradigm in computer science in which a general inductive process automatically builds a model. There are many approaches to this inductive process, such as neural networks, support vector machines etc. but this thesis will mainly discuss NNs and K-means clustering. A neural network can utilize a method known as *Supervised Learning*(SL). *SL* is a method where each point of training data comes as a pair consisting of input data and the expected output of the network. This has some inherent benefits since it allows the use of *backpropagation*(BP), which can be leveraged by other methods to find local minima, however it requires knowing the expected answer and manually labeling every data entry.

### 2.2.1 K-means Clustering

In this thesis we use K-means clustering as a baseline to compare to our custom classifier. K-means clustering is an unsupervised learning clustering algorithm, used to group unlabeled data into k clusters. The algorithm groups the data into k clusters, where each data point belongs to the cluster to which it is closest. The goal of the algorithm is to group the data points such that the average euclidean distances between each data point and the center of a cluster is minimized [6]. By using k-means clustering, one can build a notion of what underlying groups exists in the data. Once these groups have been found, subsequently collected data can directly be put into any of these groups. It works similar to a classification algorithm where each cluster corresponds to a class. How many clusters to use is often based on what is called the elbow principal. The elbow principal states that: there exists a point at which the average total distance between data points and clusters sees a sharp drop off in how much it decreases if another cluster is added. Plotting a graph over the relationship between the number of clusters and the average distance between cluster and data point will have the shape of an elbow, hence the name. [7].

The algorithm is defined in the following way [8]:

**Initialization**
Each centroid $c_i \in C$ is a assigned a random point in the data space.

**Data Assignment Step**
Each data point x is assigned to the cluster $c_i$ which is closest to it, from the collection of clusters C.

$$\operatorname*{argmin}_{c_i \in C} dist(c_i, x)^2 \tag{2.3}$$

**Centroid Update Step**
Each centroid is assigned a new position, based on the mean of the distances to all data points which belongs to it.

$$c_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i \tag{2.4}$$

The algorithm terminates when it converges and there are no changes to the position of the cluster centroids.

### 2.2.2 Artificial Neural Networks

A neural network is a computational model inspired by how neurons activate and fire in the brain. The original phrase neural network has nothing to do with computers, however its counterpart *Artificial Neural Networks (ANN)*, which are a computer adapted version of the neural networks, is often labeled as simply *neural network*.

A neural network consists of many interconnected computational units known as neurons, each producing a sequence of real-valued activations [9]. These neurons are generally organized into three different types of layers: input, output and hidden layers. The neurons in the input layer react to environmental changes, or as is usually the case, the input data. The neurons in the hidden layers instead react and value the activation's of neurons from previous layers which allows for learning non-linear casual relations between input and output. The output layer is where the activations of the neurons are measured and interpreted as the result of the network. The concept of *Learning* in such a network is to change how neurons activate, and the difficulty lies in *credit assignment* [9], namely knowing what to change in order to improve. For networks with a small number of layers, or *Shallow* networks, a method known as Back-propagation (BP) was developed for this purpose as early as the 1960s and 1970s, and is still effective today.

#### Multi Layer Feed-Forward Neural Networks

Multi layer feed-forward (MLF) neural networks consists of neurons organized into multiple layers. Any MLF network will have an input layer (the first layer), output layer (the last layer) and some number of hidden layers (the layers in-between). We will be focusing on a specific version of the MLF called Multi-Layer Perceptron

(MLP), in which the number of hidden layers is always at least one. The neurons in each of these layers feed their result to the neurons in the subsequent layer and in the densely connected version, this output is fed to every neuron in that layer, which is illustrated in Figure 2.2.

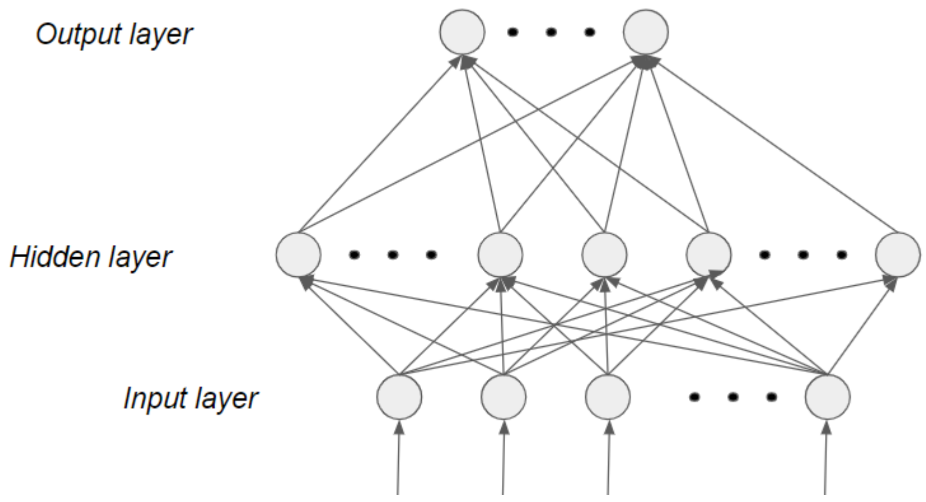


**Figure 2.2:** A typical MLF network, here with multiple output nodes as it works in classifiers

The neurons in the network all represents a function $f : X- > Y$ that takes an input vector X and outputs a single value Y. The internal structure of one such neuron can be seen in figure 2.3.

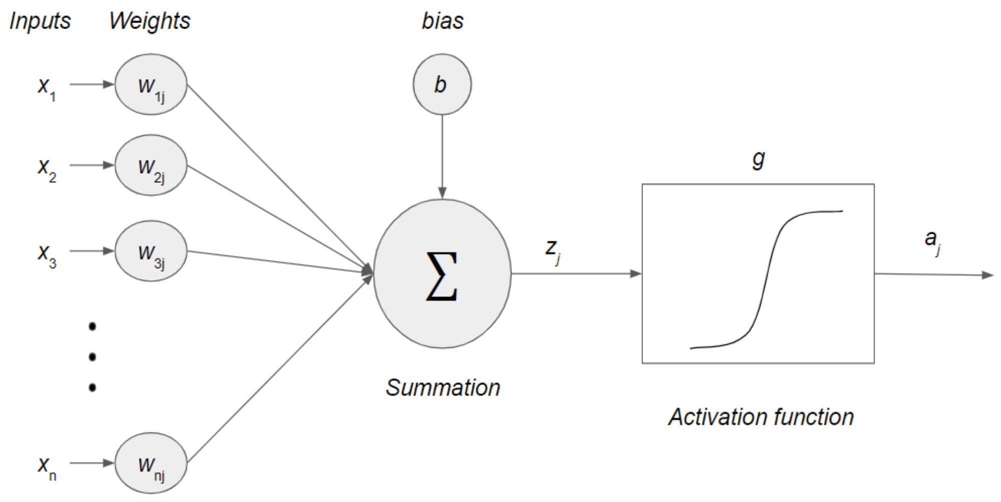


**Figure 2.3:** An inside look of a neuron in which $j$ represents the index of the neuron inside its layer. The output value $aj$ is sometimes also labeled as $y$

As can be seen in the figure, a biased weighted sum is run through some activation function $g$. A single neuron can be expressed mathematically as

$$a_j = g(z_j) = g(b + \sum_{m=1}^{n} x_m w_{mj}) \tag{2.5}$$

Extending this function from a single neuron to the whole layer in order to utilize matrix operations changes the equation to:

$$\mathbf{a}^{[l]} = g(\mathbf{z}^{[\mathbf{l}]}) = g(\mathbf{b}^{[l]} + \mathbf{w}^{[l]T}\mathbf{a}^{[l-1]}) \tag{2.6}$$

**Other Neural Network Types**
There are many other types of neural networks, with 2 specific architectures often mentioned in the contemporary literature. These are the convolutional NNs (CNN), which are usually a form of MLF, and the recurrent NNs (RNN), in which cycles are permitted and which are therefore never feed-forward. Any non-feed forward NN is referred to as an RNN. Both of these are tailored to specific problem spaces, with convolutional NNs being very effective at image recognition and any other problem with large amount of input features in which subpatterns should be recognized. Recurrent neural networks on the other hand also feeds previous answers back into the network, allowing data with temporal connection such as words in a sentence to influence following results. This means that recurrent networks are capable of learning an *order* of results, and not just singular ones.

### 2.2.3 Activation Functions

An activation function is a function that maps a set of inputs to some output. The purpose of activation functions in artificial neural networks is to introduce non-linearity to the model. A non-linear model can be used to solve non-linear problems. If the network model was linear, the model could be collapsed into a single layer and it would not be able to solve the complicated tasks they are used to solve today. Activation functions are present in each neuron of every layer, they compute a weighted sum based on the inputs received from neurons in the previous layer, its bias and the weights of its connections. The output of the activation function of this weighted sum becomes the output of the neuron.
There are many different activation functions, the ones used in the context of this thesis are presented below.

**Sigmoid**

The logistic function, or the sigmoid, is a very common activation function. It is defined as:

$$f(x) = \frac{1}{1 + exp(-x)} \tag{2.7}$$

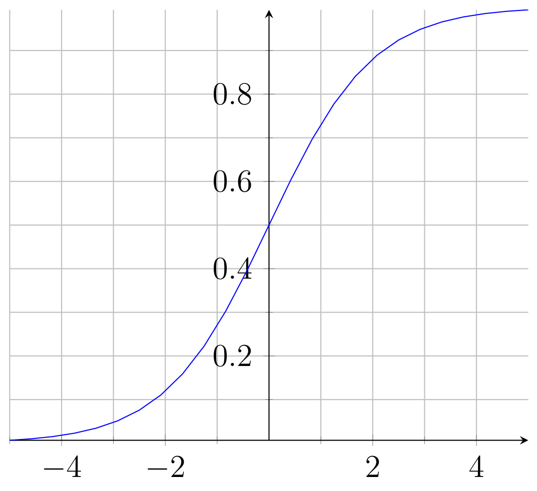


**Figure 2.4:** Logistic Function

It has the characteristics of an S-shaped curve, and that its output is squashed between 0 and 1, which makes it a good choice for binary classification. The input, which is to be classified as either class A or class B, will be squashed a value between 0 and 1 using the sigmoid function. The output spectrum can be interpreted as class A is closer to 0, and class B if it is closed to 1. It has been historically popular since it has a nice interpretation as a saturating firing rate of a neuron, when the output is close to 0 it is interpreted as a neuron which is stale, not firing at all. When the output is close to 1, the neuron is interpreted as firing constantly, at the maximum possible frequency. It has since been surpassed in popularity by other functions, as the sigmoid suffers from what is known as dead gradients, or "the vanishing gradient problem". This means that neurons with very largely negative or positive weights will always produce an output close to the boundaries of the activation function, independent of the input. Additionally it suffers from the fact that it does not have zero-centered output, which will lead to very inefficient gradient updates.

### Tanh

The hyperbolic tangent function, or Tanh, is another sigmoidal activation function which is very similar to the logistic function. What differentiates it from the logistic function is its range. Tanh ranges from -1 to 1, as opposed to the logistic function which ranges from 0 to 1. This gives Tanh an advantage over the logistic function, as negative values will be mapped strongly negative, positive values will be mapped strongly positive, and zero values will be mapped close to zero [10].

Tanh is defined as:

$$f(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.8}$$

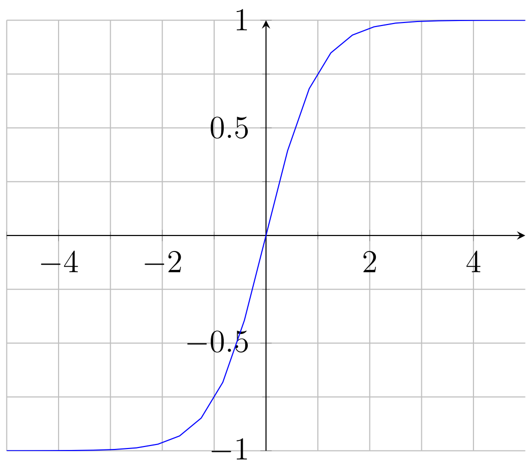


**Figure 2.5:** Hyperbolic Tangent

### ReLU

ReLu (recitifed linear unit) is a popular activation function in modern neural networks [11]. It does not suffer from the problems of zero gradients for positive inputs as is the case with sigmoid. It is also computationally efficient and converges faster than the sigmoid in practice. However ReLU has its drawbacks, it is not zero-centered and it suffers from the issue of zero-gradients in the negative domain [12]. ReLu is defined as:

$$f(x) = max(0, x) \tag{2.9}$$

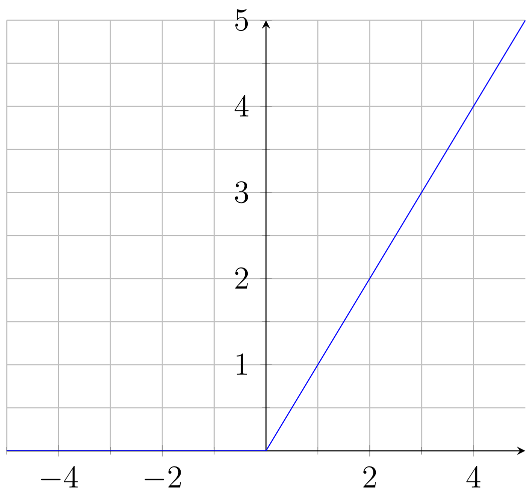


**Figure 2.6:** ReLU function

### Leaky ReLU

Leaky ReLU is a variant of ReLU that has a slight negative slope at values $x < 0$ [13]. This remedies the problem of dead gradients, while reaping the benefits of standard ReLU [12]. Leaky ReLU is defined as:

$$f(x) = max(0.01x, x) \tag{2.10}$$

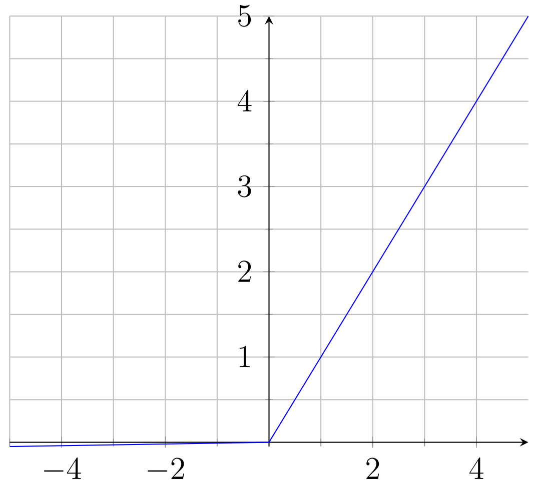


**Figure 2.7:** Leaky ReLU function

### Softmax

The softmax function is generalization of the sigmoid function to several classes. The softmax function is defined as

$$p_c = \frac{e^{y_c}}{\sum_j e^{y_j}} \tag{2.11}$$

It is often used for the output layer of neural networks performing classification tasks, as the output can be seen as a probability that an input to the network belongs to some class $C$.

### 2.2.4 Loss Functions

In order to measure how well a network is performing on a given task, loss functions are used. A loss function takes as input the network's prediction and the actual value, or ground truth, that the network should have given as its prediction. The function produces a score that reflects the difference between the network's output and the correct output. The smaller the value produced by the loss function, the more accurate the prediction was. The goal of an artificial neural network is to minimize its loss function.

**Cross-Entropy**

One loss function that sees common use in classification problems is the cross-entropy function, or logarithmic loss function. It is defined as

$$F(y, \hat{y}) = -\sum_i y_i \log \hat{y} \tag{2.12}$$

where y is the ground truth and $\hat{y}$ is the network's prediction. The output scales in a logarithmic manner with the divergence between y and $\hat{y}$.

### 2.2.5 Backpropagation

Backpropagation(BP) is a technique used in Machine Learning in the process of training artificial neural networks. It is used to calculate the gradients for the weights and biases for each neuron throughout the network, in regard to the loss function. The gradients are the directions in which the weights and biases should be moved for the network to produce results which will decrease the output of the loss function. The gradients of a layer $l$ are calculated using recursive application of the chain rule, with its origin at the loss function. The gradients we are interested in are $\frac{\partial L}{\partial w^{[l]}}$ and $\frac{\partial L}{\partial b^{[l]}}$, for each layer $l$. These gradients can be calculated using the gradients of the following layer $l+1$, which are propagated back up the network.

$$\delta^{[l]} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l+1]}} \frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} = \delta^{[l+1]} \frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} = \sum_j \delta_j^{[l+1]} w_{ij}^{[l]} g'(\mathbf{z}^{[l]}) \tag{2.13}$$

Here $z^l$ denotes the input $w^l \cdot a^{l-1} + b^l$ to the activation function of layer $l$, and a is the activation function of layer $l$. $g$ denotes the activation function used whilst $w_i j$ denotes the weight applied to the output of node i to node j.

The formulas for how the weights and biases affect the cost can now be expressed in terms of the gradient from the previous layer:

First we have the base case, namely the last layer $l$ where no $l+1$ exists. For this layer the the recursive definition of $\delta$ is defined as

$$\delta^{[l]} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} = \frac{\partial L}{\partial \mathbf{a}^{[l]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} = \delta^{[l]} \mathbf{a}^{[l-1]} \tag{2.14}$$

For all other layers, the recursive definition of $\delta$ is defined as:

$$\partial \mathbf{w}^{[l]} = \frac{\partial L}{\partial \mathbf{w}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{w}^{[l]}} = \delta^{[l]} \mathbf{a}^{[l-1]} \tag{2.15}$$

$$\partial \mathbf{b}^{[l]} = \frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{b}^{[l]}} = \delta^{[l]} * 1 \tag{2.16}$$

In 2.14, $\frac{\partial L}{\partial \mathbf{a}^{[l]}}$ is the partial derivative of the loss function, meaning its result will depend on which type of loss function is used. In a similar vein, $\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}}$ is the partial derivative of the activation function and therefore instead depends on which activation function is used in the system.
The result of using BP is a matrix of the gradients of each weight. The next step of the training process then uses these values to alter the weights in order to get closer to a good result.

### 2.2.6 Optimization

The process of training a neural network is in this context known as optimization, and a function known as the optimizer is used to carry this process out. It is during optimization that the weights and biases in the network are updated in order to minimize the loss function and achieve results that are closer to the ground truth which the network is designed to produce.

#### Gradient Descent

Gradient descent is an algorithm used to optimize the parameters of neural networks and it is one of the most popular today. It is an algorithm that minimizes an objective function, by updating the function's parameters to give the minimum result. Good results using a neural network are achieved by minimizing its loss function, this is what gradient descent does. Basic gradient descent is a simple algorithm defined as:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{2.17}$$

Where $\theta$ are the parameters of the loss function $J$, $\nabla_\theta$ are the gradients w.r.t $\theta$ and $\eta$ is the learning rate. The learning rate is used to determine the magnitude of the update done to the parameters. This form of gradient descent is also referred to as Batch Gradient Descent (BGD). It has the drawback of possibly being very slow, because the gradients for the entire dataset have to be calculated in order to perform a single update of the parameters. If the dataset does not fit into memory, the method is not applicable.
Another variant of gradient descent is Stochastic Gradient Descent (SGD). Instead of calculating gradients for the entire dataset, SGD updates parameters for single training examples and labels. It is defined as:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^i; y^i) \tag{2.18}$$

Performing updates on a training example/label basis removes the redundant computations of BGD, as it does not recalculate similar training examples. SGD perform frequent, high variance updates which makes the loss function very noisy. This noise can be good as it allows SGD to escape local minima valleys and find lower minima, but can also cause the opposite. By reducing the learning rate slowly over time this behaviour can be mitigated, causing BGD and SGD to converge in similar ways.

A third variant, the most popular variant, is Mini-batch Gradient Descent, which combines BGD and SGD by performing SGD on small batches of training examples (typically 50-256 samples). Mini-batch Gradient Descent is often referred to as SGD, as it practically is SGD performed in batches.

Optimization is performed in what is known as *epochs*. One epoch corresponds to one complete training session over the entire dataset. A validation/prediction run, during which no learning/optimization takes place, is generally performed when an epoch has terminated, in order to assess the parameter tuning of the epoch.

When solving complex problems using neural networks, gradient descent alone is not enough to converge to global minima or a proper local minima. In modern neural networks it is a building block in more complex algorithms that yield better results. Examples of algorithms that are built upon gradient descent are Adam and AdaMax. In this paper, we use Adam to minimize the loss function. Adam, and its predecessors on which it is based are listed in A. These are Momentum[14], Nesterov Accelerated Gradient[14], AdaGrad[15], RMSprop[16]

### 2.2.7 Over-/Underfitting

Overfitting and underfitting are two concepts that regard poor fitting of a Machine Learning model to data. In the context of a classifier model, overfitting is when the model is able to make accurate predictions on the training data, but not on data that was not part of the training. Underfitting on the other hand, is when the model is unable to make accurate predictions all together. Van der Aalst et. al [17] describes it as allowing behaviour for which there is not enough support. For instance; the classifier is trying to predict whether a data point is of class A or B, but the model is not able to make this distinction.

Overfitting can arise for several reasons, a few examples being; a model is over parameterized for the problem it is trying to solve, or a model has redundant features that are not distinct between classes [18]. There are many techniques in place to combat this problem, ranging from simple techniques such as early stopping, cancelling the training when a drop in validation accuracy is detected, to more complex ones such as dropout [19].

Underfitting can also occur for many reasons. Such as having a too simple model, insufficient data or data of insufficient quality. Techniques for overcoming this problem are for instance; feature engineering, trying to introduce new features that can distinguish data points of different classes, acquiring more data and increasing the complexity of the model [17].

### 2.2.8 Data Pre-processing

Most ML algorithms require not only enough data to train on, but also data that is carefully prepared. It is a requirement for most ML algorithms to normalize any data that spans varied intervals. This is because although methods such as NNs can scale data using smaller or greater weights, these weights will be set after training. If values of a different magnitude appears when the system is applied to new data, this scaling will no longer be enough, and that feature may then disrupt the classification completely. As such it is very common to pre-process(normalize) data using one of these methods:

#### Min-Max Normalization

Normalize data to a 0-1 interval based on following equation

$$x' = \frac{x - min(x)}{max(x) - min(x)} \tag{2.19}$$

This approach is simple, yet works well on well-defined data sets. It is however sensitive to outliers, since a single high or low outlier can greatly affect the resulting interval.

#### Z-score normalization

Normalize the data to have a mean of 0 and a standard deviation of 1 using this

$$x' = \frac{x - \bar{x}}{\sigma} \tag{2.20}$$

Using this approach outliers have little to no effect on other values since they should not affect the mean or standard deviation by any great margin. Because the distribution is scaled down, but not normalized inside a specific interval, outliers also remain as outliers even after standardization.

### 2.2.9 Frameworks

When working with Machine Learning, it is typical to use some kind of framework that provides tools to build and train Machine Learning models. Below we present two frameworks used in this thesis.

**TensorFlow**
*TensorFlow* is an open-source framework for high performance numerical computations. It comes with easy deployment of programs to both CPUs and GPUs, and likely because it was developed by Google Brain, it comes with a strong support for Machine Learning and deep learning. The framework can be used in any python

program and provides many features to quickly get a project going. It is well documented and has an active community, making development using it less likely to stall due to unanswered questions. *TensorFlow* also has other software libraries built on-top of it providing additional functionality and ease of use. One such software library, or API, is Keras. For more information on TensorFlow see their webpage [20].

**Keras**
*Keras* is a high-level API for constructing neural networks. It is written in python and runs on top of TensorFlow. It provides fast experimentation with different Machine Learning models. In Keras, neural layers, loss functions, optimizers, activation functions, initialization schemes and regularization schemes are stand alone modules that can easily be combined and interchanged to test many different models in quick succession and evaluate the results. According to Keras' website, it is highly adopted in the research community, as well as in the industry. It ranks as the second most mentioned framework in scientific papers regarding deep learning, and it is used by tech giants such as Netflix and Uber. The development of Keras is backed by companies like Google, Microsoft and Nvidia, and the Keras API comes packaged with TensorFlow as the tf.keras module, which has been used in this thesis to reap the benefits mentioned. [21]

# 3
# Methods

In this chapter we present our method. We start by providing a more in-depth problem description, followed by descriptions of the methods used. Finally we summarize our contribution.

## 3.1 Problem Description

Stream processing frameworks each have their own method of handling operators of different types, and also of finding this type (e.g. map, filter, join etc.). This poses a problem in a world where for instance cloud providers and frameworks coexist, as efficient resource-allocation in distributed computing depends on knowing the semantics of the application being executed. In the case of stream processing engines, it is the semantics of the operators that define how resources should be allocated, and these semantics are often known only by the framework itself, not by the party deploying applications and allocating resources. There is a demand for a method of analyzing semantics of stream processing queries that is not framework specific, such that for instance cloud providers can make more informed decision in terms of resource allocation and routing of data between allocated machines. As a first step towards this goal, we propose a method of classifying stream processing operators using machine learning, specifically artificial neural networks, that relies solely on information available to the cloud providers.

We achieve this by using only run-time metrics generated by the operators that, while in this case taken from a specific framework, theoretically should be available in any Stream Processing system. Additionally, we refrain from inspecting actual content of the data being processed in the query. Furthermore, in order to create a classifier using ML techniques, we require a dataset to use during training and validation. As no such dataset to our knowledge exists, part of this thesis is also dedicated to creating a data collector, choosing the data to collect, and finally using this collector to create a varied dataset. The method we propose is limited to classifying basic operators present in the public queries used, however a future direction of improvement could be expanding this to include custom or more rare operators as well.

## 3.2 Data Collection

The classifier we propose requires the collection of data that represents operator behaviour without looking into the specifics of the data values being processed. Early investigations revealed that, to our knowledge, no prior data set of this type of data was available. All data that is to be fed into the Machine Learning algorithm must therefore be collected in the context of this project. A simple method to collect this data, which can easily be replicated on multiple different queries is therefore required. Our proposed method is to periodically get data regarding the state of each operator while running a query. This allows for many measurements for each query, and also preserves the potential of running the classifier at runtime, which is a future goal of this project. The exact type of data that was extracted is based on the implementation specifics of the collection method, which will be elaborated on in Chapter 4.

This method was applied to multiple public queries, which utilize different data sets in order to keep the resulting data varied and as representative of the whole problem space as possible. We also complemented this data by crafting simple queries on a public data set ourselves and collecting data from these. We opted to keep data collected from queries we crafted ourselves to a small portion of the total data, in order to keep the variation in the data and the integrity of the results high.

## 3.3 Problem Assessment

In order to get an estimation of the difficulty of the task of classifying stream processing operators operators using this data, a basic clustering algorithm was implemented. The k-means clustering method is utilized to establish a baseline, as it is relatively easy to implement and has previously proven useful for Machine Learning purposes. It shows how distinct the data points are. If the data points are already distinctly separated into clusters, the problem can be solved using a much simpler technique. The results of the algorithm is also used to measure the magnitude of improvement of our contribution.

## 3.4 Classifier

There are many different ML algorithms that can be utilized to create a classifier. The primary method used in this project is the *MLF Neural Network*. There are many other valid options for algorithms, but as we could not find any prior attempts at creating a classifier for this purpose in the literature that could show which types would be suitable, we chose to use *MLF*. The choice was based on the fact that MLF is a general type of NN with a wide range of application which has shown good results in earlier classification attempts within many different fields. It also has the benefits of being easy to implement and being able to quickly create different prototypes. The newer versions of NNs, and many other ML algorithms, have been created to solve more specific issues, as can be seen from the image recognition focus of convolutional NNs [22], and the focus of a time-/sequence-dimensional relationship in recurrent NNs [23]. With the limited information on if these types of special issues will occur in this case, a more general method was prioritized.

In addition to the MLF NN, the classifier construct also contains a pre-processing step, in which input data is prepared before being fed to the NN. This is in order to allow easier classification for the NN.

## 3.5 Classes

The operator classes that the classifier handles are a subset of the basic operators defined in 2.1.1 as these are the ones present in the collected data. The operators that are used in the experiments are listed in Table 3.1.

| | |
|---|---|
| Source | The source of a stream. This can be a file, database, websocket etc. |
| Filter | Evaluates a Boolean function for each element and forwards those for which the function returns true. [24]. |
| Map | Takes one element and produces one element [24]. |
| Reduce | A "rolling" reduce on a keyed data stream. Combines the current element with the last reduced value and emits the new value [24]. |
| Flatmap | Takes one element and produces zero, one, or more elements [24]. |
| Tumbling Join | A join over fixed size, non-overlapping, contiguous time intervals . |
| Aggregation | An aggregation of incoming values. This can be used, for instance, to keep track of a sum. |
| Sliding Aggregate | Aggregation over time windows of a given length. The window moves in time, aggregating over the interval it is currently overlapping. |
| Tumbling Aggregate | Aggregation over fixed size, non-overlapping, contiguous time intervals [25]. |
| Sink | The end/output of a stream. This can be a file, database, console window, websocket etc. |

**Table 3.1:** Basic Operators used in the experiments

While the the implementation specifics can vary between different Stream Processing Engines, the specification of the tasks these operators carry out is more or less standardized. As such, we chose to classify only these operators, which should be present in all Stream Processing Engines and frameworks, making the proposed solution more general.

## 3.6 Features

The features that are fed to the classifier need to be chosen so that they can be used to distinguish classes from each other. Features such as selectivity, the number of operators sending input to, and receiving output from, an operator, and the difference in bytes flowing in versus bytes flowing out are all good candidates as these features characterize certain operators. For instance; a filter will never have a selectivity higher than 1, as filters should only remove elements, never add them. A map on the other hand should always have a selectivity of 1, as it only modifies elements, however the difference in bytes coming in and out can be different. We can see that there are certain aspects of the metrics that can be useful to distinguish operators, and the features used to classify operators are chosen based on these aspects. The theoretical features we considered part of the method can be seen in Table 3.2. These differ slightly from the actual features used during implementation because of implementation specific details that will be further elaborated on in Chapter 4.

| **Feature** | **Idea** |
|---|---|
| # of incoming bytes | This is intended as a supplementary feature to let the NN see patterns we might have missed. |
| # of outgoing bytes | This is intended as a supplementary feature to let the NN see patterns we might have missed. |
| # of incoming records | This is intended as a supplementary feature to let the NN see patterns we might have missed. |
| # of outgoing records | This is intended as a supplementary feature to let the NN see patterns we might have missed. |
| # of input streams | Sources should never have any input operators. Inputs can therefore be used to identify this operator class. |
| # of output operators | Sinks should never have any output operators. Outputs can therefore be used to identify this operator class. |
| Size ratio of records in/out | Shows if the records were altered inside the operators. This never happens for filters, but generally happens in for instance maps. |
| Selectivity | As discussed before, this can separate filters or windowed operators from maps (generally) |
| Memory allocated | The allocation of memory for an operator is a possible indication on whether it is a stateful or stateless operator. |

**Table 3.2:** Features that were identified as useful

## 3.7 Contribution

We have created a classifier for Stream Processing operators using a shallow MLF neural network. The classifier takes metrics generated by operators as input and produces an N-element vector of probabilities, where N is the number of operators which can be classified. The probabilities are corresponding to how certain the classifier is that an operator is of a certain class. The sum of all the elements in the vector is always 1. Hence, the vector [0.1, 0.2, 0.7] corresponds to that the classifier is 70% certain that the operator is of class 3, where 3 corresponds to some operator. The data set used for training and validation is based on the metrics that each operator produces during runtime when executing a Stream Processing query using a SPE. We have also designed a group of custom domain-specific features based on the metrics, that help the classifier differentiate operators. The metrics we use are filtered in order to remove intervals in which some essential feature is 0. This is done as the metrics are collected in intervals over time, and in some cases no data has passed through the operator during a time interval. This corresponds to no action taken by the operator, which provides no distinction as there is no data (e.g. two operators doing nothing are externally inseparable).

We show the effect of: varying certain hyper-parameters, utilizing the domain-specific features, varying the NN neuron layout, and finally, we also show the accuracy of our classifier with the optimal hyper-parameters active and compare it to a basic implementation of the k-means algorithm.

# 4
# Implementation

In this chapter we present our details regarding our implementation of the K-means algorithms, our classifier model and how data used to train the model was collected. We also present the origin of the extracted data, how the data was pre-processed, the features that were used, and finally, how the operators were mapped to our classes.

## 4.1 K-means baseline

As mentioned in 3.3, we use k-means clustering in order to establish a baseline to which our results are compared, and to get an understanding of how clustered the data is in the feature space. Since the data is already labeled, the algorithm is not used in the traditional sense. First, it is run like normal until it converges and there is no movement of the clusters centroids. The clusters are then given a class based on the class of which it has the highest amount of operators. For instance, a cluster with more maps than anything else would be considered a map cluster. All operators that belong to a cluster of its own class are considered correctly classified, while operators that belong to a cluster of a different class from itself are considered incorrectly classified. The number of clusters are chosen according to the elbow method. The algorithm is implemented in python using the scikit learn library.

## 4.2 Classification process

In order to classify operators using only stream processing queries, we go through a series of steps. We will present an overview of these steps in this section, however they will be elaborated on in the coming sections.

The dataset is extracted from a running query. This is then stored and repeated with multiple different queries. The data is then cleaned (e.g. removing empty intervals etc.) and formatted. The formatted data is what is then used to both act as, and also create new, features. The features are then normalized, following which the classes of the operators are remapped.

The end-result of this process is what the ML model utilizes to train and output validation results. These validation results are what will be shown in the majority of the evaluation, however for the final result, we will instead utilize the trained MLF to attempt to classify another set of data known as the test dataset.

An overview of the entire classification process, including data collection and formatting, can be seen in 4.1

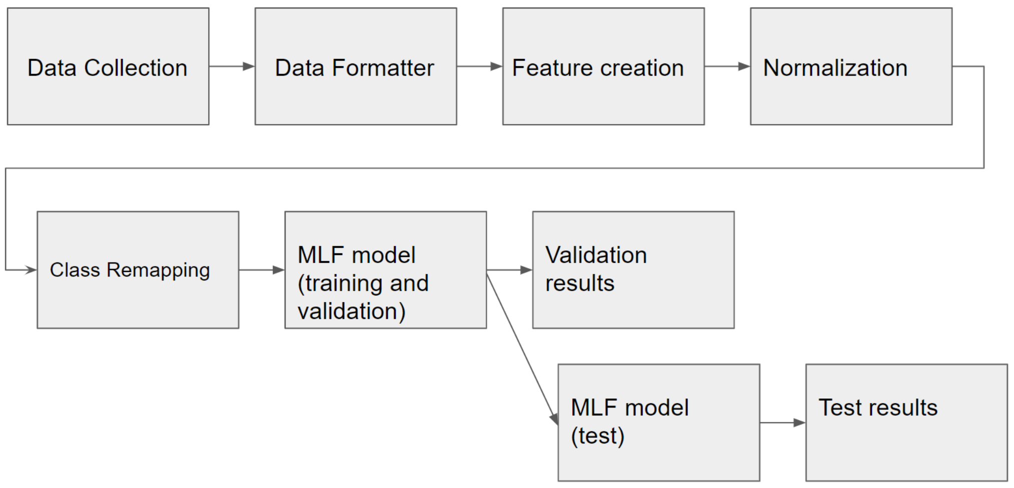


**Figure 4.1:** An overview of the parts that make up the complete classifier

## 4.3 Model

The model consists of an MLF Neural Network. Like any MLF, it has an input layer, a set of hidden layers and an output layer. The input layer has 19 nodes, one for each feature. The three hidden layers have 64, 64 and 512 nodes respectively. All hidden layers use the *Tanh* function as their activation function. The output layer consists of 10 nodes, one for each class. It uses the softmax function to produce a probability distribution, corresponding to how certain the network is that the operator to be classified is of a certain class. The optimizer function used is *Adam* which is presented in Section A.1. The entire model is summarized in table 4.1

The model was fully implemented using *keras* base implementations of layers, loss functions, activation functions and optimizers. Keras' default parameters were used for all of these implementations, these can be seen in table 4.2.

| Layer Index | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Layer Type | Input | Hidden | Hidden | Hidden | Output |
| Amount of Neurons | 20 | 64 | 64 | 512 | 9 |
| Activation Function | - | *Tanh* | *Tanh* | *Tanh* | *Softmax* |

**Table 4.1:** Network Layout

| **Parameter** | **Value** |
|---|---|
| Learning Rate | 0.001 |
| Beta 1 | 0.9 |
| Beta 2 | 0.999 |
| Epsilon | None |
| Decay Rate | 0.0 |

**Table 4.2:** Adam Parameters

## 4.4 Hardware setup

We utilized 3 different computers in this thesis. Machine 3 was only utilized during the data-collection phase, primarily due to its abundance of RAM. Machine 1 and 2 were each used to perform some of the experiments presented in this thesis. Each experiment will denote which machine setup was used to run it.

*Machine 1*:
Processor: Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz (4CPUs)
GPU: Nvidia GeForce GTX 1060 6GB
RAM: 16GB
Python version: 3.6.2
Node version: 8.10.0

*Machine 2*:
Intel(R) Core(TM) i5-4690K CPU @ 4.0GHz (4CPUs)
GPU: Nvidia GeForce GTX 970 3.5GB
RAM: 16GB
Python version: 3.6.2
Node version: 8.9.4

*Machine 3*:
Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz (4CPUs, 8 logical CPUs)
GPU: NVIDIA Quadro M1200
RAM: 32GB
Python version: 3.6.2
Node version: 10.15.3

## 4.5 Data Extraction

Apache Flink is the stream processing engine used as the platform for collecting operator data. The framework is popular and well documented and we also had access to some queries that use this SPE when the work initiated. Flink provides a convenient REST-API for fetching data during runtime of queries, which serves the needs and purposes of this thesis. Additionally, at the moment of writing, there are public queries, along with the corresponding datasets they are intended to run on, available for Flink from which data can be collected.

An application was created that utilizes the aforementioned REST-API to periodically gather metrics from running queries. The interval at which the API is polled is kept as low as possible in order to keep realtime viability and maximize the number of data points. Empirical testing led to an interval of 5 seconds, as shorter intervals often resulted in getting the same values multiple times. For every request, all the metrics generated for the total duration of the query execution is given in response. For instance, the response from the API always contains the total number of bytes sent or received since the query started executing. In order to get interval specific metrics, we calculate the difference for each metric between interval i and interval i-1 and store the result.

In order to utilize the data extraction application on an arbitrary query written in Flink, it can be necessary to perform some minor alterations on it. The main thing to note is that for the metrics to actually be collected on a per-operator basis, operator chaining needs to be disabled, as otherwise metrics will be collected "per-chain" instead. Except for that, no other changes to such a query should be required.

### 4.5.1 Metrics

The metrics that are collected are all the ones available at the task/operator-level scope. This will include both the default metrics and any user-defined metrics. However since the user-defined metrics vary from query to query, they will not be considered, nor listed here. The default metrics that are always present can be seen in Table 4.3.

| **Metric** | **Explanation** |
|---|---|
| numBytesInRemote | Number of bytes that have flowed in via remote channels |
| numBytesInRemotePerSecond | The rate of bytes per second received via remote channels |
| numRecordsOutPerSecond | The rate of records(tuples) output per second. |
| currentInputWatermark | The watermark of the last processed tuple |
| numBytesOut | Number of bytes that have been output |
| numRecordsInPerSecond | The rate of records(tuples) received per second via remote channels |
| numBytesInLocal | Number of bytes that have flowed in through local channels |
| numRecordsIn | Number of records that have been received |
| checkpointAlignmentTime | Time in nanoseconds that the last barrier alignment took to complete. |
| numRecordsOut | Number of records(tuples) that have been output |
| numBytesOutPerSecond | The rate of bytes output per second |
| numBytesInLocalPerSecond | The rate of bytes per second received via local channels |
| nrOfInputs | The number of operators who feed their output to this task |
| parallelism | The current parallelism of the task |

**Table 4.3:** Default metrics present in Flink in the task/operator scope

There is one metric one would expect to be in the list that is missing, namely the number of outputs, since the number of inputs exists. Although this metric is not present natively, it can be derived by looking at which nodes have which inputs. As it seems to be a natural extension of the metrics, and is also something that clearly denotes at least one operator (e.g. Sinks have 0 outputs), we have chosen to create a *nrOfOutputs* feature when extracting the other metrics. Some of these metrics will be considered features for the ML algorithm to consume.

Out of the default metrics, we consider some to be too dependent on external factors to say anything about the operator itself, and as such chose to exclude these. These can be seen in table 4.4 and are not part of any evaluation in Chapter 5.

| **Metric** | **Explanation** |
|---|---|
| currentInputWatermark | Not used in all applications, also only depends on how "far" the query has come along. |
| checkpointAlignmentTime | Says nothing of interest regarding the operator, but is susceptible to the environment which it is run in. |

**Table 4.4:** Removed metrics

### 4.5.2 Data Formatting

The metrics are cleaned up before being fed to the classifier. This is done in a stand-alone data formatter. This formatter removes the metrics shown in Table 4.4. Furthermore it also combines the remote and local input values for bytes and bytes-PerSecond into "bytesIn"(numBytesInLocal+numBytesInRemote) and "bytesInPer-Second" (numBytesInLocalPerSecond+numBytesInRemotePerSecond) respectively. Finally it compounds an arbitrary number of JSON data structures into a single "comma-seperated values" (.csv) file, which is what the classifier consumes. Any other data manipulation is part of the classifier and will be further elaborated on when discussing pre-processing.

## 4.6 Data

The dataset used to train the model was as mentioned prievously extracted from Flink. Queries from different flink applications were executed, and metrics produced by the operators of these queries were co llctedein real-time.
The queries and corresponding data sets used are presented in table 4.5. References to the queries and datasets can be found in Appendix X.

| **Query** | **Dataset** |
|---|---|
| Flink ICU | Generated at run-time |
| QCLCD Flink Experiments[26] | QCLCD Weather Data 200705 -> 201712[27] |
| Genealog [28] | Dataset accompanying paper[28] |
| Twitter Sample Query (Provided by V. Gulisano) | Provided by V. Gulisano |
| Flink Aggregation Sample Query (Provided by V. Gulisano) | Provided by V. Gulisano |
| Taxi Combination Query (Self-made) [29] | Public data-set from the New York City Taxi and Limousine Commission (TLC) [30] [31] |
| TaxiRides (Multi-query project) [29] | Public data-set from the New York City Taxi and Limousine Commission (TLC) [30] [31] |
| Stormbreaker [32] | Data-set from the DEBS 2016 Grand Challenge [33] |
| IOT Traffic Monitor [34] | Generated at run-time |

**Table 4.5:** A table of the queries that metrics were collected from and their respective datasets

The data collected while running the queries on the entire respective dataset will be called the training dataset. In addition to this training dataset, a test dataset was also created. This test dataset was made by collecting data only from the Stormbreaker query, with slightly differing interval and delay.
The operators the training dataset contains, and their distribution, can be seen in Table 4.6. The same information for the test dataset can be seen in Table 4.7.

| **Operator Name** | **# of Entries** | **%** |
|---|---|---|
| Source | 6432 | 14.12% |
| Map | 14067 | 30.9% |
| Aggregate Sliding | 431 | 0.95% |
| Sink | 6960 | 15.29% |
| FlatMap | 2272 | 5% |
| KeyedReduce | 1506 | 3.31% |
| Filter | 3511 | 7.71% |
| AggregateTumbling | 9133 | 20.06% |
| JoinTumbling | 1212 | 2.66% |
| **Total** | 45524 | - |

**Table 4.6:** Operator distribution of the training dataset. This is the final number after removing stale entries.

| **Operator Name** | **# of Entries** | **%** |
|---|---|---|
| Source | 1798 | 8.35% |
| Map | 5795 | 26.92% |
| Aggregate Sliding | 0 | 0% |
| Sink | 4487 | 20.85% |
| FlatMap | 1784 | 8.29% |
| KeyedReduce | 0 | 0% |
| Filter | 104 | 0.48% |
| AggregateTumbling | 6558 | 30.47% |
| JoinTumbling | 998 | 4.64% |
| **Total** | 21524 | - |

**Table 4.7:** Operator distribution of the test dataset. This is the final number after removing stale entries.

## 4.7 Pre-processing

Before the data is fed to the neural network it is processed. As mentioned in 2.2.8 all entries that are fed to the neural network need to be converted to vectors of equal length, containing numerical values. In the case of our implementation, all features (except the labels) are already of numerical nature, and as such only need to be compounded into a vector.

Input vectors are constructed for each interval in the dataset, for every operator present in that interval. Every slot of a vector corresponds to a feature, such as bytes sent and bytes received. In this step, a numerical label corresponding to the operator type is appended to the vector. Additionally, during pre-processing the data is also normalized using min-max normalization. We opted to use this method, as it produced the best results in terms of accuracy. The experiment showing this will be presented in detail in Section 5.3.4.

The pre-processing step is also where the engineered featured are created based on incoming metrics. The metrics are also analyzed, and any row for which the essential bytes/record in/out values are 0, is removed. This applies to both training data and test data.

## 4.8 Feature Engineering

The metrics that were obtained from extraction do not cover the ones listed as potential features in Section 3.6. While humans are able to derive a ratio between bytes using bytes-in and bytes-out, the network was not able to. As such we resorted to employ Feature Engineering.

The features we engineered are derived from the metrics we have available, and aim to cover as much of the desired features that were discussed in the method as possible. Outside of these primary features, we also added others that we hypothesized could improve the accuracy of the network. These were then tested empirically and kept if they improved the results. The specific features that provided improvements, and as such were a part of the final classifier can be seen in table 4.8.

| **Feature** | **Explanation** |
|---|---|
| recordsOutInRatio | Intended to substitute selectivity |
| recordsOutInRatio2 | Calculated using the "perSecond" values instead of absolutes. More stable than its absolute equivalent. |
| bytesOutInRatio | Created on the idea of using something similar to selectivity, but in data size instead of tuple amount |
| bytesOutInRatio2 | Calculated using the "perSecond" values instead of absolutes. More stable than its absolute equivalent. |
| bytesPerRecordIn | Combined with its output counterpart it should help characterize changed to tuples in the operator |
| bytesPerRecordOut | Combined with its input counterpart it should help characterize changed to tuples in the operator |
| bytesPerRecordIn2 | Calculated using "PerSecond" values. More stable than its absolute equivalent. |
| bytesPerRecordOut2 | Calculated using "PerSecond" values. More stable than its absolute equivalent. |
| byteDiff | Differences in bytes in and out, shows if an operator is adding or removing data. Found empirically. |
| recordDiff | Differences in records in and out, shows if an operator is dropping, adding or just forwarding records. Found empirically. |
| bytesPerRecordDiff | Difference in the bytes per record of incoming versus outgoing data. Complements the similar ratio values. |
| bytesPerRecordOutInRatio | The ratio of the bytes per record going out versus coming in. |
| inputsPerOutput | Part of a successful empirical test, very marginally affects accuracy |

**Table 4.8:** Engineered features

Out of the listed features, the ones that created to cover *selectivity* initially only resulted in small increments in accuracy. We noted that the use of selectivity in differentiating between *basic operators* can be seen to largely consists of answering which of these predicates are fulfilled; Is selectivity greater than 1, smaller than 1 or equal to 1? It can be seen in the definition of the *basic operators* in 2.1.1 that these are the 3 points of interest when trying to classify the operator. For instance, Maps have a selectivity of 1, FlatMaps have a selectivity of either 1 or, more often, greater than 1. Filters generally have a selectivity of smaller than 1 (could also be 1 at times where no tuples are discarded). As such, we simplified these features to only denote which one of the 3 points of interest it belonged to, in this case represented by -1 ($S < 1$), 0($S = 0$) or 1($S > 1$). We refer to this as discretization. The effect of this discretization on the results can be seen in Section 5.3.6.

## 4.9 Features

All the features that are used in training and validation are presented in table 4.9. Note that in this table some of the original metrics are absent. These were removed since early experiments showed that they either decreased or had no impact on the performance of the classifier. The decision to remove rather than keep those which had no impact is elaborated on in the feature pruning subsection.

| **Feature** | **Origin** |
|---|---|
| bytesIn | Metric (combined local and remote) |
| recordsIn | Metric (combined local and remote) |
| inputs | Metric |
| outputs | Metric (calculated during data collection) |
| parallelism | Metric |
| bytesInPerSecond | Metric |
| bytesOutPerSecond | Metric |
| inputsPerOutput | Engineered |
| bytesPerRecordIn | Engineered |
| bytesPerRecordOut | Engineered |
| byteDiff | Engineered |
| recordDiff | Engineered |
| bytesPerRecordIn2 | Engineered |
| bytesPerRecordOut2 | Engineered |
| bytesOutInRatio | Engineered |
| bytesOutInRatio2 | Engineered |
| recordsOutInRatio | Engineered |
| recordsOutInRatio2 | Engineered |
| bytesPerRecordDiff | Engineered |
| bytesPerRecordOutInRatio | Engineered |

**Table 4.9:** Features

### 4.9.1 Feature Pruning

As previously mentioned, some metrics were absolved from being considered features. This was mainly because part of the aim of this thesis was to lay a foundation for future work, and as such, we attempted to prune (remove) any feature that did not result in improved performance.

The features that were pruned were chosen based on experiment results. These experiments consisted of removing a feature and running tests to see how the results were affected. If the results were better or unchanged after pruning the feature, it was no longer considered a feature. This type of test was then also run with removing multiple of the "bad" features at once, to make sure that removing them together also had no adverse effect. The features that ended up being pruned can be seen in Table 4.10.

| **Feature** |
|---|
| recordsOut |
| recordsInPerSecond |
| recordsOutPerSecond |
| bytesOut |

**Table 4.10:** Features that were pruned

## 4.10 Mapping Classes

In order to achieve high accuracy with a low rate of false positives and false negatives, accurate labeling is important. Many of the queries from which training data was acquired contain operators that are custom, created by a Flink user, while still performing identical operations to standard Flink operators. They also contain operators that are part of the standard Flink operators, but that has been given more descriptive names, by the user, for the context it runs in.

Using the names of these operators poses a problem for the classification process, as two different operators of type $T$, might have different names, $A$ and $B$. Using the names as labels without any modification will make $A$ and $B$ two different types from the perspective of the neural network, while in reality they are of the same type $T$. Given an operator of type $A$, the network might predict that the operator is of type $B$, which would be considered a faulty prediction, while in reality it is a valid prediction. For this reason, manual work is put into relabeling $A$ and $B$ to $T$. This is sometimes trivial as many operators have names that give away their type. For instance; "Sink-print-to-std-out" and "Sink-write-to-file" are, given their names, obviously of type *Sink*. Other operators; such as "Friendship-Count-24hours" requires more thorough inspection of the source code to determine the correct type. An overview of the special mappings can be seen in table 4.11 with the rest of the mappings being straightforward mapping on partial names (e.g. TumblingWindow(10000)_Aggregate maps to Aggregate_Tumbling and Custom_File_Source maps to Source).

| **Operator name** | **Mapping** |
|---|---|
| FriendshipCount24hours | Aggregate_Tumbling |
| FriendshipCount7Days | Aggregate_Tumbling |
| FriendshipCount60Minutes | Aggregate_Tumbling |
| Window...LinearRoadVehicles | Aggregate_Sliding |
| Window...LinearRoadAccidents | Aggregate_Sliding |

**Table 4.11:** Special operator name mappings

# 5 Evaluation

In this chapter we will present the results of our experiments. First we present the results of the K-means baseline experiment, followed by experiments on how engineered features alter the accuracy. This is then followed by the results of altering values for the hyper-parameters of the model in order to find an optimal configuration. We conclude with the final results, which were acquired using the optimal hyper-parameter values found previously, as well as the optimal features. The results of each experiment will be discussed separately in its own section. In every experiment except for the last test of the final model, section 5.4, the training and validation sets are used exclusively. In the last experiment for the final results the test data set is used as well for evaluation of the final model.

## 5.1 K-means baseline

The k-means clustering experiment to establish the baseline was run 200 times using *Machine 2*, 5 times for each cluster amount and running with a k value of 1-40 clusters, which was enough to find the elbow point. The results are presented in figure 5.1 and table 5.1.

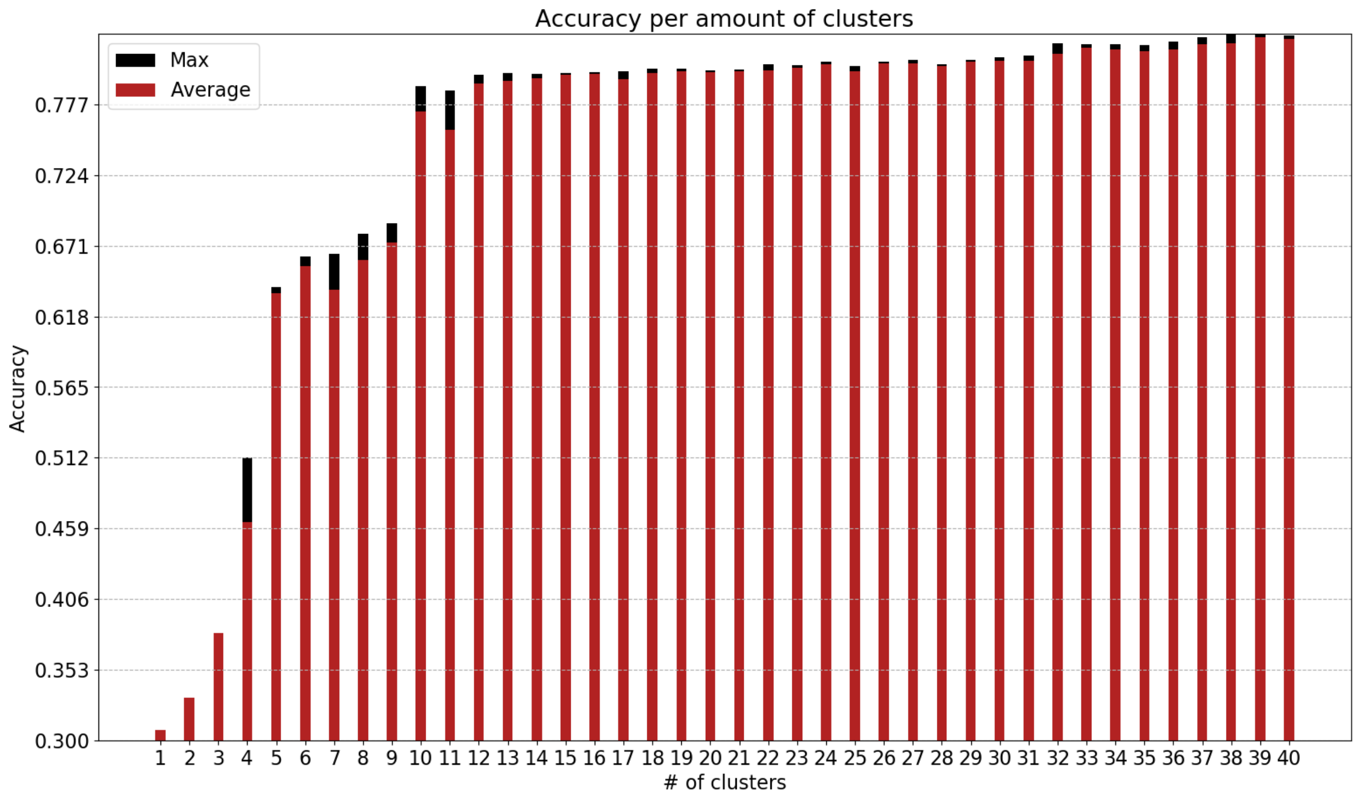


**Figure 5.1:** The results of the k-means clustering experiment on the training data

| Amount of clusters | Average Accuracy | Maximum Accuracy | Comment |
|---|---|---|---|
| 1 | 0.3074 | 0.3074 | Minimum |
| 9 | 0.6732 | 0.6878 | Amount of classes |
| 10 | 0.7719 | 0.7907 | Elbow Point |

**Table 5.1:** The first plateau and elbow point of the k-means clustering experiment

The elbow is found to be at 10 clusters, which is one more than the 9 classes that the classifier has to choose from. The maximum accuracy at this point was 0.79076, and the maximum accuracy at 9 clusters was 0.6878 as can be seen in table 5.1. There is an initially steep climb in accuracy when adding more clusters, this is because for each added cluster, another operator group can be assigned its own cluster which will count them as correctly classified. The steepest climb is at the five first clusters, after this point it slows down rapidly, which to an extent reflects the distribution of the operators within their respective categories, however it is far from a strict correlation. The elbow point is observed at 10 clusters, with an accuracy of 0.7907. This suggests that there is some organic clustering of the data, however there is no distinct separation between every operator class in the feature dimensions that we have been able to identify as meaningful.

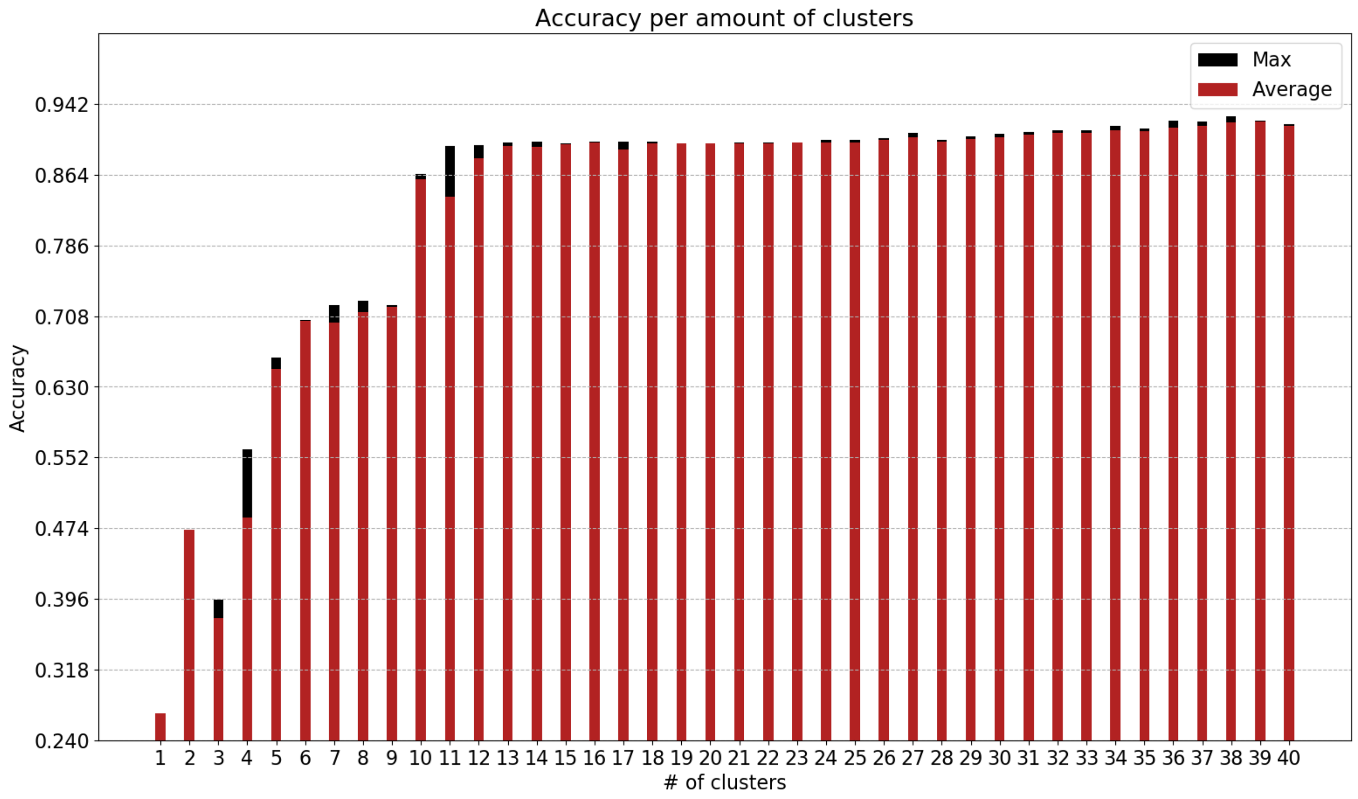


**Figure 5.2:** The results of the k-means clustering experiment on test data

The same algorithm was also run using the test dataset in order to better compare it with our final results. This experiment shows a higher accuracy across the board, indicating that the test dataset has more distinguishable operators. The elbow starts at 10 clusters here as well, however there is still a noticeable increase at 11 clusters. In order to compare this to a classification algorithm, we should however use the amount of clusters that were determined using the training dataset. As such, we consider the k-means algorithm to have achieved an accuracy of 0.89667.

The results presented here indicated that regular k-means clustering performs fairly well, however it is not sufficient to solve the problem of operator classification using this dataset.

## 5.2 Features

This experiment was conducted in order to quantify the effect of the engineered features we proposed in 4.9 versus utilizing only the original metrics described in 4.5.1. The experiment compare three different attempts: *MetricsOnly* which is only utilizing the collected metrics listed in 4.3, *FullFeatures* which has the engineered features in addition to all the metrics, which are listed in table 4.9, and *PrunedFeatures* which has the features in table 4.10 removed. This experiment did 20 tries for each version and was run on *Machine 1*

| | MetricsOnly | Full Features | Pruned Features |
|---|---|---|---|
| Max | 0.8704 | 0.97310 | 0.97340 |
| Average | 0.8700 | 0.97148 | 0.97120 |

**Table 5.2:** Accuracy when using only metrics, all the features or with some features pruned

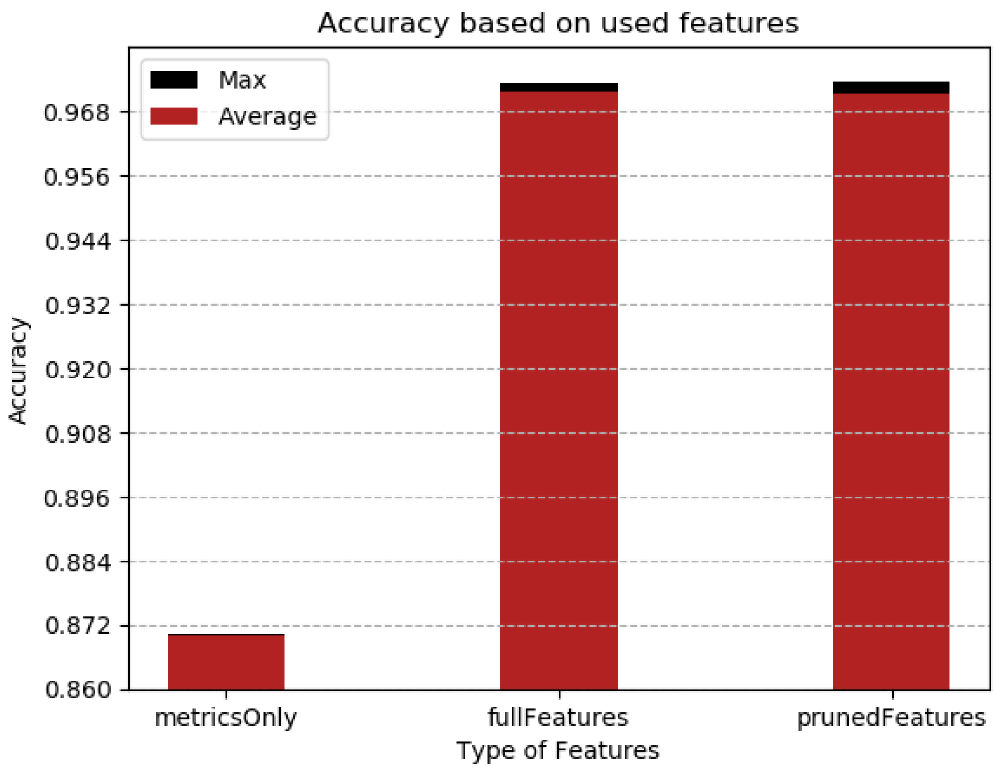


**Figure 5.3:** A graph depicting the accuracies that result from using only metrics, all features and metrics, and a pruned list of features

As can be seen in the graph, utilizing only the original metrics results in significantly worse results than adding our proposed engineered features. Pruning some of the original metrics shows almost no difference in accuracy compared to *FullFeatures*. Originally we had a thought that the information most of the engineered features we propose provide should have been figured out by the model itself when training using the metrics, however it can be clearly seen in the graph that this is not the case. Whether *FullFeatures* or *PrunedFeatures* is the superior way to proceed is hard to definitively say as they both have their potential benefits for future work. *FullFeatures* preserves the most amount of information about operators, meaning it *might* result in reaching higher accuracy if the training data is expanded. *FullFeatures* is the opposite, however this can also be a benefit as it will be more resilient to overfitting and also *might* succeed better in generalization. To know for sure would require more training data and more validation/testing data, so verification will be left as future work. In following experiments we will proceed utilizing *PrunedFeatures*.

## 5.3 Model parameters

This section contains results and descriptions of the different hyper parameter experiments. Here we discuss in detail how the experiments were conducted, what the results were and what hyper-parameters we chose for the final model.

### 5.3.1 Network layout

A good neural network layout was found through testing many different layouts. The experiment consisted of two runs. The first run was an semi-exhaustive search of all combinations between node and layer amounts, with {8,16,32,...,1024} nodes per layer, and 1 to 4 hidden layers. In this run, one test per network layout was performed, and the tests were split between *Machine 1* and *Machine 2*. In the second run, the top 20 models were tested again, with five runs each, in order to determine which network layout had the best accuracy. This run was only run on *Machine 1*. The results of the second run are presented in table 5.3 and figure 5.4. The data of the first run is not presented.

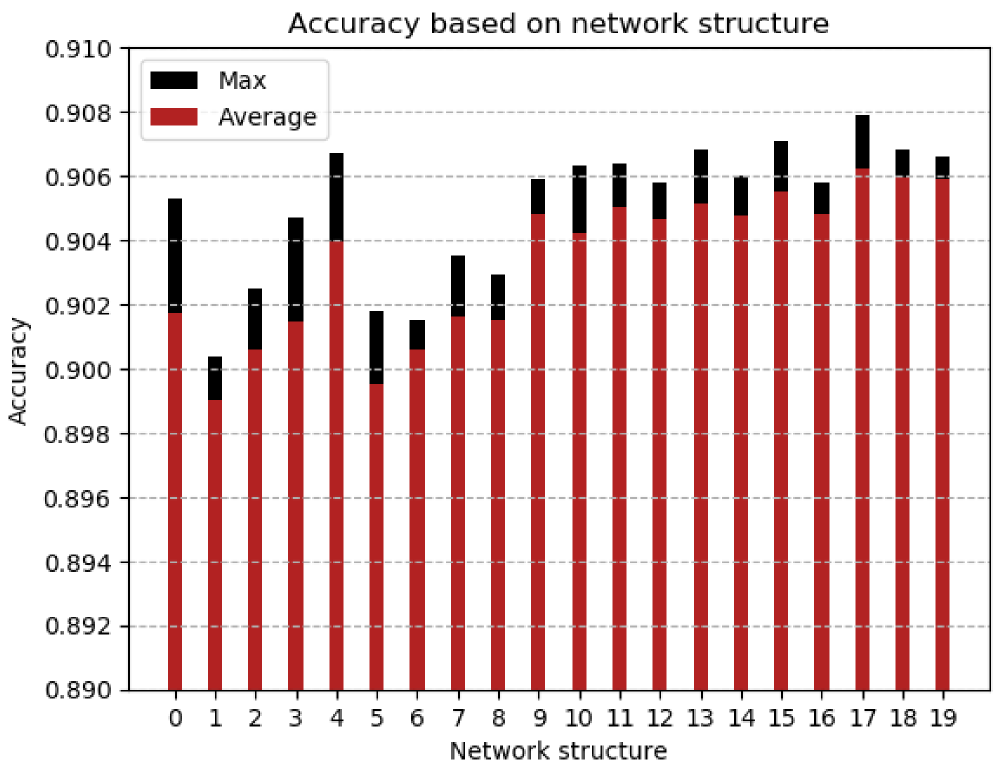


**Figure 5.4:** A graph depicting the maximum and average accuracy over 10 runs for each network layout

| Index | Network layout | Max | Average |
|---|---|---|---|
| 0 | 32 → 32 → 16 | 0.9053 | 0.9017 |
| 1 | 8 → 64 → 256 | 0.9004 | 0.8990 |
| 2 | 64 →32→ 8 | 0.9025 | 0.9006 |
| 3 | 8 → 32 → 64 | 0.9047 | 0.9014 |
| 4 | 16 → 1024 → 32 → 16 | 0.9067 | 0.904 |
| 5 | 64 → 64 | 0.9018 | 0.89951 |
| 6 | 16 → 1024 → 8 → 256 | 0.9015 | 0.9006 |
| 7 | 256 → 512 | 0.9035 | 0.9016 |
| 8 | 64 → 64 → 64 | 0.9029 | 0.9015 |
| 9 | 64 → 16 → 128 | 0.9059 | 0.9048 |
| 10 | 32 → 256 → 16 | 0.9063 | 0.9042 |
| 11 | 16 → 1024 → 1024 | 0.9064 | 0.9050 |
| 12 | 32 → 1024 → 1024 | 0.9058 | 0.9047 |
| 13 | 128 → 128 | 0.9068 | 0.9051 |
| 14 | 128 → 64 | 0.906 | 0.90478 |
| 15 | 64 → 64 → 512 | 0.9071 | 0.9055 |
| 16 | 16 → 1024 → 16 → 256 | 0.9058 | 0.9048 |
| 17 | 128 → 1024 | 0.9079 | 0.9062 |
| 18 | 64 → 64 → 256 | 0.9068 | 0.9059 |
| 19 | 32 → 512 → 32 | 0.9066 | 0.9059 |

**Table 5.3:** Table showing the network layout each index in the graph corresponds to and their respective average and maximum accuracy values

It is apparent from figure 5.4 that there is no clear correlation between the amount of nodes and layers and the accuracy. The difference in maximum accuracy between the worst of these networks and the best is 0.0075 (0.9079 and 0.9004 respectively), which is not much of a difference. The variation of the networks, meaning their maximum value against their average are also stable, and there is no apparent connection between the sizes of the networks and their stability. Because of the very similar results, we chose to move forward using the best layout in each of the 2, 3 and 4 layer categories. These can be seen in table 5.3 and are #17, #15 and #4 respectively. Experiments done using these layouts showed over time that #15 (the 3 layer layout) had the best, and most stable results. Each experiment takes a fair amount of time to run, which is multiplied by how many models they are run for. As such, we chose to only run the coming tests using #15.

It should also be noted that although the results for this second run are extremely similar, there were layouts with up to 10% less accuracy (e.g. sub 80%) in the first run. Most of these were simplistic layouts such as one-layer ones, but some larger networks also showed this trend.

### 5.3.2 Batches

According to Masters et al. a good batch size for accuracy is somewhere between 2-32 [35]. In an earlier paper, Bengio [36] suggest that while 32 is a good default value, batch sizes of up to approximately 200 are typically chosen. We found that having a too small batch size had a large impact on the time each experiment took to complete. Through experiments with different batch sizes, performed using *Machine 2*, a good trade-off between accuracy and run time was found. The results of the batch size experiments are shown in the graph and table below.

| Batch size | 4 | 8 | 16 | 32 | 100 | 200 | 500 | 1000 |
|---|---|---|---|---|---|---|---|---|
| Max | 0.9503 | 0.9505 | 0.9505 | 0.9513 | 0.9511 | 0.9501 | 0.9468 | 0.9424 |

**Table 5.4:** Accuracy of the network based on the batch size of the training steps

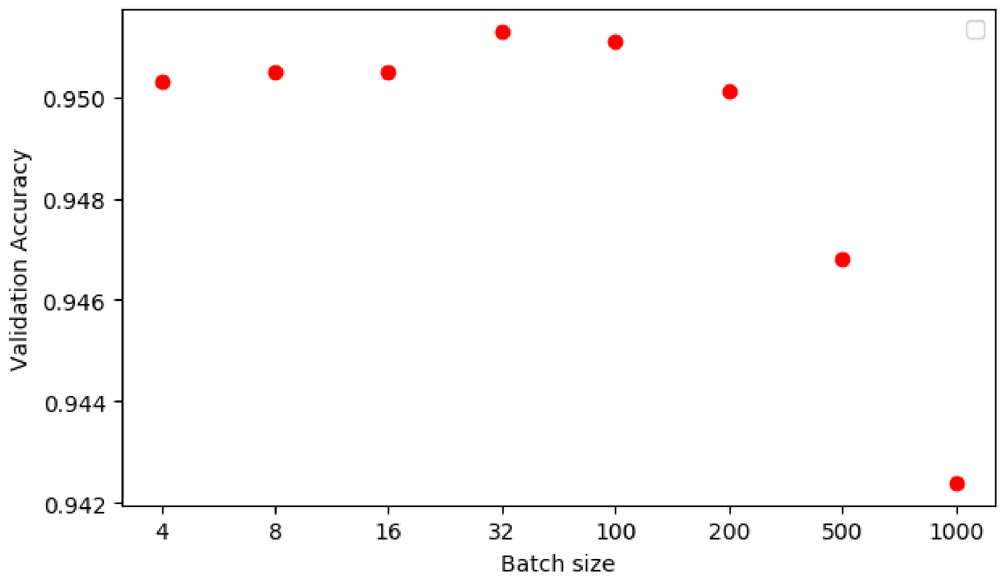


**Figure 5.5:** Accuracy of the network based on the batch size of the training steps

In table 5.4 and figure 5.5, we see that the best accuracy is achieved with a batch size of 32, as suggested above. However, with a batch size of 32, an epoch would take approximately 5 seconds to complete on one of our test rigs. A typical experiment in the context of this thesis runs between 1000-1500 epochs, which would yield a run time of 5000-7500 seconds for a single experiment. We therefore decided to settle for a batch size of 200 when tuning hyper-parameters, as this resulted in a good trade-off between run time and accuracy. For the final experiment, we instead used a batch size of 32 as this was the best result in this experiment, in order to represent the highest possible accuracy we could currently achieve.

### 5.3.3 Activation Function

Determining which activation function to use was done through testing different activation functions that are suggested in literature to be good choices. The traditional Sigmoid and Tanh, which as pointed out in 2.2.3 and 2.2.3 have some flaws, were also tried for good measure. We also tested some less common ones, such as SeLU and ReLU6, which are available in Keras. Each function candidate was tried five times using *Machine 2*. The results from the activation function experiments are presented in table 5.5 and figure 5.6.

| | Sigmoid | tanh | Leaky ReLU | SeLU | PReLU | ReLU | ReLU6 |
|---|---|---|---|---|---|---|---|
| Max | 0.9455 | 0.9543 | 0.9515 | 0.9528 | 0.9534 | 0.9533 | 0.9540 |
| Average | 0.9441 | 0.9516 | 0.9509 | 0.9520 | 0.9522 | 0.9530 | 0.9538 |

**Table 5.5:** Accuracy for different activation functions

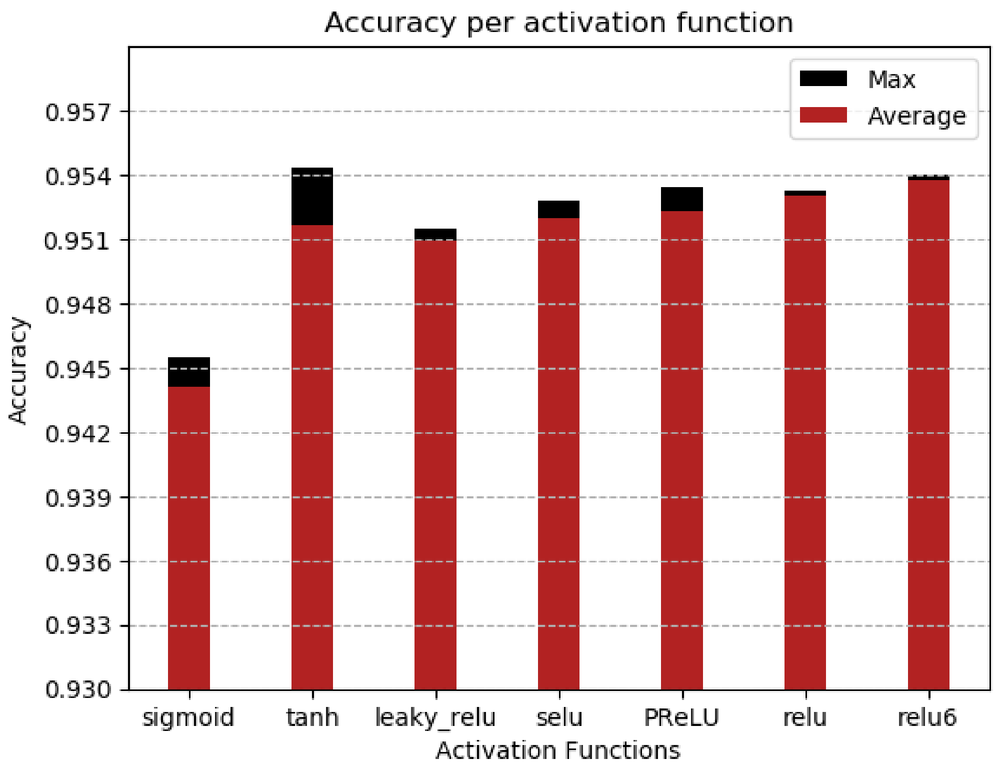


**Figure 5.6:** Accuracy for different activation functions

The function yielding the best result in terms of maximum accuracy was Tanh. As Tanh suffers from the vanishing gradient problem, it was surprising it performed better than leaky ReLU, which is supposed to amend this flaw. However, the results indicate that the characteristics of Tanh fit the properties of the problem better than the other candidates. Another candidate that should be considered is relu6, as although it has a lower maximum, it has a higher average, indicating that it might be a more stable choice. Because tanh and relu6 showed such similar results, an extra round of experiments were performed on just these two to decide which one to proceed with.

| | Tanh | ReLU6 |
|---|---|---|
| Max | 0.9734 | 0.9732 |
| Average | 0.9725 | 0.9711 |

**Table 5.6:** Relu6 vs. Tanh

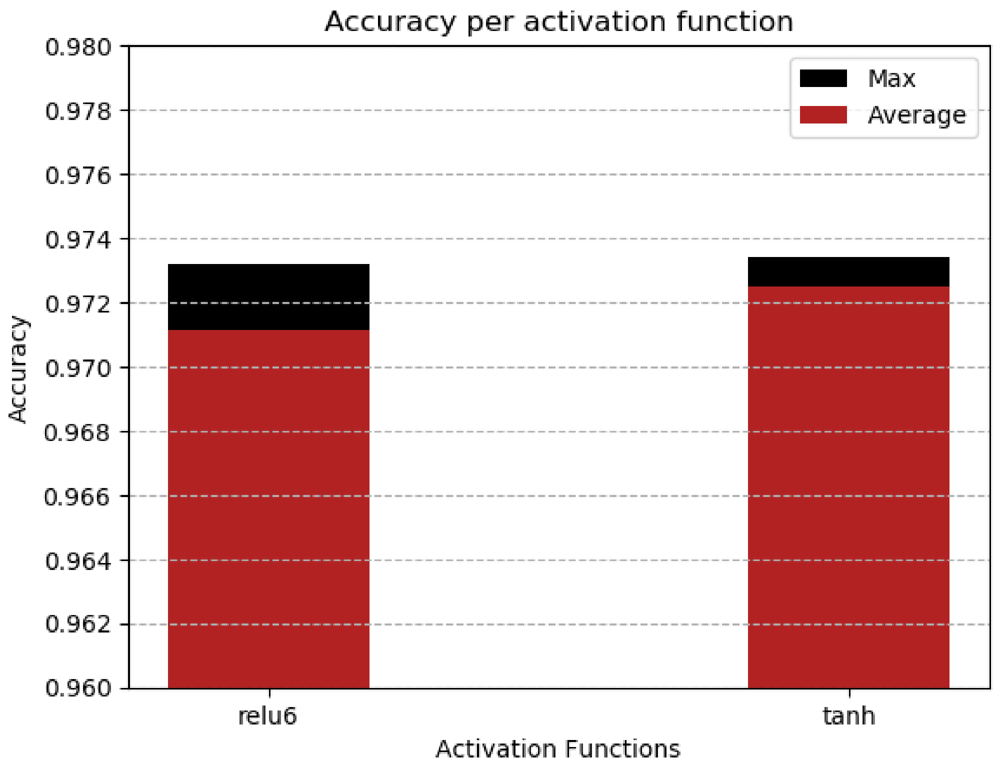


**Figure 5.7:** Relu6 vs. Tanh

This second round displayed in Figure 5.7 shows Tanh with both a higher maximum and average accuracy. While this clearly displays that the accuracy is prone to variation, Tanh still showed the best maximum results in both tests, and the best average in one.As such, we opted to go for Tanh as the activation function of the hidden layers in the final model.

### 5.3.4 Min-max vs Z-score normalization

This experiment was done to see the effect of changing the normalization method on the result. Intuitively this should result in almost no difference when only using training and validation datasets, as the network will learn using the normalized values, and all values are included in the normalization setup. The two methods compared are the *Z-score normalization* and *min-max normalization* described in section 2.2.8. This experiment was run using *Machine 1*

| | Min-Max Normalization | Z-score normalization |
|---|---|---|
| Max | 0.975 | 0.9734 |
| Average | 0.9725 | 0.97115 |

**Table 5.7:** Accuracy for each normalization method

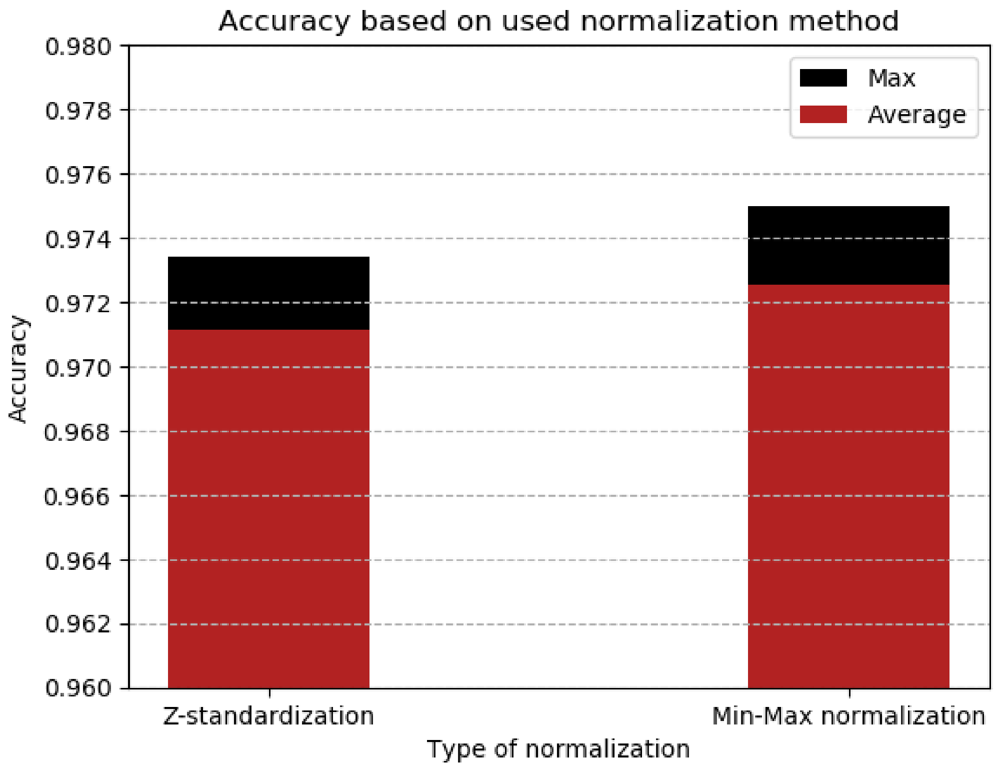


**Figure 5.8:** *A graph depicting the accuracies that result from varying the method of normalization*

It is clear in the graph that *min-max-normalization* performs best for our current setup. Both the average and the maximum accuracy shows slight improvements over the *z-score normalization* method. This is a slightly surprising results, as *z-score normalization* is often referred to as the superior method, because of its better handling of outliers. One reason could be that the data used in this test contain few outliers, however we hypothesize that the main reason that *z-score normalization* performs worse is that this test only uses training and validation datasets, meaning no real "unknown" data can occur. This greatly reduces the risk of outliers, as all the data used is part of the establishing of min-max boundaries (standard deviation and mean for z-score) during normalization, meaning no data ever ends up outside the 0-1 range.

Although we speculate that *Z-standardization* may be the superior method once test data is introduced, we still opted to go for min-max normalization moving forward as it is the clear choice from the results we presented.

### 5.3.5 Optimizer

Which optimizer to use for our model was determined in the same way the activation function was. The accuracy of the model was measured using different optimizers. Each experiment was conducted five times using *Machine 2*, in order to exclude the possibility of a good result just being a lucky run. The results of the optimizer experiments are shown in table 5.8 and figure 5.9 below. All algorithms use the parameters that are default in Keras.

| | Adadelta | SGD | RMSProp | AdaMax | Nadam | Adam | Adagrad |
|---|---|---|---|---|---|---|---|
| Max | 0.8679 | 0.9318 | 0.9726 | 0.9751 | 0.9754 | 0.9747 | 0.8981 |
| Average | 0.8583 | 0.9274 | 0.9700 | 0.9718 | 0.9738 | 0.9730 | 0.8934 |

**Table 5.8:** Maximum and average accuracy for different optimizers

The optimizer that proved to be the best candidates for our model is Adam or Nadam, with RMSProp and AdaMax closely behind. We opted to go for Adam for the final model.

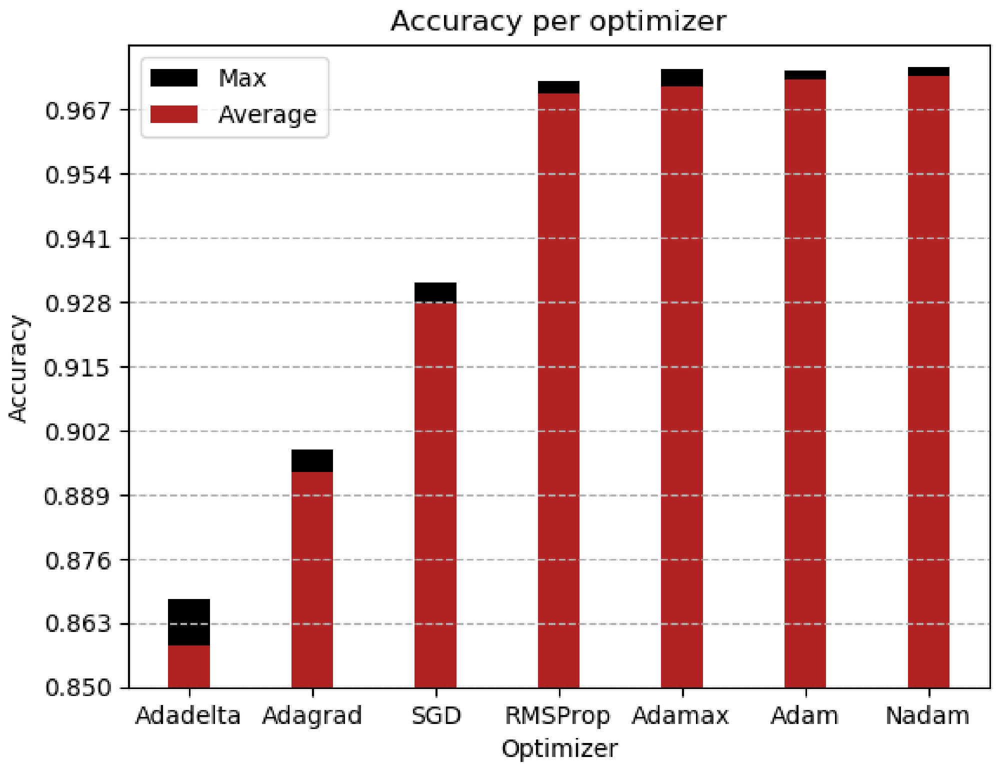


**Figure 5.9:** Maximum and average accuracy for different optimizers

### 5.3.6 Discrete comparisons

This experiment tests the hypothesis presented in Section 4.8. The experiment test each combination of using the real or discrete value for bytes/records. Each combination was run a total of five times using in order to lower the risk of accidental outliers. The results of the runs are presented in Figure 5.10 and a clearer view of the results can be seen in Table 5.9. This experiment was performed using *Machine 1*

| | bDrD | bDrR | bRrD | bRrR |
|---|---|---|---|---|
| Max | 0.9731 | 0.9479 | 0.9743 | 0.9355 |
| Average | 0.97138 | 0.944 | 0.97239 | 0.93358 |

**Table 5.9:** Accuracy for each combination of discretization

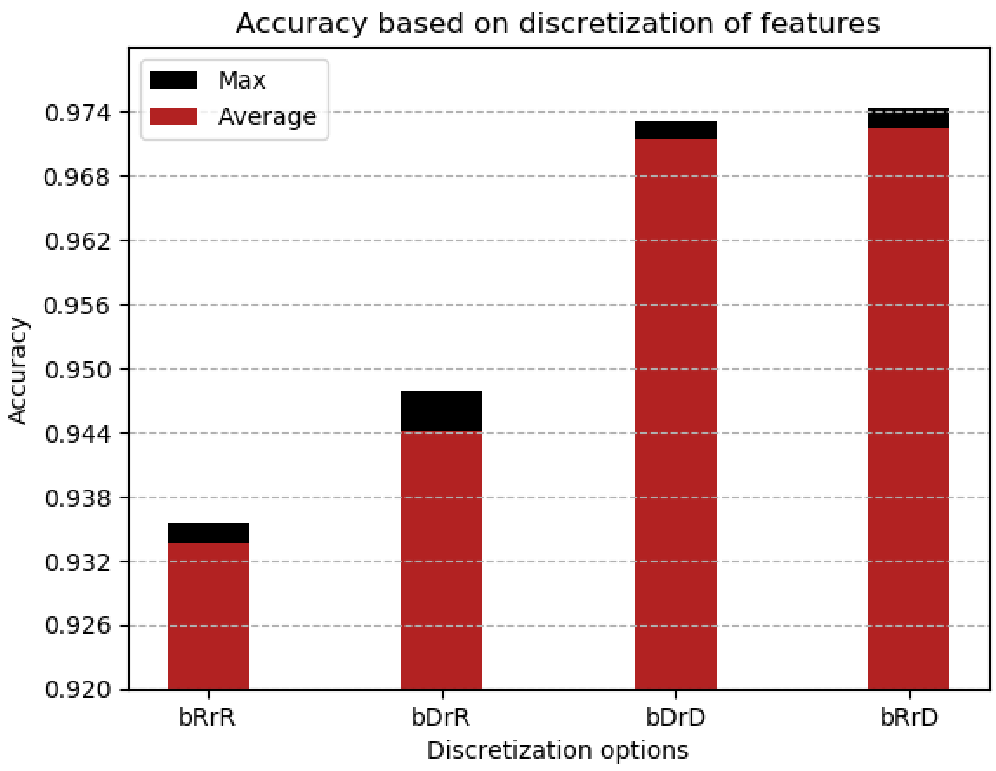


**Figure 5.10:** *A graph depicting the accuracies for different ways of discretizing. The labels on the y-axis shows what values were real respectively discrete in each test. '**bRrD**' means that the byte ratios were real(**b**yte**R**eal) and record ratios were discrete (**r**ecords**D**iscrete).*

It can be clearly seen in Figure 5.3 that using our discretization scheme is superior to utilizing the real values. This is shown as 'bRrR', which is the test where both values are real, lends the worst possible result. The best result is achieved when the ratios for records are discretized and the byte ratios are real ('bRrD'). This can be considered slightly strange when looking at the result for 'bDrR' as this shows a clear improvement when swapping from real byte values used in 'bRrR'. We hypothesize that the reason that discretizing the byte values lends worse results when the record values are already discrete is because the classes the byte values helped identify in the 'bDrR' case are already covered in this instance. Therefore, using real byte values could instead add extra information that potentially helps differentiate additional classes. In the final test the 'bRrD' setup is used.

## 5.4 Final Results

This experiment was performed in order to quantify how well the classifier performs when operating on unknown data. The classifier was setup according to the values yielding the highest accuracies in the previous experiments presented in this thesis. A concrete summary (and reminder) of the network layout and hyper parameters utilized can be seen in Table 4.1 and Table 5.10 respectively. The network was trained using the same validation and training data as previous experiments. This experiment consists of using the trained network to classify the entries in the *Test* dataset presented previously. There are multiple facets to the results that warrant presentation. As such, we present below: A graph showing the learning process(best run), a normalized confusion matrix, a regular confusion matrix and finally an overview of the accuracy of each run in the experiment. The experiment was run 47 times in order to prevent outliers from influencing the overall result. It was run using *Machine 1*.

| **Parameter** | **Value** |
|---|---|
| Optimizer | Adam |
| Error Function | Spare Categorical Cross Entropy |
| Batch Size | 32 |
| Features | 'Pruned' |
| Discretization | *bRrD* |
| Normalization | Min-max normalization |

**Table 5.10:** Model Parameters

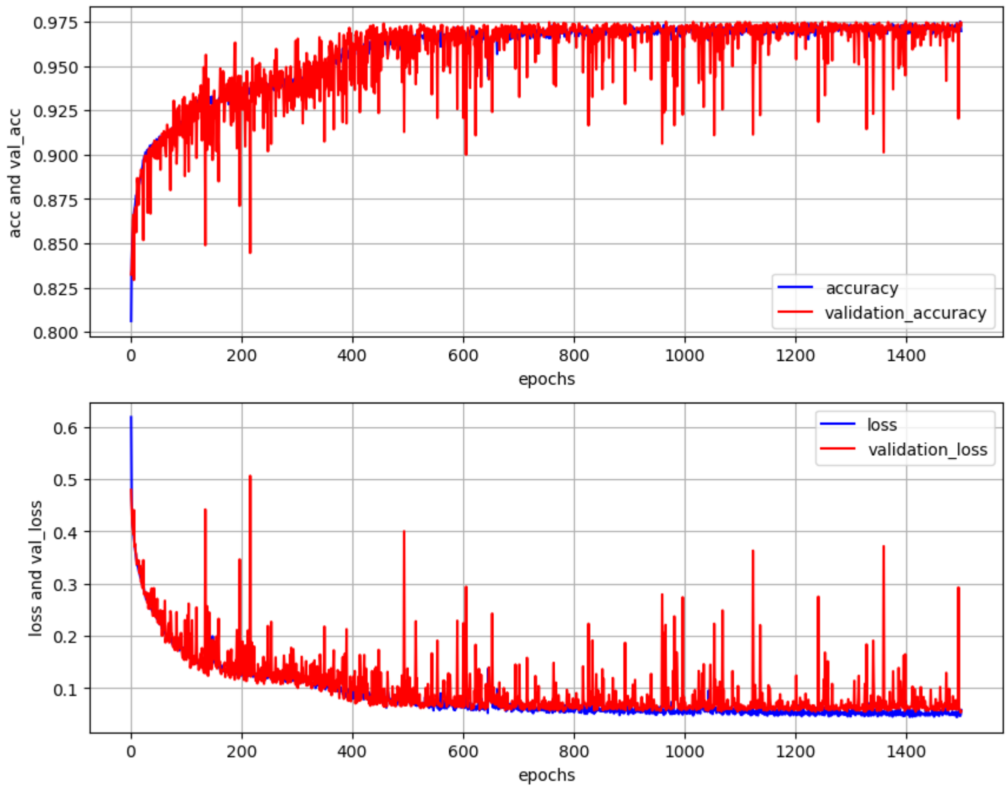
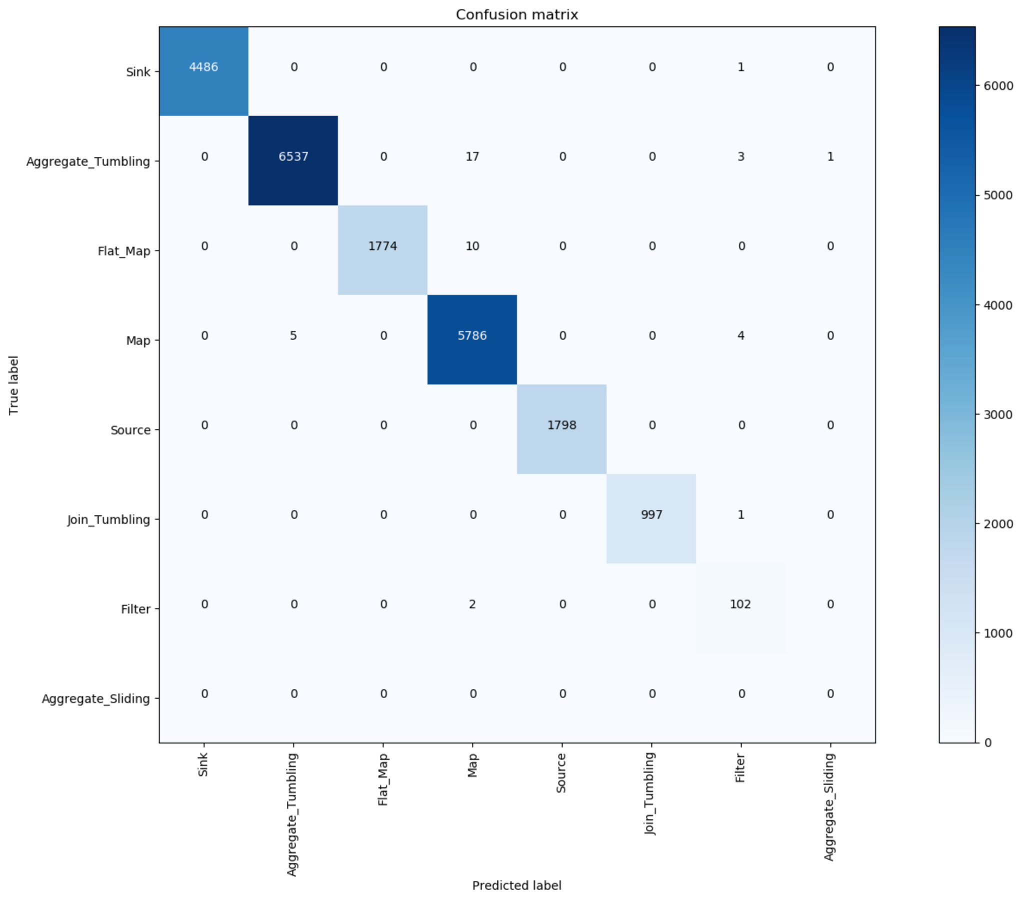
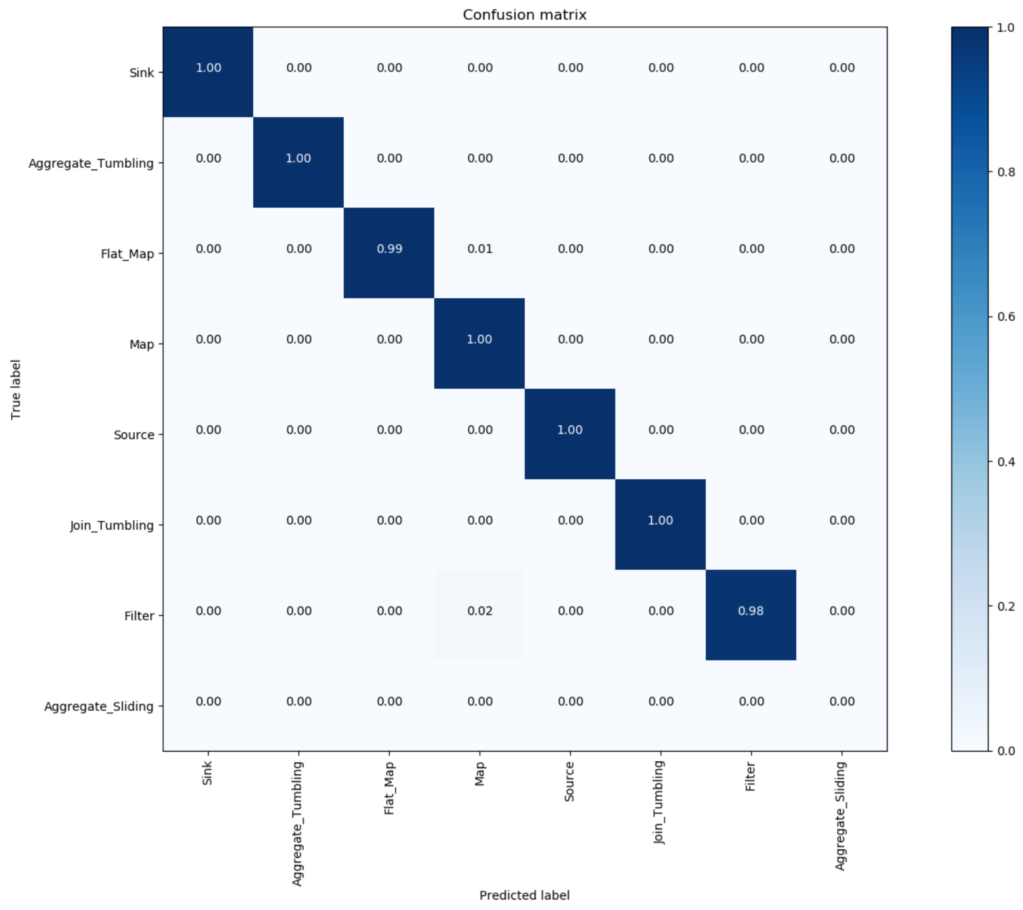


**Figure 5.11:** Accuracy and loss from training and validation.



| | Sink | Aggregate_Tumbling | Flat_Map | Map | Source | Join_Tumbling | Filter | Aggregate_Sliding |
|---|---|---|---|---|---|---|---|---|
| Sink | 4486 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Aggregate_Tumbling | 0 | 6537 | 0 | 17 | 0 | 0 | 3 | 1 |
| Flat_Map | 0 | 0 | 1774 | 10 | 0 | 0 | 0 | 0 |
| Map | 0 | 5 | 0 | 5786 | 0 | 0 | 4 | 0 |
| Source | 0 | 0 | 0 | 0 | 1798 | 0 | 0 | 0 |
| Join_Tumbling | 0 | 0 | 0 | 0 | 0 | 997 | 1 | 0 |
| Filter | 0 | 0 | 0 | 2 | 0 | 0 | 102 | 0 |
| Aggregate_Sliding | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Figure 5.12:** Confusion matrix from the best classification of the test dataset.



| True label \ Predicted label | Sink | Aggregate_Tumbling | Flat_Map | Map | Source | Join_Tumbling | Filter | Aggregate_Sliding |
|---|---|---|---|---|---|---|---|---|
| Sink | 1.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Aggregate_Tumbling | 0.00 | 1.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Flat_Map | 0.00 | 0.00 | 0.99 | 0.01 | 0.00 | 0.00 | 0.00 | 0.00 |
| Map | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Source | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 0.00 | 0.00 |
| Join_Tumbling | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 0.00 |
| Filter | 0.00 | 0.00 | 0.00 | 0.02 | 0.00 | 0.00 | 0.98 | 0.00 |
| Aggregate_Sliding | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

**Figure 5.13:** Normalized confusion matrix of the classification run on the test dataset.

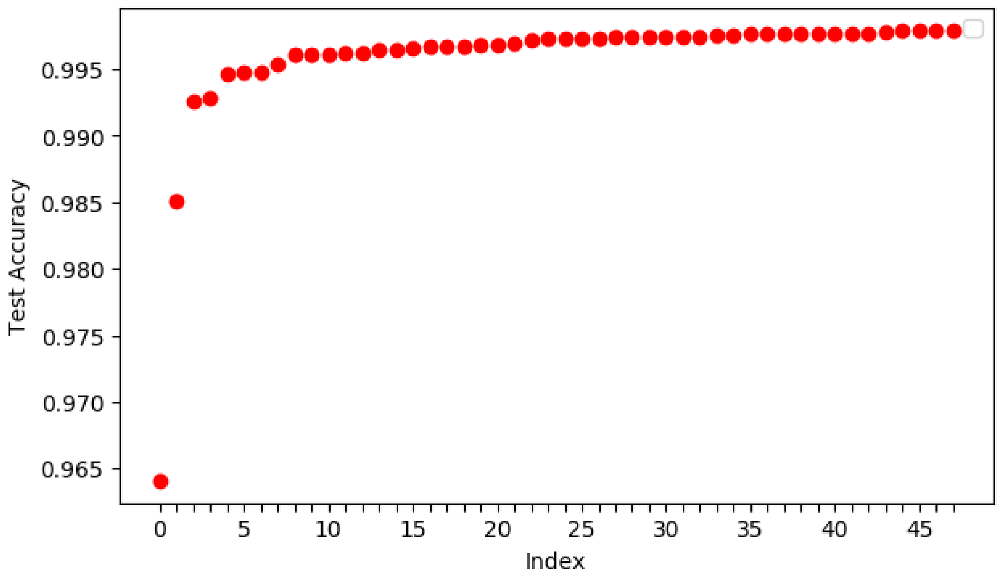


**Figure 5.14:** Accuracy of the classification runs performed on the test dataset.

| | |
|---|---|
| Max | 0.99796 |
| Average | 0.99588 |
| Minimum | 0.96409 |

**Table 5.11:** Minimum, average and maxiumum classification accuracy on the test dataset.

The best run yielded a validation accuracy of 97.51%, and a test accuracy of 99.796%. From the overall results we can see that the classifier performs very well on the test dataset we have used, with accuracy higher than on the validation set. It is not normally the case that the test accuracy outperform the validation, however in this case, we can see some possible reasons for this. First and foremost, we hypothesize that it comes down to the similarity between the test and the training data. The test dataset is as mentioned collected from the same project (Stormbreaker) as part of the training data, which makes the test data to some extent "known" by the classifier. This is further augmented by the fact that the Stormbreaker project is responsible for close to 50% of the training data. With these factors in mind, the about 2.3% increase from validation to test classification no longer seems strange.

The operators that are missing in the test dataset compared to the validation are as shown in Table 4.7, KeyedReduce and AggregateSliding. These are 2 of the 3 least represented classes in the training dataset, which could lead to them also being the classes that the classifier has the hardest time generalizing. This could be an additional reason for the success of the test dataset.

Comparing the results to that of our implementation of the k-means baseline algorithm, it is safe to say that this neural network approach is better in terms of classification accuracy, and that the steps taken in order to improve has proven fruitful.

It should be noted that while the test results may be slightly inflated, the fact that the validation accuracy lies at 97.51%, with test accuracy even higher, means that the method itself shows promise. To confidently state whether out method can work in the general case it needs to be trained and tested on much larger and more varied datasets, however, this is something we leave for future work.

.

# 6
# Related Work

While there are many papers regarding Stream Processing, and also regarding Machine Learning and classification, there are very few combining the few that we could find. Papers we could find were about issues unrelated to this thesis, often related to for instance using a Stream Processing framework to run Machine Learning algorithms. This lack of directly related work can be considered to, to some extent, cement the pioneering status of the work. It also means there is fairly little to discuss in terms of related work, however there are still some aspects for which such a discussion could be had.

What our topic is related to network traffic classification, as we also perform classification based on metrics generated by a network, such as records, byte sizes and neighboring nodes. This field has seen much research, below are summaries of a few representative articles that use Machine Learning in order to classify internet traffWhat our topic is related to is related to network traffic classification using Machine Learning, or other similar techniques.

Bar-Yanai et al. [37] present a method for classifying encrypted traffic using Machine Learning where they reach a classification accuracy of up to 99.9%. They use a Machine Learning model which is a hybrid of the k-means and the k-nearest neighbours algorithms. They also use a normal k-means clustering algorithm as the baseline of their measurements, which is also done in this thesis. The paper is vague in what exact metrics or data it is utilizing, however the notion of looking at general data in classification was inspired by this paper. It should be noted that whilst it was a necessity in their case, with the data being encrypted and as such unreadable, it was for us mostly to future-proof our method, since almost no users wants to use programs that inspect their data content. Their data collection also share a similarity to the way it is handled in this thesis. Bar-Yanai et. al used multiple sources of data, only one of which was a generated database. As this dataset was missing specific types of data, it was then complemented by manually recording data from a controlled environment. We chose to follow a similar route, meaning manually complementing the public data, with our self-created queries, although in our case it was more to complement the data volume rather than some missing types. Overall it can be said that while our paper is very loosely related to that of Bar-Yanai et. al, it has still been a source of ideas and guidance on some specific aspects.

Nguyen and Armitage present a survey of different Machine Learning techniques used to classify network traffic. The approaches that are brought up are clustering algorithms, supervised learning and hybrid solutions. They find that Machine Learning is well suited for the task of network classification, and that techniques such as neural network based approaches and feature space reduction (feature pruning) were able to achieve high accuracy (up to 99%) [38]

# 7
# Discussion & Conclusion

In this chapter we have a small discussion regrading this thesis as a whole. This is then followed by a conclusion regarding the results of the thesis. Finally we attempt to concretize some directions for future work.

## 7.1 Discussion

This section will be an overall discussion where we highlight some points that were made in previous chapter, delve deeper into some, and also discuss new points of interest.

The data collection that was a part of this thesis ultimately came with a few issues. Finding public queries with accompanying datasets (to operate on) presented a larger challenge than initially anticipated. This is the main reason why the test dataset is generated from the same project as part of the training data, which is something that would have been preferable to avoid. There are a few basic operator types that are never classified because they did not appear in any of the queries used to create the datasets. Some of the classes, which we actually classify, are underrepresented in the data, namely AggregateSliding(0.95%), JointTumbling(2.66%) and KeyedReduce(3.31%). The classifier currently handles these very well, however it is hard to say if that just comes down to the locality of the data. Investigating how well the features which we have provided manage to distinguish these with more varied measurements is something we would have liked to include, but will have to be left for future work.

Another thing to note about the collected data is that it was slightly flawed. When we attempted to retrieve data from the Flink API at higher frequency, it was often observed that the metrics of some operators did not update, even though logically a fair amount of data should be flowing through the operator at that time. We also observed some strange measurements where for instance a map had no records in during an interval, but multiple records out. We believe both of these issues are related to how, and mainly when, the metrics are updated by Flink, which sometimes led to what seemed to be half-updated or stale values. This is however just a hypothesis as we never managed to pinpoint the origin of the issue. This is of course something to take note of in regards to our results, however it should be noted that the data generally aligned with expectations.
Certain limitations were established for this thesis. One such limitation was that

we only implemented a neural network as our machine learning algorithm. Whilst we discussed our choice of using MLF NN instead of other NN options such as recurrent or convolutional, it was not weighed against other options. It would be an interesting approach to take the classifier, and compare using different machine learning algorithms commonly used in classification such as for instance the *Support-vector machine*. This is a venue for future work wanting to optimize the solution.

Another limitation was that only basic operators were considered, and any custom operators, unless easily mappable to a basic type, were not considered. While this method is not intended to classify custom operators as "custom operator type 1", or another arbitrary classification, there is a benefit this method could provide in the future once it has reached high enough accuracy. This benefit is that custom operators can be classified as one of the basic operators, and this would (if done successfully) indicate that this custom operator behaves very similarly to that basic operator, and can be handled the same way in terms of efficient handling. This will however only be applicable once the confidence in the method becomes very high.

It has been mentioned a few times throughout this thesis that our solution is intended to be a first step toward a general method of classifying operators. This has characterized some decisions when it comes to features and data-collection. Firstly, it can be seen that we avoid any direct information about the data. The number of records, and bytes is the most information that is collected, and that in itself reveals little about the data. Secondly, as we want future implementations of the classifier to be able to operate on other frameworks than Flink, we attempted to keep the metrics we collected to metrics that are potentially acquirable by analyzing network packets. This assumes of course that each operator is alone on a single node, and all communication takes place over network, which is not generally the case, but workarounds for this is another angle of future work. The final features we use are all based on these metrics, meaning that the potential for more general use, while untested and unfinished, exists.

## 7.2 Conclusion

In this thesis, we set out to investigate the possibility of using machine learning in order to classify stream processing operators. In this context, based on the results of running the model on the test dataset, our conclusion is: yes, it is very much possible. Additionally, our conclusion is that the classifier performs well in the context of this thesis. It achieves a high accuracy, well above the baseline k-means implementation we compare it to, on the data we used. The features we propose, which are used in the classifier, allow the NN to distinguish between all the basic operators we attempt to classify. However, it should be noted that the volume of data, both for training and testing, is not large enough to be able to make a statement regarding how the model would perform outside the context of this thesis. While the classifier is able to distinguish between these basic operators with high accuracy, there is much room for improvement for the future, especially in the realm of data analysis. We believe that by analyzing the data thoroughly, patterns could be detected that one can base new features on that will help the classifier to generalize to more varied data and that could increase the accuracy even more. This is especially the case if more data is available. The decisions and actions that have been taken in order to improve the accuracy in this thesis are mostly based upon hypotheses and trial and error, not so much upon analysis of the dataset.

## 7.3 Future Work

In this section we will present, and summarize previously mentioned, ideas and suggestions for future work.

Part of the purpose of this thesis is to investigate the possibility of more generalized operator classification, and lay the groundwork for such classification. This inherently comes with an idea for future work, which is to build upon either our discoveries or our method toward a generally applicable classifier. We do however not see this as a single endeavour, but rather as a few parts.

One such part is data collection. In order to use a general classifier it is necessary to be able to extract the same information from a query, regardless of framework. Possible directions for this is firstly to get the metrics via network packet analysis or statistics, as operators need to communicate regardless of framework. This poses many challenges in how to properly interpret information, but also on how this could be extended to work when multiple operators possibly could be present on the same node. Another way would be for the frameworks themselves to make sure they provide a way to extract these metrics from a running query, similar to the way Flink exposes the metrics this thesis collects. Other methods could also be possible, but we leave that up to future papers to explore.

Another part is extending and optimizing the classifier itself. This is also a problem with multiple avenues of inquiry available. A first step would be further verifying and possibly extending the set of proposed features used in this thesis. It would also be interesting to compare utilizing different ML algorithms in the classifier in order to find if another type or implementation would work better.

The results of the model proposed in this thesis operating on test data shows promise, however in order to verify the model's viability as a general purpose operator classifier, the model needs to undergo heavier scrutiny using a larger dataset. This data has to be more varied, including more operator classes, and should be collected from a much wider range of queries, such that the full scope of stream processing use cases is included. As part of this we would also suggest developing a more accurate data collector, that can give more exact metrics than what could be done in this thesis. Having data which is accurate would be of great benefit to the classification process, as some operators produce metrics which are very similar, it is sometimes important to have data which is exact down to the byte. As mentioned in the conclusion 7.2, we would also recommend thorough statistical analysis of the dataset, to find characteristics in the data on which new features can be based. An example would be to look at the distribution of the operators in each feature dimension, which could help with pin-pointing which features heavily characterize certain operators and derive new better features from these.

In the discussion we mention the classification of custom operators. More specifically, to classify a custom operator as the basic operator it is most similar to. This could bring a twofold benefit. First, it could allow frameworks to handle these custom operators the same way that the corresponding basic operator would be, allowing potential performance improvements. This is especially interesting in the case of custom operators as, while frameworks differentiates between a "map" and "filter" already, they can't accurately cater to the need of custom ones. Secondly, users that create a custom operator may realize that their custom operator is actually just for instance an aggregate. This could allow users, especially inexperienced, to take a step back and consider if the custom operator could be written as the corresponding basic operator instead. This would be the best outcome as frameworks know exactly how to handle these basic operators as efficiently as possible.

Another task for left for future work is to make the classifier operate in real-time as part of a stream processing engine or other stream processing pipeline. This is arguably essential in order to get the full potential out of a stream processing operator classifier, as a lot of the value lies in being able to perform fine-grained operator placement, to boost the performance of stream processing. This applies both to the learning process, having the classifier constantly learning on new incoming data, as well as the actual classification and making decisions based on the information that is gained regarding what operators a query consists of.

# Bibliography

[1] Apache, *Apache Storm*, 2016. [Online]. Available: `http://storm.apache.org/%20http://storm.apache.org/index.html`.

[2] ——, *Apache Flink: What is Apache Flink?* [Online]. Available: `https://flink.apache.org/flink-architecture.html`.

[3] S. Perera, *What is Stream Processing?*, 2018. [Online]. Available: `https://wso2.com/library/articles/2018/05/what-is-stream-processing/`.

[4] IBM, *IBM Knowledge Center - Streams processing applications.* [Online]. Available: `https://www.ibm.com/support/knowledgecenter/SSCRJU_4.0.0/com.ibm.streams.dev.doc/doc/streaming_applications.html`.

[5] Scala, *The Scala Programming Language.* [Online]. Available: `https://www.scala-lang.org/`.

[6] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm", *Applied Statistics*, vol. 28, no. 1, p. 100, 1979, ISSN: 00359254. DOI: `10.2307/2346830`. [Online]. Available: `https://www.jstor.org/stable/10.2307/2346830?origin=crossref`.

[7] Andrea Trevino, *Introduction to K-means Clustering*, 2016. [Online]. Available: `https://www.datascience.com/blog/k-means-clustering`.

[8] P. Jeffcock, *K-Means Clustering in Machine Learning, Simplified | Oracle Big Data Blog*, 2018. [Online]. Available: `https://blogs.oracle.com/bigdata/k-means-clustering-machine-learning`.

[9] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview", *Neural Networks*, vol. 61, pp. 85–117, Apr. 2015. DOI: `10.1016/j.neunet.2014.09.003`. [Online]. Available: `http://arxiv.org/abs/1404.7828%20http://dx.doi.org/10.1016/j.neunet.2014.09.003`.

[10] S. Sharma, *Activation Functions: Neural Networks – Towards Data Science*, Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6, 2017. [Online]. Available: `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`.

[11] Yann Lecun, Y. Bengio, and G. Hinton, "Deep learning", *Nature, International Journal of Science*, vol. 521, no. 1, pp. 436–444, 2015. DOI: `doi:10.1038/nature14539`. [Online]. Available: `https://www.nature.com/articles/nature14539.pdf`.

[12] S. Y. Fei-Fei Li Justin Jognson, *(44) Lecture 6 | Training Neural Networks I - YouTube*. [Online]. Available: https://www.youtube.com/watch?v=wEoyxE0GP2M&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=6.

[13] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier Nonlinearities Improve Neural Network Acoustic Models”, in *Proc. icml*, vol. 30, 2013, p. 3. [Online]. Available: https://pdfs.semanticscholar.org/367f/2c63a6f6a10b3b64b8729d601e69337ee3cc.pdf.

[14] S. Ruder, “An overview of gradient descent optimization algorithms”, *CoRR*, vol. abs/1609.0, pp. 1–14, 2016, ISSN: 0006341X. DOI: 10.1111/j.0006-341X.1999.00591.x. [Online]. Available: http://arxiv.org/abs/1609.04747.

[15] J. Duchi JDUCHI and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization * Elad Hazan”, Tech. Rep., 2011, pp. 2121–2159. [Online]. Available: http://delivery.acm.org/10.1145/2030000/2021068/p2121-duchi.pdf?ip=129.16.140.73&id=2021068&acc=OPEN&key=74F7687761D7AE37.3C5D6C4574200C81.4D4702B0C3E38B35.6D218144511F3437&__acm__=1541446280_6ea446be214762a4593164f933be45f5.

[16] G. E. Hinton, “06 Optimization: How to make the learning go faster”, *Coursera*, vol. 4, pp. 26–31, 2012. DOI: https://www.coursera.org/learn/neural-networks/lecture/YQHki/rmsprop-divide-the-gradient-by-a-running-average-of-its-recent-magnitude. [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[17] W. M. P. Van Der Aalst, · . V. Rubin, · . H. M. W. Verbeek, · . B. F. Van Dongen, · . E. Kindler, and · . C. W. Günther, “Process mining: a two-step approach to balance between underfitting and overfitting”, *Softw Syst Model*, vol. 9, pp. 87–111, 2010. DOI: 10.1007/s10270-008-0106-z. [Online]. Available: http://www.processmining.org.

[18] D. M. Hawkins, “The Problem of Overfitting”, *Journal of chemical information and computer sciences*, vol. 44, pp. 1–12, 2004. DOI: 10.1021/ci0342472. [Online]. Available: https://pubs.acs.org/doi/full/10.1021/ci0342472.

[19] N. Srivastava, G. Hinton, A. Krizhevsky, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, Tech. Rep., 2014, pp. 1929–1958. [Online]. Available: http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf?utm_content=buffer79b43&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer.

[20] TensorFlow, *TensorFlow, https://www.tensorflow.org/*. [Online]. Available: https://www.tensorflow.org/.

[21] Keras, *Keras*. [Online]. Available: https://keras.io/.

[22] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going Deeper With Convolutions*, 2015. DOI: 10.1109/CVPR.2015.7298594. [Online]. Available: https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Szegedy_Going_Deeper_With_2015_CVPR_paper.html.

[23] B. Pang, L. Lee, and S. Vaithyanathan, "Thumbs up?", in *Proceedings of the ACL-02 conference on Empirical methods in natural language processing - EMNLP '02*, vol. 10, Morristown, NJ, USA: Association for Computational Linguistics, 2002, pp. 79–86. DOI: 10.3115/1118693.1118704. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1118693.1118704.

[24] Apache, *Apache Flink 1.7 Documentation: Operators*. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/.

[25] M. Kinsman, M. MacReady, and A. Raizman, *Tumbling Window (Azure Stream Analytics) - Stream Analytics Query | Microsoft Docs*, 2016. [Online]. Available: https://docs.microsoft.com/en-us/stream-analytics-query/tumbling-window-azure-stream-analytics.

[26] P. Wagner, *FlinkExperiments*. [Online]. Available: https://github.com/bytefish/FlinkExperiments.

[27] QCLCD, *QCLCD Weather Data*. [Online]. Available: https://www.ncdc.noaa.gov/orders/qclcd/.

[28] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou, *GeneaLog: Fine-Grained Data Streaming Provenance at the Edge*. New York, USA: ACM, 2018, pp. 227–238.

[29] DataArtisans, *TaxiRides*. [Online]. Available: https://github.com/dataArtisans/flink-streaming-demo.

[30] New York City Taxi and Limousine Commission (TLC), *TLC Yellow Rides Janaury 2010*. [Online]. Available: https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2010-01.csv.

[31] ——, *TLC Yellow Rides January 2017*. [Online]. Available: https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2017-01.csv.

[32] BigDataPlayground, *Stormbreaker*. [Online]. Available: https://github.com/BigDataPlayground178/Stormbreaker.

[33] V. Gulisano, Z. Jerzak, S. Voulgaris, and H. Ziekow, *The DEBS 2016 Grand Challenge*, DEBS '16. Irvine, California: ACM, 2016, pp. 289–292, ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933519. [Online]. Available: http://doi.acm.org/10.1145/2933267.2933519.

[34] S. B. Gdaim, *IoT Traffic Monitor*, 2016. [Online]. Available: https://github.com/harsh86/iot-traffic-monitor-flink.

[35] D. Masters and C. Luschi, "Revisiting Small Batch Training for Deep Neural Networks", *arXiv preprint arXiv:1804.07612*, vol. abs/1804.0, Apr. 2018. [Online]. Available: http://arxiv.org/abs/1804.07612.

[36] Y. Bengio, *Practical recommendations for gradient-based training of deep architectures*. Jun. 2012, pp. 437–478. [Online]. Available: `http://arxiv.org/abs/1206.5533`.

[37] R. Bar - Yanai, M. Langberg, D. Peleg, and L. Roditty, "Realtime Classification for Encrypted Traffic", in *International Symposium on Experimental Algorithms*, Springer, Berlin, Heidelberg, 2010, pp. 373–385. DOI: `10.1007/978-3-642-13193-6{\_}32`. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-13193-6_32`.

[38] T. T. T. Nguyen and G. Armitage, "A Survey of Techniques for Internet Traffic Classification using Machine Learning", *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, 2008. DOI: `10.1109/SURV.2008.080406`. [Online]. Available: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.474.858&rep=rep1&type=pdf`.

[39] D. P. Kingma and J. Lei Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014. [Online]. Available: `https://arxiv.org/pdf/1412.6980.pdf`.

[40] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An Elastic and Scalable Data Streaming System", *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012, ISSN: 1045-9219. DOI: `10.1109/TPDS.2012.24`. [Online]. Available: `http://ieeexplore.ieee.org/document/6127868/`.

[41] Arvind Rai, *Java 8 Stream reduce() Example*, 2018. [Online]. Available: `https://www.concretepage.com/java/jdk-8/java-8-stream-reduce-example`.

# A
# Appendix 1

## A.1 Optimizers

This section contains more detailed explanations of the different optimizer algorithms that are either used in this thesis, or that algorithms used in the thesis are built upon.

### Momentum

Momentum improves SGD's ability to navigate in areas which contain surfaces that are steeper in one direction than in others. By adding a momentum term $\gamma$, it makes SGD move in the right direction and prevents it from going back and forth between the steeper parts of the area. $\gamma$ is a fraction of the update vector of the past time step, which is added to the current update vector [14].
The momentum term can intuitively be interpreted as the physical momentum of a ball rolling down a hill and into a small valley. With enough momentum, the ball will escape the valley. Without enough momentum, the ball can stop and remain trapped.

### Nesterov Accelerated Gradient (NAG)

A drawback of Momentum is that it might overshoot, due to too much momentum carrying over from the previous step. NAG mitigates this problem by approximating where the parameters are going to be after the next update, and calculating the gradients w.r.t to these approximated future parameters [14]. Returning the the example with the ball in the previous section, NAG approximates where the ball will be some small amount of time in the future, and takes this into consideration when calculating the gradients. This prevents the ball from escaping a global minima, due to too much momentum.

### Adagrad

Adagrad [15] improves NAG by adapting the learning rate $\eta$ w.r.t to the significance of the parameters, for each parameter indivudially. Frequent parameters require smaller updates, while infrequent parameters require greater updates. This makes Adagrad well suited for working with sparse datasets, or datasets with features that have low occurrence. Adagrad's update rule can be defined as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{A.1}$$

Here, $g_t$ are the gradients of the loss function at time step t, $G_t$ is a diagonal matrix in which the elements are the sum of the squares of the gradients w.r.t $\theta_i$ up untill time step t. $\epsilon$ is a term added to avoid division by zero. $\odot$ denotes element wise matrix-vector multiplication. Because $G_t$ contains the sum of the squares of past gradients, during training the sum will keep growing and in turn will cause the learning rate to keep decreasing. Eventually it becomes so small that learning is effectively stopped.

### RMSprop

RMSprop [16] tries to remedy the problem of accumulating squared gradients which cause learning to eventually halt. It is accomplished by dividing the sums by a exponentially decaying running average of the squared gradients [14].

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2 \tag{A.2}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{A.3}$$

Here, $\gamma$ is the momentum term, $E[g^2]_t$ is the running average of the squared gradients at time step t.

### Adam Optimization

Adam [39] (Adaptive Moment Estimation) is an optimization function used in neural networks, which builds on gradient descent. Like RMSprop, Adam utilizes an exponentially decaying average of the past squared gradients, $v_t$ to combat the decay of the learning rate. The difference from RMSprop is that it also uses an exponentially decaying average of past *gradients*, $m_t$, which works similar to momentum [14].

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t \tag{A.4}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \tag{A.5}$$

$\beta_1$ and $\beta_2 \in [0, 1)$ are hyper-parameters that control the rates of the exponential decay of the moving averages. $m_t$ and $v_t$ are estimates of the 1st moment (the mean) and the 2nd raw moment (uncentered variance) of the gradient. A problem arises as these moving averages are initially 0-vectors, causing the moment estimates to be biased towards 0. This is countered by calculating bias corrected terms for $v_t$ and $m_t$ respectively [39]:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{A.6}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{A.7}$$

These terms are then used to construct Adam's update method.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{A.8}$$

Kingma and Lei-ba emiprically show in their paper that this method produces results better than their adaptive-learning competitors such as Adagrad and RMSprop [39].

Below follows the algorithmic procedure:

**Algorithm 1** Adam Algorithm

```
Require: $\alpha$: Step size
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$: Initialize $1^{st}$ moment vector
  $v_0 \leftarrow 0$: Initialize $2^{nd}$ moment vector
  $t_0 \leftarrow 0$: Initialize time step
  while $\theta_t$ not converged do
      $t \leftarrow t + 1$
      $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
      $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
      $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
      $\hat{m}_t \leftarrow m_t/(1 - {\beta^t}_1)$ (Compute bias-corrected first moment estimate)
      $\hat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
      $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$: Update parameters
  end while
  Return $\theta_t$ (Resulting parameters)
```

### A.1.1 AdaMax

Adamax is an extension of Adam. In Adam, the $v_t$ factor is scaling the gradient inversely to a scaled $L^2$ norm of the current and past gradients. This $L^2$ norm can be generalized to an $L^p$ norm in the following way:

$$v_t = \beta_2^p v_{t-1} + (1 - \beta_2^p)|g_t|^p \tag{A.9}$$

When $p$ is large, these norms can become numerically unstable, however Kingma and Lei-ba show that if $p \to \infty$ they become stable. This fact is used to derive a new update rule, based on the infinity norm constrained version of $v_t$, here called $u_t$, which is used in AdaMax.

$$u_t = \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty)|g_t|^\infty = max(\beta_2 \cdot v_{t-1}, |g_t|) \tag{A.10}$$

Note that the decay terms, $\beta_2$, are parameterized as $\beta_2^p$.
This then yields the AdaMax algorithm:

**Algorithm 2** AdaMax Algorithm

```
Require: $\alpha$: Step size
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$: Initialize $1^{st}$ moment vector
  $u_0 \leftarrow 0$: Initialize the exponentially weighted infinity norm
  $t_0 \leftarrow 0$: Initialize time step
  while $\theta_t$ not converged do
      $t \leftarrow t + 1$
      $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
      $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
      $u_t \leftarrow max(\beta_2 \cdot u_{t-1}, |g_t|)$ (Update the exponentially weighted infinity norm)
      $\theta_t \leftarrow \theta_{t-1} - (\alpha/(1 - \beta_1^t)) \cdot m_t/u_t$: Update parameters
  end while
  Return $\theta_t$ (Resulting parameters)
```

# B

# Appendix 2

## B.1 Operators

Here we provide a more in-depth and formal description of some of the basic operators mentioned in this thesis. The algebraic portions are heavily paraphrased from the supplementary material provided by Gulisano et. al for this paper [40].

***Map***
The *Map* operator is a generalized projection operator. It is defined by

$M\{A'_1 \leftarrow f_1(t_{in}), ..., A'_n \leftarrow f_n(t_{in})\}(I, O)$.

In this function $I$ and $O$ represent the input stream and the output stream respectively. $t_{in}$ is the input tuple to the operator and $A$ $(A'_1, ..., A'_n)$ is the schema of the output stream. The operator transforms the input tuple using the provided set of user-defined functions $\{f_1, ..., f_n\}$, in effect mapping the incoming tuple onto the schema of the outgoing stream. The incoming and outgoing schema may differ, however the outgoing tuple preserves the timestamp of the incoming one.

***FlatMap***
The *FlatMap* operator is an extended version of the Map operator. While the two are almost the same, the main difference is that a *FlatMap* outputs an arbitrary number of tuples for each incoming tuple. In practice, this means that *Map* operators can be considered a subset of the *FlatMaps*. *FlatMaps* vary somewhat based on implementation, but the usual behaviour is that 0, 1 or more output tuples are created based on each input tuple. Note that this could be 0 for the first tuple, then 4 for the next, meaning the *FlatMap* decides the number of output tuples based on the incoming one, rather than it being a chosen constant number.

***Filter***
The *Filter* operator is an operator that is used to either discard tuples based on some predicate, or split them into different outgoing streams. It is defined by

$F\{P_1, ..., P_m\}(I, O_1, ..., O_m, [O_{m+1}])$

In this function $I$ represents the input stream and $O_1, ..., O_m, [O_{m+1}]$ represents an ordered list of output streams. $P_1, ..., P_m$ is an ordered set of user-provided predicates. As can be seen in the function definition, the amount of predicates is either equal to the number of output streams, or there is one extra output stream. Each incoming tuple is forwarded to the first stream whose associated predicate it satisfies. In a more formal definition, $t_in$ is forwarded over $O_j$ where $j = \min_{1 \leq i \leq m} \{i \mid P_i(t_{in}) = TRUE\}$. If a tuple matches none of the predicates, it is either routed to the $O_{m+1}$ stream if it has been provided, or discarded entirely. It can be seen in this function definition that the incoming tuples are never altered, and as such the output tuple remains the same as the input tuple.

***Aggregate***
The *Aggregate* operator, as the name suggests, is used on windows of tuples to compute aggregate functions such as sums or averages. It is defined as :

$$Ag\{Wtype, Size, Advance, A'_1 \leftarrow f_1(W), ..., A'_n \leftarrow f_n(W),$$
$$[Group - by = (A_{i_1}, ..., A_{i_m})]\}(I, O)$$

In this definition, tuples on the incoming stream $I$ are stored in the window $W$ until it is full. The type of window is determined by $Wtype$ and can be either $Wtype = time$, which means the window is based on timestamps, or $Wtype = numTuples$, which means the window is based on tuple count instead. The parameter $Size$ determines the size of the window. When $Wtype = time$ a window is considered full if the difference in time between the incoming tuple and the first one in the window exceeds the value of $Size$. When $Wtype = numTuples$ a window is instead considered full if it contains $Size$ amount of tuples.

An Aggregate only produces output when the window is full (regardless of the type of window). The output tuples are sent over the output stream $O$ and use the timestamp of the **earliest** tuple in the window. The schema of the output tuple is represented by $\{A'_1, ..., A'_n\}$, and $\{f_1, ..., f_n\}$ represents the set of user-defined functions (e.g. average, count, sum, etc.) that are computed over all the tuples in the current window.

The update of the window happens each time an output tuple is propagated. In this step all stale tuples are discarded from the window according to the parameter *Advance*. For time-based windows ($Wtype = time$) a tuple $t$ is considered stale if, for an incoming tuple $t_{in}$, $t_{in}.ts - t.ts > Size$. If the window is count-based ($Wtype = numTuples$), the earliest *Advance* number of tuples will instead be considered stale.

The final part of the definition, *Group-by*, is optional, and is utilized to separate tuples into different windows based on some properties. Consider a case where *Group-By* $= A_i$ where $A_i$ is one attribute of the input schema. In this case, operators will be divided into separate windows for each possible value of $A_i$.

***Join***
The *Join* operator is used to, as the name suggests, join multiple streams together.

The operator is defined by:

$$J\{P, Wtype, Size\}(S_l, S_r, O)$$

It has an output stream denoted by $O$, and two input streams $S_l, S_r$ which are generally referred to as respectively *left* and *right.* $P$ denotes a predicate over a pair of tuples (one from the *left* and one from the *right*). $Wtype$ and $Size$ are parameters for the windowing similar to how they are defined for the *Aggregate* operator.

The operator keeps track of two separate windows, $W_l, W_r$, one for each of input streams. Tuples arriving on the *left* side are stored in the *left* window but are used to slide the *right* window, and vice versa for the *right* side. For time-based windows ($Wtype = time$) this means that on arrival of tuple $t_{in} \in S_l$, window $W_r$ is updated by removing all tuples $t$ such that $t_{in}.ts - t.ts \geq Size$. For count-based windows ($Wtype = numTuples$) on arrival of tuple $t_{in} \in S_l$, window $W_r$, if full, is instead updated by removing the earliest tuple.

After the window has been updated, for each tuple $t \in W_r$, the concatenation of $t_{in}$ and $t$ is produced as a single output tuple provided that the predicate $P(t_{in}, t)$ results in a positive (TRUE) outcome.

The updating of windows, evaluation of the predicate and propagation to the output for input tuples of the *right* stream are done in the same fashion (simply swap $r$ for $l$ and vice versa for the windows and input streams in every step).

***Reduce***
The *Reduce* operator continuously combine sequential tuples with the current result. We have only found implementation specific definitions and descriptions of this operator, and will as such not provide a formal definition, but instead an explanation and example based on documentation from java [41] and Flink [24].

This operator works by combining its "current" value (its last output tuple) and the incoming tuple using a set of user defined functions $f_1, ..., f_n$. Consider a case where the input sequence consists of tuples $[t_1, t_2, ..., t_m]$. The operator would output its first tuple once the first two tuples were received following. The output tuple would be constructed using this definition

$$O'_2 \leftarrow \{A'_1 \leftarrow f_1(t_1, t_2), ..., A'_n \leftarrow f_n(t_1, t_2)\}$$

In this case $O'_2$ denotes the tuple that is output on the arrival of $t_2$. For any other tuple it instead follows the more general definition

$$O'_i \leftarrow \{A'_1 \leftarrow f_1(O_{i-1}, t_i), ..., A'_n \leftarrow f_n(O_{i-1}, t_i)\}$$

where $i$ $(2 < i \leq m)$ is an arbitrary index of the incoming tuple. Reduce operators also have an optional $Group - by$ operation that works in the same way as for the *Aggregate* operator, but where the window only contains the last output tuple $O'_i - 1$ and the incoming tuple $t_i$.