Spark-based Application for Abnormal Log Detection

Georgios Koutsoumpakis
Abstract

Spark-based Application for Abnormal Log Detection

Georgios Koutsoumpakis

Ericsson is a world-leader in the rapidly-changing environment of communications technology and thus it is important to provide reliable and high quality networks. Automated test loops are executed frequently, trying to find problems in Ericsson’s products but, since test cases alone are not always adequate, machine learning techniques are sometimes used to find abnormal system behaviour. The Awesome Automatic Log Analysis Application (AALAA) tries to find such behaviour by checking the log files produced during the testing, using machine learning techniques. Unfortunately, its performance is not sufficient as it requires a lot of time to process the logs and to train the model. This thesis manages to improve AALAA’s performance by implementing a new version that uses Apache Spark, a general purpose cluster computing system, for both the processing of the data and for the training of the machine learning algorithm.
Contents

1 Introduction .................................................................................................................. 9
  1.1 Background & Motivation ....................................................................................... 9
  1.2 Problem Description ............................................................................................... 10
  1.3 Methodology .......................................................................................................... 10
  1.4 Thesis Structure .................................................................................................... 11

2 Background .................................................................................................................. 12
  2.1 Apache Spark .......................................................................................................... 12
    2.1.1 Architecture .................................................................................................... 12
    2.1.2 Resilient Distributed Datasets ........................................................................ 13
    2.1.3 Shared Variables ............................................................................................. 14
    2.1.4 Differences with Hadoop ................................................................................. 15
    2.1.5 MLlib ............................................................................................................... 16
  2.2 Awesome Automatic Log Analysis Application 2.0 ............................................ 16
    2.2.1 Overview ......................................................................................................... 16
    2.2.2 Logs ................................................................................................................ 17
  2.3 Self Organizing Map .............................................................................................. 17
    2.3.1 Parallel Self-Organizing Map .......................................................................... 19
    2.3.2 Batch Learning ............................................................................................... 20

3 Implementation .......................................................................................................... 22
  3.1 Feature Extraction ................................................................................................. 22
    3.1.1 Character bi-gram ......................................................................................... 23
    3.1.2 Time-stamp statistics ..................................................................................... 23
    3.1.3 Number of Lines ............................................................................................ 24
    3.1.4 Feature matrix ............................................................................................... 24
  3.2 Feature Transformation ........................................................................................... 24
    3.2.1 Normalization ................................................................................................. 24
    3.2.2 Dimensionality Reduction .............................................................................. 25
  3.3 Learning Algorithm ............................................................................................... 25
    3.3.1 Batch Learning on Spark ................................................................................. 26
    3.3.2 Sofm-Dist ....................................................................................................... 27
    3.3.3 Evaluating the trained model .......................................................................... 27

4 Results and Discussion ............................................................................................... 29
  4.1 Experiment Setup ................................................................................................... 29
  4.2 AALAA on Spark ..................................................................................................... 29
4.3 Comparison with previous AALAA ........................................... 30
4.4 One versus Three computers .............................................. 33
4.5 Execution on larger data ..................................................... 34
4.6 Advantages and Disadvantages ........................................... 35

5 Future Work ............................................................................. 37

6 Conclusion ................................................................................. 38

References .................................................................................. 39
List of Figures

Figure 2.1: Overview of Spark’s architecture ............................................. 13
Figure 2.2: Overview of AALAA 2.0 ................................................................. 17
Figure 2.3: Illustration of a 5-by-5 hexagonal Self Organizing Feature Map 18
Figure 3.1: AALAA’s architecture ................................................................. 22
Figure 4.1: Feature Transformation comparison between AALAA and AALAA on Spark (10 dimensions) ................................................................. 31
Figure 4.2: Feature Transformation comparison between AALAA and AALAA on Spark (22 dimensions) ................................................................. 32
Figure 4.3: Comparison of training between AALAA and AALAA on Spark (10 dimensions) ................................................................. 33
Figure 4.4: Comparison of training between AALAA and AALAA on Spark (22 dimensions) ................................................................. 33
Figure 4.5: Comparison of F-score between AALAA and AALAA on Spark ................................................................. 34
Figure 4.6: Comparison between using one and three computers ............ 34
Figure 4.7: Execution time on a larger training set .................................... 35
1. Introduction

This thesis was conducted at Ericsson. Ericsson is a world-leader in the rapidly-changing environment of communications technology - providing equipment, software and services to mobile and fixed network operators all over the globe. As some 40 percent of global mobile traffic runs through networks supplied by Ericsson, and more than 1 billion subscribers globally every day are managed by them [1], it is important to provide reliable and high quality networks. This is why it is important to make sure that Ericsson’s products are thoroughly tested and thus automated test loops are executed frequently, trying to find problems. Since test cases alone are not always adequate, machine learning techniques are sometimes used to find abnormal system behaviour.

1.1 Background & Motivation

Ericsson’s Continuous Integration Infrastructure (CII) department is constantly working on improving the automated test loops and the testing process. In continuous integrated systems, test cases are executed all the time in an effort to catch failures that were introduced to the system due to the frequent software or hardware changes. They, however, cannot be blindly trusted since it is rather difficult to create test cases that pick up all the possible errors. The log files that are produced during the test loop can, however, contain meaningful information and might help identifying abnormal system behaviour. Such log files are called abnormal logs.

These log files are usually many and thus distributed techniques are required for their processing. Apache Hadoop [2] is usually used to process them. Hadoop is an open source framework that contains both a distributed storage system and a distributed computing framework. It is based on the MapReduce model [3] that has two steps; the map step splits the problem into smaller sub-problems and the reduce step collects and combines the solutions of the sub-problems to form the final solution.

There is a research on the Hadoop ecosystem ongoing in Ericsson, and as a result, the Awesome Automatic Log Analysis Application (AALAA) [4] was created. AALAA, which is further described in section 2.2 tries to identify abnormal logs by using machine learning techniques. However, AALAA’s current performance is not sufficient for two reasons. Firstly, some of its parts are
still executed sequentially, and secondly, during the feature extraction, where meaningful information is extracted from the logs, several passes on the input data, which are stored on disk, are being done.

1.2 Problem Description

Automated log analysis is difficult and heavy in terms of computations tasks. Awesome Automatic Log Analysis Application tries to solve this problem by using Hadoop’s MapReduce in some of its sub-tasks. Unfortunately, the performance is not sufficient since it still requires several hours of training in order to select the appropriate parameters that lead to a model that performs well. This is directly linked to the fact that AALAA uses Hadoop only for extracting relevant features from the logs, while the dimensionality reduction using principal component analysis and the training of the neural network are not done in a distributed platform, but they are performed in a serial way.

Even though the training is done off-line, further reducing the training time can be definitely beneficial. Since the logs files two different test cases produce might not be of the same format, AALAA treats the log files of each test case separately and thus a model must be build for each test case.

This thesis tries to improve the performance and scalability of AALAA.

1.3 Methodology

Apache Spark [5], which is further described in section 2.1, is a rather new competitor in the Big Data field. Spark’s design is not tied to the MapReduce model, thus allowing it to be used in more applications, while it also promises to be able run up to 100 times faster than Hadoop’s MapReduce in certain cases [6]. Spark supports in-memory computing and performs much better on iterative algorithms, where the same code is executed multiple times and the output of one iteration is the input of the next one.

This thesis tries to improve the performance of AALAA by using Apache Spark as a computational engine for all parts mentioned in the previous section, namely the feature extraction, dimensionality reduction and training of a machine learning algorithm.

As there are currently no implementations of Self-Organizing Map, the machine learning algorithm used, that execute on Spark, this thesis work introduces in section 3.3 how they can be trained on a Spark cluster.
1.4 Thesis Structure

The rest of the report is structured as follows:

Chapter 2, provides the background information required for the understanding of the thesis. Firstly, it briefly describes Apache Spark and then it makes a description of the Awesome Automatic Log Analysis Application. Finally, it provides information about Self-Organizing Feature Maps.

Chapter 3 explains the implementation details behind the new Awesome Automatic Log Analysis Application. It describes how the feature matrix was created and how the machine learning algorithm can be trained on a Spark cluster.

Chapter 4 draws a comparison in terms of performance between the current and the Spark implementation of AALAA.

Chapter 5, provides interesting future work that could be done in order to further improve AALAA.

Finally, chapter 6 presents a conclusion of the thesis work.
2. Background

This chapter provides the background information one might need in order to understand this thesis work. In the beginning, a short introduction to Apache Spark is made, describing its architecture and its core features, while in section 2.2 the Awesome Automatic Log Analysis Application is described. Finally, in section 2.3 Self Organizing Feature Maps, the machine learning algorithm used in AALAA is explained.

2.1 Apache Spark

Apache Spark [5] is a cluster computing system. Spark started as a research project at the UC Berkeley AMPLab in 2009, and was open sourced in early 2010. In 2013, it became an Apache Project, and thus it became more popular and gained a larger community.

2.1.1 Architecture

Figure 2.1 displays the basic components when running an application across a Spark cluster. As shown in the figure, to run a Spark-based application three different parts are required:

**Driver program**

Driver Program is the core application which coordinates the execution across the cluster. It is an application written for the specific problem, for example an application that counts the number of times each words appear on a text. It first initializes a connection to the cluster, then it creates tasks and requests for the cluster to execute them.

**Worker Nodes - Executor**

Worker nodes refer to physical devices that can perform computations. On each worker node an executor task is running, which receives tasks from the driver program and executes them. Moreover, the executor handles the memory and the cache for this worker.
Cluster Manager

The cluster manager is responsible for handling all resources available on the cluster. CPU and memory are examples of such resources. When the driver program starts its execution, it connects to a cluster manager supported by Spark and it requests available executors on the working nodes. When these resources are allocated, the application is shipped to the executors so that the actual computations can be performed.

Spark is agnostic to the underlying cluster manager as long as Spark can acquire executor processes and those executor can communicate with each other. This allows Spark to run on top of existing clusters, alongside with existing applications like Hadoop. Moreover, it allows Spark to benefit from the cluster manager’s features. For example, when forming a cluster using the Apache Mesos cluster manager, Spark can dynamically share CPUs between different Spark applications. Other examples of cluster managers are:

- Spark Standalone – A basic cluster manager that is shipped along with the spark installation.
- Hadoop YARN – Hadoop’s resource manager

2.1.2 Resilient Distributed Datasets

The main idea behind Spark is a memory abstraction called Resilient Distributed Datasets (RDD) [8]. RDDs are collections of elements that are scattered across the cluster and on which operations can be executed in parallel.

Currently, two kind of collections are supported:
Parallelized Collections are distributed datasets that originate from existing collections of elements like an Array.

Hadoop Datasets are distributed datasets that were created by reading any file stored in the Hadoop Distributed File System (HDFS) [9] or other storage systems supported by Hadoop (including the local file system).

RDDs are immutable collections, meaning that there are no functions that can alter the contents of an existing RDD directly. If someone wants to change some of its values, an RDD containing the new values will be created instead. There are two types of operations that can be performed on RDDs:

**Transformations** create a new dataset starting from an existing one. For example, *filter* is a transformation that only keeps the elements that fulfil some criteria, while discarding everything else.

**Actions**, on the other hand, perform computations on the dataset and then return a value to the driver program. *Count* is such an action, which calculates the number of elements that exist in an RDD.

All transformations in Spark are lazy, meaning that they do not compute their results right away. Instead, all the transformations on an RDD are being tracked and those operations are executed only when an action is called. There are two main benefits of having such behaviour. To start with, it allows Spark to optimize the execution. For example, if Spark was to execute a *count* action after a *filter*, there is no reason to actually filter out the values first and then perform the count, but it can count them while filtering them on-the-fly. Secondly, tracking the transformations makes the RDDs fault tolerant, meaning that even if one computer stops working the whole RDD can still be available, without having to replicate the data. Even if there is a failure on a worker, the RDDs can still be recreated on the rest of the cluster as long as the original data are still available.

By default, the transformed RDDs are recomputed every time an action is performed on it. Spark supports maintaining computed datasets around on the cluster for faster access, without having to recompute them. It supports persisting in the memory (caching) and/or on disk. In case of memory persistence, if the cluster does not have enough memory to keep the whole RDD in the memory then some partitions of the RDD will be discarded from the cache, and they will be recomputed from the original data when required.

### 2.1.3 Shared Variables

Spark automatically delivers a copy of any variable of the driver program used inside a Transformation or an Action to the cluster. Since that might not be very efficient in case of large arrays, Spark also supports manually broadcast-
ing variables. These variables are copies and if their values are updated on the cluster, the driver will not retrieve their new values.

Additionally, Spark supports a special kind of shared variables called *accumulators*. From the perspective of the workers, accumulators are write-only variables that the executors can only add values to. On the other hand, the driver program can only read that value. This makes accumulators ideal for finding, for example, global sums in an RDD. Each worker will calculate its local sum and on the driver’s request the global sum will be computed.

### 2.1.4 Differences with Hadoop

As mentioned previously, Hadoop relies on the MapReduce model to perform computations. This model, which has two phases, namely Map and Reduce, requires the data to be handled as Key-Value pairs. Spark is not tied to this model, but it can support the MapReduce model using the functions `flatMap` and `reduceByKey`.

Additionally, the Hadoop implementation of the MapReduce model requires the input and the output of its phases to be stored on disk. This may not be a problem normally, but for example in cases of iterative algorithms it can affect the performance negatively. Since in iterative algorithms, like KMeans [10], the output of one iteration becomes the input of the next one, a hadoop based implementation requires multiple disk accesses. Spark on the other hand, can maintain the data in the memory, thereby reducing the execution time [8].

The frequent disk access is not a conceptual limitation of the MapReduce model, but rather it a Hadoop specific implementation in order to make the applications fault tolerant and resilient. In case of a node failure, the data are still available on disc and the process can easily continue. Instead of restarting the whole job, the computations can be continued on a healthy node.

Attempts have been made in order to implement a MapReduce engine that can work without frequent accesses on the disk. Shinnar et al. [11] implemented a MapReduce engine, called M3R, which can cache results in-memory, greatly improving performance on iterative algorithms. Unlike Hadoop, M3R caches the output of the reducers before writing the results on the disk, and, in case the same file is accessed at a later time, instead of loading the contents from disk it will read them from memory. Moreover, it supports temporary results, ones that will not be written on disk at all, but will be read directly in subsequent jobs. Unfortunately, in order for M3R to achieve that, it sacrificed resilience since it can not revive after a node failure. On the other hand, Spark’s RDDs not only allow caching of the results, but the they are also fault tolerant.
2.1.5 MLlib

MLlib [12] is a collection of machine learning algorithms that run on top of Spark. It contains algorithms for binary classification, clustering, linear regression, collaborative filtering and an gradient descent optimization. Unfortunately, MLlib doesn’t contain any methods to train artificial neural networks, like Self-Organizing Maps.

2.2 Awesome Automatic Log Analysis Application 2.0

Awesome Automatic Log Analysis Application (AALAA) [4] is Ericsson’s method for finding abnormal log entries. In this section more information about the log files AALAA uses are provided, as well as its overview of architecture.

2.2.1 Overview

In figure 2.2, an overview of AALAA’s workflow is presented. During the training phase, AALAA processes the old log files, creating a feature matrix that will be used in training the machine learning algorithm. This builds a model which will be used against future logs, the ones that we are interested in categorizing as normal or abnormal. These future logs, will go through the same feature extraction process, and then, using the previously trained model, will be identified as normal or abnormal.

AALAA consists of three different parts:

1. Feature Extraction, where only some relevant information about each log file is extracted. Example of such extracted features are character bigram [13] and timestamp statistics. In this step Hadoop is used both as a storage system (HDFS) and as a computation engine for extracting the features.

2. Feature Transformation, where normalization and dimensionality reduction with Principal Component Analysis [14] is performed on the extracted features, in an effort to reduce the size of the data.

3. Learning algorithm, where a Self Organizing Feature Map is trained on the data, and the produced model will be later used to identify abnormal log entries.

More information regarding these three parts as well as regarding how they were implemented on Spark are provided in chapter 3.
2.2.2 Logs

For each log file that is used for the training of the system, information about the state of the test case that produced it is also provided. This means that it is known if the specific test case passed or failed. Since it is not possible or easy to have experts provide logs that are considered abnormal for the training, the test case’s state is used for evaluating the different trained models. When a test case failed, its log file is considered abnormal. This way the system tries to predict logs files that are similar to logs of failed test cases.

2.3 Self Organizing Map

*Self-Organizing Maps* or *Self-Organizing Feature Maps* (SOFM) [15] is a type of artificial neural network. Trained using unsupervised learning, SOFM produces a low-dimensional, which typically is two-dimensional, discretized representation of the input space, called a map. The difference between Self-organizing maps and other artificial neural networks is that they use a neighbourhood function to preserve the topological properties of the input space, meaning that two points that are close in the input space, will also be close in the low-dimensional space.

*Figure 2.2. Overview of AALAA 2.0*
To further understand SOFM, the following terms should be explained:

**Neuron** refers to each node in the artificial neural network. Each neuron tries to "win" the training data in an effort to move closer to them.

**Weight Vector** Each neuron contains a weight vector. It belongs in same dimension as the input and it is represented as the weights of the connections between the input and output layer. This vector is used to calculate the distance between the neuron and a training point.

**Best matching unit** When the network is fed with a new training sample, all neurons compete for it by finding the distance of their weight vector with that input. The winner neuron, the one whose weight vector is closer to the training point, is called BMU.

A Self Organizing Map consists of two layers, the input and the output (or competitive) layer. The neurons on the output layer are placed in some topology, for example a hexagonal or rectangular grid. Figure 2.3 illustrates the output layer of a Self Organizing Map placed in a 5-by-5 hexagonal grid. Here, each hexagon is one neuron and the input of each neuron is connected to all neurons of the input layer.

![Figure 2.3. Illustration of a 5-by-5 hexagonal Self Organizing Feature Map](image)

The network is trained using competitive learning. The training is repeated for several iterations and one training iteration covers one pass though all the data. Each input \( x \) is assigned to the output neuron \( c \) closer to \( x \), the best matching unit. During the training, the winner’s weight vector is moved towards \( x \), so that it is more likely to win it again, in another iteration. Additionally, the winner’s neighbours according to the grid also get updated towards \( x \), depending on their distance from the winner. Given \( K \) is the number of output neurons, let \( w_k \) denote the weight vector for the \( k \)-th neuron. Equation 2.1 displays how the weights are updated at a given time \( t \).

\[
    w_k(t + 1) = w_k(t) + \eta h_{ck}(t)(x(t) - w_k(t)) \tag{2.1}
\]
Here, $\eta$ is the learning rate which controls the magnitude of correction to the weight vectors. $\eta$ is reduced monotonically during the training phase and $0 < \eta < 1$. The neighbourhood function $h_{ck}$ determines how much a training point will affect the winner’s neighbours. Since neurons closer to the winner should update more, usually, a Gaussian function is used.

Algorithm 1 offers an overview of the learning algorithm.

### Algorithm 1 Basic Learning Algorithm

**Input:** The number of neurons $K$, the training set $data$

**Output:** Weight vectors of neurons

initialize weight vectors and $\eta$

$t \leftarrow 0$

for all iterations do

reduce $\eta$

for all $x \leftarrow data$ do

$t \leftarrow t + 1$

compute winning node $c$

for $k = 1$ to $K$ do

compute $h_{ck}$

update weight vector $w_k$ according to eq. 2.1

end for

end for

end for

2.3.1 Parallel Self-Organizing Map

In the field of Machine Learning several attempts have been made to train an artificial neural network in a distributed or in general in a parallel way. Usually, there are two types of implementations [16, 17, 18]:

- network-partitioned, where the network itself is divided and then each CPU is assigned with only a portion it. Of course, communication is required for the different nodes during the training, since each node contains information only about its assigned part.
- data-partitioned, where rather than partitioning the neural network, a partitioning on the training data is performed. Here, every CPU maintains a copy of the whole network, which gets trained on its assigned data and finally, at the end of each training iteration the final network is computed.

Lawrence et al. [16] proposed a Message Passing Interface (MPI) implementation of Self Organizing Map for both data and network partitioned neural network. Their data partitioned network is trained using the batch learning
algorithm, which is further explained in section 2.3.2, and is similar to the approach used for Spark.

Similarly, Weichel [17] trained a data-partitioned SOM using Hadoop’s MapReduce, while Myklebust and Solheim [18] evaluate an additional hybrid implementation of data-network partitioning, which led to better performance.

2.3.2 Batch Learning

Let $w_k$ denote the weight of the $k$-th neuron, $x(t)$ the $t$-th training vector and $c$ the neuron which was the winner for that input. Also, let $t_0$ and $t_f$ be the state before and after one pass on all data is completed. The distance $d_c$ of the input with the winner node is calculated as depicted in equation 2.3, while the distance $d_k$ of each neuron with the input is calculated in equation 2.2. The main difference with the basic training algorithm is that, when selecting the winner neuron, the weight vectors of the previous iteration are used, instead of the previous input vector.

\[
\tilde{d}_k(t) = \|x(t) - w_k(t_0)\|^2 
\]  
\[
d_c(t) = \min \tilde{d}_k(t) 
\]  

The neighbourhood function $h_{ck}(t)$ controls the weight change each data point will bring to the whole network by allowing the winner node $c$ to move closer to the input than the others neurons. Let $r_k$ be the position of a neuron in the grid. In equation 2.4, $\|r_c - r_k\|$ is the distance of the winner node to the current node $k$. Additionally, $\sigma(t)$ should be a monotonously decreasing function, since as more iterations of the algorithm are repeated, the input vector should affect the nodes neighbouring the winner with decreasing effect.

\[
h_{ck}(t) = \exp\left(-\frac{\|r_c - r_k\|^2}{\sigma(t)^2}\right) 
\]  

Finally, the updated weights at the end of the iteration are calculated according to equation 2.5

\[
w_k(t_f) = \frac{\sum_{t'=0}^{t_f} h_{ck}(t')x(t')}{\sum_{t'=0}^{t_f} h_{ck}(t')} 
\]
Algorithm 2 Batch Learning Algorithm

Input: The number of neurons $K$, the training set $data$

Output: Weight vectors of neurons

Initialize weight vectors

for all iterations do
    initialize numerator and denominator of eq. 2.5 to 0
    for all $x \leftarrow data$ do
        compute winning vector according to eq. 2.3
        for $k = 1$ to $K$ do
            compute $h_{ckx} \text{ and } h_{ck}$ according to eq. 2.4
            accumulate numerator and denominator in eq. 2.5
        end for
    end for
    for $k = 1$ to $K$ do
        update weight vector $w_k$ according to eq. 2.5
    end for
end for

Algorithm 2 offers an overview of the batch learning algorithm.

The batch learning algorithm removes all the time dependencies the first training algorithm had, thus allowing the training to be performed on each training input independently.
3. Implementation

As mentioned in section 2.2, the Awesome Automatic Log Analysis Application consists of three different parts, namely the feature extraction, the feature transformation and finally the learning algorithm. Figure 3.1 displays an overview of the Spark-based implementation of AALAA. This chapter describes in more details how the three parts are implemented on Spark.

![Figure 3.1. AALAA’s architecture](image)

3.1 Feature Extraction

The feature extraction is the most important task when using machine learning techniques. Selecting features that can differentiate between normal and abnormal logs can lead to better results, while selecting irrelevant features will
lower the performance of the learning algorithm. Appropriate features will lead to improved model interpretability, shorter training times, and enhanced generalisation by reducing the chance of over-fitting, a situation where the model learned to recognise the specific training inputs instead of the underlying relationship [19].

This section describes how the feature matrix was created.

3.1.1 Character bi-gram

Character bi-grams is a very common feature when using machine learning techniques on free text [13]. Character bi-grams belong to the more generic n-gram. An n-gram is a sequence of \( n \) items taken from a string. In character bi-gram specifically, the sequence of all two adjacent characters is produced.

For example, in the string "Text" the following bi-grams are produced:

"Te", "ex", "xt"

In AALAA, for each log file, the number of times each bi-grams appears is calculated and then divided by the total number of bi-grams in that log, thus the feature matrix is added with the frequency of all possible bi-grams. Given that there are 98 possible characters that are accepted in AALAA, there are \( 98 \cdot 97 + 98 \) possible bi-grams. In the end, the total number of bi-grams that appeared in that log is also included in the feature matrix, totalling in a vector with 9605 values.

3.1.2 Time-stamp statistics

Log files contain time-stamp information for each of their entries. The total execution time of a test case can definitely help identify abnormal logs. For example, the fact that a test case was running for too long, for example for one hour, most likely indicates that something is not working properly and further investigation is required. Therefore the time difference between adjacent log entries are computed.

Let \( t(l, e) \) be the time-stamp for the entry \( e \) of the log file \( l \), \( |l| \) the number of entries in \( l \). If \( \Delta t(l, e) \) denotes the time difference between the entry \( e \) and \( e + 1 \), as shown in equation 3.1, then \( T_\Delta(l) \) denotes all time differences in the log file \( l \).

\[
\Delta t(l, e) = t(l, e + 1) - t(l, e)
\] (3.1)
Then the following statistics are calculated for each log and are added in the feature matrix:

- Average: \( \text{mean}(T_\Delta(l)) \)
- Variance: \( \text{var}(T_\Delta(l)) \)
- Sum: \( \text{sum}(T_\Delta(l)) \)
- Maximum: \( \text{max}(T_\Delta(l)) \)

3.1.3 Number of Lines

The length of a log file can also help identify abnormal logs. A very large log might not be considered normal. Therefore, the number of entries each log has is added in the feature matrix.

3.1.4 Feature matrix

To conclude, the feature matrix is constructed by concatenating all the above mentioned vectors. So, each vector of the matrix consists of:

\[ \text{[Character bi-grams | Timestamp statistics | Number of Lines]} \]

3.2 Feature Transformation

The step following the Feature Extraction is the Feature Transformation. Here the feature matrix is normalized and its dimensionality is reduced in an effort to improve the training process.

3.2.1 Normalization

A common problem is having some features with a large range of values, while others having rather small ones. For example, in AALAA’s feature matrix, the frequency of the character bi-grams are in \([0, 1]\), while the number of lines, of even the time-stamp statistics have no upper limit. In this case, the larger one becomes dominant.

To minimize this problem, data normalization [20] is performed on the feature matrix in order to make it have consistent statistical properties. The normalization is done by subtracting the average of each column of the feature matrix, and then dividing by the column’s standard deviation, thus, forcing the ranges to be in \([-1, 1]\) during training.
3.2.2 Dimensionality Reduction

The way the feature matrix was created, it is in a high dimensional space. This necessitates the reduction of the matrix since it might lead to dimensionality disaster [21], a case where the distance between two data points becomes meaningless. Additionally, having higher dimensionality increases the training time. Luckily, most of the character bi-grams never appear or appear rarely, and so most of the values in that matrix are zero, thus the matrix is sparse. We can safely reduce the dimensionality without sacrificing too much accuracy.

In this thesis, two different implementations of PCA using Singular Value Decomposition are used.

In the first one, the MLlib’s (version 1.0.0) implementation of PCA is being used. Unfortunately, there is official support only for matrices with rather low dimensionality, less than 1000. In AALAA’s case the feature matrix consists of 9610 columns and thus the performance is rather poor. The reason for using this implementation even though the performance might not be very good, is that it allows AALAA to train on large training sets, as the memory of one computer will no longer be the limit.

In the second implementation, the calculation of the principal components is done locally, using a serial implementation, but the whole cluster is utilized when calculating the reduced matrix. This hybrid approach is used unless mentioned differently and it performs better for smaller datasets.

3.3 Learning Algorithm

Since Spark’s Resilient Distributed Datasets are distributed collections of elements, a data-partitioned implementation seems more appropriate in order to efficiently train a self-organizing map. This will allow the training inputs to be scattered across the cluster and the training to be performed independently on each input vector.

An approach similar to the basic training Algorithm 1 can not be used. This is mainly because in the basic version, the weights update at each step, meaning that each input will affect the next ones. On the other hand, RDD operations can be applied on each element separately from the others. This necessitates the use of a batch learning algorithm, where the weights are updating only in the end of each iteration. This thesis follows the batch learning algorithm proposed by Lawrence et al. [16] and Weichel [17].
3.3.1 Batch Learning on Spark

A Spark based implementation is very similar to the Algorithm 2. The main difference is that now the input vector \( data \) is loaded as an RDD. Algorithm 3 shows how a Self Organizing Map can be trained on the cluster, using the batch learning algorithm. Since RDDs are distributed datasets any operations that are performed on top of them will be executed on the cluster and not on the driver program.

Given that the input vector’s length is large, most of the computation cost is on calculating the nominator and the denominator of equation 2.5 as it contains one pass though all the training data. Since in the batch learning algorithm each input vector doesn’t affect the others, each worker-executor can calculate independently the local sums of the input vectors that were assigned to it, and later, the global sum can be computed. As mentioned earlier, Spark supports shared variables called accumulators. Because accumulators are variables that the workers can only add values to they are ideal for calculating the nominator and the denominator.

**Algorithm 3 Batch Learning Algorithm on Spark**

**Input:** The number of neurons \( K \), the training set \( data \)

**Output:** Weight vectors of neurons

- initialize weight vectors
- load input vectors to an RDD called \( data \)
- for all iterations do
  - initialize the accumulators for numerator and denominator of eq. 2.5
  - for all \( x \leftarrow data \) do \( \triangleright \) RDD action: foreach
    - compute winning vector according to eq. 2.3
    - for \( k = 1 \) to \( K \) do
      - compute \( h_{ckx} \) and \( h_{ck} \) according to eq. 2.4
      - add their values to the accumulators
    - end for
  - end for
- retrieve the values of the accumulators
- for \( k = 1 \) to \( K \) do
  - calculate weight vectors \( w_k \) according to eq. 2.5
- end for

In this approach the weight vectors are stored locally. Since they are used inside an RDD action, Spark automatically sends them to all workers. Given that the number of neurons is rather small, while the number of training vectors is large, the fact that the calculation of the new weight vectors are done locally, on the driver program, should not reduce the performance too much.
3.3.2 Sofm-Dist

In order to categorize a log file as abnormal, an algorithm called Sofm-Dist is used. Sofm-Dist, which is heavily inspired from K-means, calculates the distance between one data point and the weight vector of the Best Matching Unit. Those logs that have distance values higher than some threshold, are considered as abnormal instances. This threshold, which is called distance threshold, is a parameter that critically affects the accuracy of the system. For high values, AALAA tends to categorize every log file as abnormal, while for small values, AALAA tends to categorize every log file as normal. Thus, several values must be tested in order to select an appropriate value for this parameter.

3.3.3 Evaluating the trained model

As mentioned in section 2.2.2, the training log are accompanied with information about its state; if the test case that produced that log file passed or failed. This information is used for evaluating the distance threshold and select an appropriate value. For comparing the different trained models, F-score [22] is used to evaluate AALAA’s results. Here, a brief introduction of the F-score is presented.

Firstly, the following terms are explained.

**True Positive (TP)** corresponds to the number of abnormal logs that were correctly categorized by the learning method as abnormal.

**False Negative (FN)** corresponds to the number of abnormal log files incorrectly predicted as normal logs.

**False Positive (FP)** corresponds to the number of the incorrectly categorized logs as abnormal logs.

**True Negative (TN)** corresponds to the number of normal logs that were correctly categorized.

Recall and precision are two widely used metrics for evaluating classifiers [22]. Precision, as shown in equation 3.3, on the one hand, measures the proportion of the abnormal logs with those categorized as abnormal logs by the learning method. Recall 3.2, on the other hand, measures the proportion of the abnormal logs that were correctly categorized. When having high recall, the learning method rarely miss-predicts abnormal logs as normal logs, while when having high precision, it rarely miss-predicts normal logs as abnormal ones. Recall and precision are defined as:

\[
r = \frac{TP}{TP + FN}, r \in [0, 1]
\]  

(3.2)
\[ p = \frac{TP}{TP + FP}, p \in [0, 1] \] (3.3)

Only maximizing one of them is however not enough. Therefore, a model that maximizes both recall and precision is needed. F-score, which defined in equation 3.4, tries to maximize both terms.

\[ F = \frac{2 \cdot r \cdot p}{r + p} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}, F \in [0, 1] \] (3.4)

Because the input is not very large, it is difficult to have a separate test set. That is why k-fold cross-validation is being used. The training set is divided into \( k \) parts and each time the \( i \)-th part is used as test set, while the rest are used to train the neural network. Then, the average F-score of the k-folds is computed and it is considered as the F-score for this trained model. Finally, the threshold that displayed the highest F-score is selected, and then a self organizing map is trained again, this time for all data.
4. Results and Discussion

4.1 Experiment Setup

For this thesis three computers were used to form a cluster. Since Spark requires one computer to work as a master, handling the connection with the driver program, the following set-up is being used. All three computers were used as slave nodes, performing the actual computations, while one computer is also used as a master node and as the executor of the driver program. All 3 computers are equipped with 32GB of RAM and one CPU Intel Xeon E5-2665 at 2.40GHz, which has 8 cores. All cores are available for Spark to use. Finally, the cluster manager used in this thesis is Spark Standalone.

For this thesis, the log files provided by Ericsson where used. The set consists of 1244 log files totalling in 343MB of raw data. Each log contains time-stamp for every one of its entries, which is required for calculating the time-stamp statistics.

Concerning the implementation’s parameters, the Self Organizing Map trained has a grid of 5-by-5 neurons. For distance thresholds, a total of 71 different values were tested in order to find the model that performs the best, similarly to how the previous AALAA performed. Finally, for evaluating the model, 10-fold cross-validation is used.

4.2 AALAA on Spark

In table 4.1, the execution time of the different components of the Spark-based implementation of AALAA is being presented. In this case, the target dimension of the PCA, meaning the dimension after the reduction, is 10. Although this might be excessive, since the number of dimensions drops from 9610 to 10, it still maintains 68% of the original information. The reason that this value is selected is because this is the target dimension that the previous AALAA uses. Unfortunately, the best average F-score found in this case was only 0.10.

In an attempt to improve the F-score, a target dimension of 22 in the PCA is tested. The reduced matrix now contains 76.8% of the original information. The results are present in table 4.2. As one can see, the training time is now
longer than in the previous case, while the feature extraction and transformation parts where not affected that much. In this case, the best average F-score was 0.3, which is much better than the previous case.

Finally, the results of a target of 100 dimensions is also presented in table 4.3. In this case, the 91.5% of the original information is maintained. As expected, the training time increased since it now requires more computations. Unfortunately, even though the reduced matrix has higher dimensionality, the F-score achieved was worse than for 22 dimensions, only 0.185.

4.3 Comparison with previous AALAA

The previous version of AALAA was executed on the same data. In table 4.4, the execution time of the previous version of AALAA is presented. Here, the results for both 10 and 22 dimensions after PCA are presented.

As shown in table 4.4, the Feature Extraction stage in the previous AALAA is much slower. There are several reasons for this behaviour. Firstly, the feature extraction was implemented using Hadoop. In that implementation several Map Reduce phases where used, each one requiring reading and writing
data on the disk. This affects negatively the performance. Moreover, the log files used have rather small size, below 2MB while Hadoop works better for large files. On the other hand, Spark allows to cache the data into the main memory requiring only one access the disk, given that the data can fit in the memory.

In the Feature Transformation stage the performance is slightly degraded. Since the implementation used is the hybrid one and the principal components are calculated locally, it needs to fetch the matrix from the cluster through the network, calculate the principal components and then send them back to the cluster to calculate the reduced matrix. This obviously comes with some additional overhead which is present in figures 4.1 and 4.2.

Looking at the figures 4.3 and 4.4, the total training time required in the previous AALAA is much higher than in the Spark-based implementation. This difference is mainly because of the number of training parameters that AALAA uses. As mentioned before, the new AALAA has only one training parameter that needs to tune, the distance threshold. On the other hand, since the previous AALAA uses the basic training algorithm, it had several more pa-

<table>
<thead>
<tr>
<th></th>
<th>Execution Time (10 dim)</th>
<th>Execution Time (22 dim)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Extraction</td>
<td>2 hours 44 min</td>
<td>2 hours 23 min</td>
</tr>
<tr>
<td>Feature Transformation</td>
<td>1 min 22 sec</td>
<td>2 mins 28 sec</td>
</tr>
<tr>
<td>Training</td>
<td>2 days 1 hour</td>
<td>4 days 14 hours</td>
</tr>
</tbody>
</table>

Table 4.4. Execution time when using the previous version of AALAA with 10 and 22 dimensions after PCA
parameters to iterate through and select appropriate values, like the neighbouring range, the learning range etc. This requires it to train a huge number of self-organizing maps in total, in an effort to pick the best set of parameters. On the other hand, since in AALAA on Spark the distance parameter doesn’t affect the training itself, only 10 SOFM are trained in total, one for each of the 10 folds. Moreover, in the previous AALAA the number of dimensions greatly affects the execution time since increasing the dimensions from 10 to 22, the required time increased more than two times, as the training was done in a serial way.

Finally, figure 4.5 displays the best F-score found for each version of AALAA. For AALAA on Spark, the best F-score was when 22 dimensions are kept after the PCA while in the previous AALAA, when 10 dimensions where used. The best average F-score that the previous AALAA succeeded was far better than the Spark-based implementation. What causes this difference is the fact that in the Spark implementation the batch learning algorithm is used, which apparently forms the clusters differently. Here, two points are worth mentioning. Firstly, the reason the Spark-based implementation received such a low F-score is the fact that it tends to categorize more logs as abnormal and not missing the true positives. Secondly, having lower F-score doesn’t necessarily mean that the new AALAA classifies logs incorrectly. Since the main idea of AALAA is to find abnormal logs and not only ones that failed the test cases, which the F-score is based on, there is a possibility that it performs better in that sense. Thus, further investigation should be made in these files to evaluate if they are actually abnormal or not.
4.4 One versus Three computers

In order to check how AALAA on Spark performs under different number of computers, the application was run on the 1244 logs using only one computer. The results are presented in figure 4.6.

Here, when using only one computer the performance is not degraded too much. While on 3 computers AALAA finishes its execution in 47 minutes, under only one computer it requires approximately 61 minutes. The reason for not scaling up that nicely is the fact that the log files are not large enough. The increased overhead of communicating over the network and the time spend scheduling the cluster is rather large compared to the actual tasks running on
Finally, in order to check the performance of AALAA in larger data, some artificial data were created. This data are 2.1GB or raw log files and had a total number of 7460 logs. In figure 4.7, the execution time of the different parts of AALAA are displayed. Here, the hybrid approach for the Principal Component Analysis didn’t manage to produce the reduced matrix and thus the MLlib implementation is used instead. The target dimension is again set as 22.

Concerning the Feature Transformation part, on the new dataset the required time increases only by a factor of 4 compared to the normal one. More-
over, the training requires now only 44 minutes, although in the original set it took 39. The reason that, even though the training data are now 6 times bigger than the original one, the required time only changes little in those two parts is that, now, the computations are more intensive. This benefits Spark since the additional computations cover some of the communicating and scheduling overhead and thus are hidden.

As it can be seen, the Feature Transformation part now takes a significant amount of time. The reason behind this increased time is the fact that the MLlib implementation of the Principal Component Analysis is used, which is optimized for *tall-and-skinny* matrices, meaning matrices that have rather small number of dimensions. Unfortunately, in AALAA’s case the number of dimensions is high.

![Figure 4.7. Execution time on a larger training set](image)

Since it is not easy to generate and try several different training sets, of different size, this thesis work made no attempts to see how AALAA performs in even larger datasets. It would be interested though for AALAA to be tested on larger sets and thus, it should be examined in some future work.

### 4.6 Advantages and Disadvantages

To sum up, compared to the previous AALAA, the Spark-based implementation of the Awesome Automatic Log Analysis Application has the following advantages:

- Much faster execution time
- Less training parameters to iterate through
- Scales better

Disadvantages:
- Lower F-score, even though this doesn’t necessarily mean that its prediction is worse than the previous AALAA.
5. Future Work

The Spark-based implementation of AALAA created for this thesis work definitely managed to improve its performance. In order to further improve AALAA the following should be done.

Firstly, more research should be done in order to understand why, when using the batch learning training algorithm for the Self Organizing Feature Maps, the prediction of the system doesn’t seem to perform that well in comparison to the previous system, but it seems to classify too many logs as abnormal. Just having better performance is not adequate for replacing the existing system and thus manual check of the logs should be done in order to verify if what the new AALAA finds as abnormal is correct.

Additionally, for the current thesis work the training set used was rather small in size. Since distributed techniques are appropriate for larger datasets it is interesting to check how AALAA performs under larger training data. Thus, more experiments should be done in order to understand how AALAA scales on big data.

Moreover, it would be interesting to check how AALAA behaves under different number of computers. In this thesis work AALAA was tested only under one or three computers. If AALAA is executed on big data more computational power might be needed in order for the performance to stay appropriate. So, further investigation should be done in that aspect.

Finally, two implementations of Principal Component Analysis were tested. The first one is a hybrid implementation that calculates the principal components locally and the second one is using the MLlib’s implementation of PCA, which unfortunately doesn’t work efficiently under matrices with many dimensions. That is why implementing a purely distributed PCA, one that works well for matrices with such a high dimensionality, might be useful to further improve AALAA’s performance.
6. Conclusion

To conclude, in this thesis work a Spark-based implementation of the Awesome Automatic Log Analysis Application was produced. AALAA, which tries to categorize log files as normal or abnormal, required too much training time. The Spark-based version of AALAA managed to outperform the previous one in terms of performance, since the whole process, including the feature extraction, the feature transformation and the training, now takes minutes instead of several hours. Unfortunately, this comes with lower accuracy in terms of what logs AALAA finds as abnormal, since the Spark-based implementation tends to categorise more logs as abnormal.
References