A Hadoop Map-Reduce Script for the creation of Hidden Markov Models

PAD - Distributed Enabling Platforms Course
1st Semester of Academic Year 2018-2019

March 17, 2019

Daniele Gadler
d.gadler@unipi.it
Department of Computer Science
University of Pisa
Pisa, Italy

ABSTRACT

Understanding the behavior of a system is a key step in the analysis of a system and represents a pre-requisite for the optimization of the usability and structuring of a software application. One of the most widely adopted approaches for capturing the usage of a system is via a statistical model (e.g: Markov Chains). A Markov Chain is a statistical model apt for capturing a system characterized by multiple states, with transitions among the different states occurring with a certain probability. Currently, there exists an automatic approach for building Hidden Markov Models programmed with the R programming language that, however, does not scale well to big data. In this report, I present a proof-of-concept Hadoop Map-Reduce script for creating Hidden Markov Models apt for crunching big data and evaluate its performance from a theoretical and experimental point of view. I also compare the performance of the Hadoop script with the performance of the existing R-based script. Experimental results carried out on dedicated parallel hardware show that the performance of the Hadoop script overcomes the performance of the R-based script when handling large quantities of data, whereas on a commodity laptop the performance of the Hadoop solution overcame the performance of the R-based solution for any amount of data considered. Finally, experimental results also indicate that the effective performance of the Hadoop script comes very close to its ideal theoretical performance, especially if handling a very large quantity of data. Hence, close-to-ideal scalability is attained.

1 INTRODUCTION

As part of my Bachelor thesis in Computer Science and Engineering at the Free University of Bolzano-Bozen, I worked on the creation of a method to automatically and iteratively build process models, capturing the intents of users while interacting with a software system. In such process models, intents are expressed as a set of actions performed by a user to a system to achieve specific use goals [8].

In my method, I applied the theory of Hidden Markov Models (HMMs). HMMs are Markov Chains where the system under study is assumed to have hidden states [10]. HMMs' *states* represent users' intentions, *symbols* represent unique user actions and *observations* consist of the overall set of interactions with the system under study.

My approach was inspired by Damevski et al. 's Interactive and Iterative approach [4]. Damevski et al. build IIHMMs (Interactive Iteratively built Hidden Markov Models), whose construction requires the interaction of a human expert to identify interesting sequences. My approach is aimed at automatizing Damevski et al.'s approach, by automatizing the process of creating HMMs, with no need for human intervention.

2 PROBLEM STATEMENT

The current approach for creating AIHMMs (Automatic Iterative Hidden Markov Models) described in [8] is currently implemented in R and was parallelized as part of the Paralell and Distributed Systems course (SPM - 2nd Semester of A.Y. 2017-2018) in order to process large quantities of data by means of a map-reduce mechanism [5]. However, native R is a rather slow and very high-level language, not well suited for efficient high-performance computing, in which fine-level tuning of lower-level mechanisms may need to be applied. Hadoop, on the other hand, is a Java-based framework, providing efficient higher-level programming mechanisms for cruching big data, while at the same time allowing for a tigher control of the objects, data types and mechasisms involved in the computation, specifically optimized for Map-Reduce programs. A Hadoop-based implementation of the AIHMM scripts would hence hence possibly allow for a more efficient utilization of the underlying resources wrt. an R-based implementation. In particular, the *Initialization Phase* described in 4.2.1 of the HMM costruction phase represents the major bottleneck of the program and was parallelized. Such Hadoop Map-Reduce implementation of a part of the initialization phase code for AIHMM generation is hereby presented.

3 RELATED WORK

The present work is mainly based on the work of Damevski et al. in [4], who apply an interactive and iterative approach for HMM construction for the discovery of developer debugging intentions based on interesting sequences. In their proposed approach, interesting sequences consist of sequences of symbols that can possibly give rise to a new state. Interpretation of interesting sequences is performed manually by a human expert, who decides whether a new state could better capture a set of interesting sequences. Interesting sequences consist of sequences that are well represented by the data, but not by the model. Should an expert decide to add a new state to the HMM based on the interesting sequences found, the symbols

timestamp	developer_id	message
2014-09-09 21:51:07	2653	View.Server Explorer
2014-09-09 21:51:09	2653	View.Properties
2014-09-09 21:51:16	2653	View.Solution Explorer
...	...	...

Table 1: A few rows from the BlazeData_Dev_2653.csv log file of the dataset

present in the interesting sequences picked will be constrained to that new state and a new iteration of the algorithm is triggered, where the HMM model is trained with the Baum-Welch Algorithm [12]. Originally, interactive HMM construction based on interesting sequences was proposed by S. Jaroszewicz in [9], who consider this approach as able to "produce accurate but easy to understand models". In my approach, instead, I mined usage logs of a system. Logs are one of the major data sources used in building prediction models in empirical software engineering [11]. System logs have been extensively used in diagnosing faults, scheduling applications and services of hardware [1, 7], or in dependable computing, and in computer networks management and monitoring [14, 17]. In software engineering, system logs have been employed to model the workflow design of the activity processes [15], like Petri nets or to predict system misbehavior [13]. There are few papers that mined usage logs of a system, as use logs are not automatically collected by modern logging services. Among these, Astromskis et al. [2] mined use logs to understand the usability of the same ERP system that was also considered in [8] by analyzing the frequency of users' chains of interactions.

4 BACKGROUND

4.1 Dataset

The dataset under study [3] consists of log files of developers' interactions with the Visual Studio IDE at the ABB Group ©. The dataset consists of 145 log files from as many developers. Each dataset row consists of a tuple with three columns: a timestamp, the ID of the developer that generated the interaction, and the message (e.g., View.Properties). Overall, the dataset has 8197261 rows and weighs 351.1 MBs. A few examples of entries in the dataset are given in Table 1.

4.2 HMM Construction Process

The procedure for AIHMM construction is made up of two phases: an *Initialization Phase*, and an *Iterative Phase*.

4.2.1 Initialization Phase. The Initialization Phase represents the program's bottleneck, as the sequential version of the initialization phase takes longer than one month to run with the dataset described in Section 4.1. The initialization phase is made up of the following phases:

(1) **Dataset loading and pre-processing**: A set of M independent datasets of developers' interactions with an IDE (observations) are loaded from the disk. From the loaded observations, outlier symbols are removed. Outlier symbols are all those symbols that are either too frequent or extremely unfrequent in the dataset.
(2) **Sequences' identification**: Messages in the set of observations loaded from the dataset are grouped into sequences (i.e., actions performed in a timeframe of no more than 30 seconds from one another carried out by the same developer) and a sequence ID is assigned to every sequence of time-contiguous messages. Too long sequences (i.e., lasting longer than 6000 seconds) and too short sequences (i.e: lasting less than 120 seconds) are discarded.
(3) **HMM initialization and training**: An HMM is initialized with one single state and unique actions in the dataset are taken as symbols of the HMM. Then, the HMM is trained with the Baum-Welch algorithm, a Maximum Likelihood Estimator algorithm, which takes the loaded sequences (observations) O from phase (2) as input and tunes the HMM's parameters so as to maximize $P(\theta|O)$, the probability of the observations O to be captured by the HMM model θ.
(4) **Sequences' Sorting**: After all sequences are identified in phase (2), these are sorted in lexicographical order to allow for efficient θ-frequent and θ-probable sequences' computation, as by [4].
(5) **θ-frequent sequences identification**: Sequences occurring very frequently in the dataset are identified according to Algorithm 2, as by [4].
(6) **Unconstrained HMM training and new log-likelihood computation**: In this step, an unconstrained HMM with two states is created and trained over the set of observations of the dataset. Furthermore, the log-likelihood of this newly created unconstrained HMM is computed and compared to that of the HMM created in phase (3).

4.2.2 Iterative Phase. The iterative phase is aimed at finding an HMM model characterized by the least possible log-likelihood (i.e: the best possible model). In every iteration, a new state is added, onto that symbols present in the interesting sequences are constrained. The iterative phase is made up of three sub-phases, which are outlined in the following paragraph. If either of the halting conditions outlined in the three sub-phases is met, the iterative phase stops.

(1) **Most interesting sequences' identification**: The most interesting sequences are identified based on the dataset's observations and the $\text{HMM}^{(n)}$, where n is the current iteration index. Interesting sequences are defined as sequences that are well captured by the dataset's observations, but are not well captured by the $\text{HMM}^{(n)}$ model [9] . If no interesting sequences are identified, the AIHMM generation phase stops and $\text{HMM}^{(n)}$ is returned.
(2) **Log-Likelihood comparison**: $\text{HMM}^{(n+1)}$ is created by adding a new state to $\text{HMM}^{(n)}$. The symbols present in the interesting sequences are then constrained to this new state. Afterwards, the log-likelihood of $\text{HMM}^{(n)}$ and $\text{HMM}^{(n+1)}$ are compared. L stands for log-likelihood of a model θ, given observations O, i.e: $L(\theta) = \log(P(\theta|O))$

$$\begin{cases} \text{if } L(\text{HMM}^{(n+1)}) > L(\text{HMM}^{(n)}): \quad \text{HMM}^{(n+1)} \text{ passed to (3)} \\ \\ \text{if } L(\text{HMM}^{(n+1)}) \leq L(\text{HMM}^{(n)}): \quad \text{HMM}^{(n)} \text{ is returned} \end{cases}$$

(3) **Comparison of constrained HMM$^{(n+1)}$ with unconstrained HMM$^{(n+1)}$**: In every iteration, after symbols identified in interesting sequences are constrained onto the new state of HMM$^{(n+1)}$, an unconstrained model of HMM$^{(n+1)}$ is created and trained (i.e: with no symbols constrained onto the newly added state). Then, the log-likelihood of these HMMs is compared. HMM$_C$ stands for "HMM Constrained", whereas HMM$_U$ stands for "HMM Unconstrained".

$$\begin{cases} \text{if } L(HMM_C^{(n+1)}) > L(HMM_U^{(n)}) : \text{HMM}_C^{(n+1)} \text{passed to Itr. } n+1 \\ \\ \text{if } L(HMM_C^{(n+1)}) \leq L(HMM_U^{(n)}) : \text{HMM}_C^{(n+1)} \text{is returned} \end{cases}$$

5 PARALLELIZATION METHODOLOGY

The program was parallelized according to the *Map-Reduce paradigm*. Mappers are used in steps (1), (2) of the *Initialization Phase*, whereas the reducer represents step (3) of the *Initialization Phase*. The Map-Reduce architecture adopted is shown in Figure 2.

Map-Reduce Modelling: Firstly, Log files are distributed to the different mappers by the Hadoop scheduler. Then, during the *map phase* at each mapper, their content is parsed in a sequence-by-sequence fashion and outlier messages are removed. Contiguous messages belonging to the same sequence are then grouped into sequences, and too long or too short sequences are filtered out. Finally, at the reducer, all sequences are merged into a set of sequences to be used to train an HMM and are output for further processing in phase (4) of the *Initialization phase*, where they need to be sorted.

Based on the Parallelization scheme outlined in Figure 2, I modelled the Completion Time of phases (1), (2), (3) as follows. The superscript of each step is the index of the phase considered.

$$T_{id}(n) = \mathbf{T^{1,2}_{Mapper(n)}} + \mathbf{T^{3}_{Reducer(n)}}$$

$T^{1,2}_{Mapper}$ was parallelized via a map and can be expressed as:

$$T^{1,2}_{Mapper(n)} = \frac{\sum_{i=1}^{M} TFile_i}{\frac{N}{n}}$$

Where M is the amount of datasets used, N is maximum amount of existing mappers, n is the amount of mappers in a job instance. $TFile_i$ is the time to load, pre-process and identify the sequences contained in the i-th log file.

$T^{3}_{Reducer}$ represents the ideal time required to gather all sequences at the reducer, initialize the HMM and train the initialized HMM with the loaded sequences.

6 IMPLEMENTATION DETAILS

In my HMM Hadoop Map-Reduce script[1], I implemented phase (1), (2), (3) of the *Initialization Phase* outlined in Section 4.2.1 whose architecture was described in Section 5. The implementation underwent three main versions, before achieving a final fully-functioning and optimized program in the third version.

6.1 Implementation Challenges

6.1.1 Full Log File processing. It is important to note that during phase (2) of the *Initialization Phase* described in Section 4.2.1, single log files need to be processed atomically (i.e., we cannot split a single log file's content among two or more mappers). In fact, if we were to split a log file's content among multiple mappers, a sequence's content would erroneously be assigned to two separate sequences. For this reason, I made use of the `WholeFileInputFormat` class in the `setInputFormatClass` and `WholeFileRecordReader`, as described by Tom White in [16], in order to process a single log file in a single mapper.

6.1.2 Sequence ID assignment. Sequence IDs need to be assigned to contiguous sequences of messages as described in phase (2). Because processing of different log files occurs in parallel, we must ensure that each sequence's sequence ID must be different from the other log files' sequence IDs. To this end, I made use of the developer ID contained in a single log file name (e.g., BlazeData_Dev_603.csv contains messages of a developer with ID 603) as initial sequence ID, multiplied by an upper bound of the amount of sequences present in a file (100000). This initial sequence ID is increased whenever a sequence's end is reached at every mapper.

```
Data: logFileName, sequences
multiplier ← 1000000
sequenceID ← findDevIDInFile(logFileName)∗ multiplier
sequencesIDs ← ∅
while end of file not reached do
    curMessage ← sequences[i]
    i ← i + 1
    sequencesIDs[i] ← sequenceID
    if curSequence == lastMessageInSequence then
        //last message of the current sequence
        sequenceID ← sequenceID + 1
    else
    end
end
```

Algorithm 1: Pseudocode algorithm for assigning a sequence ID to the messages in a log file, executed by every mapper in parallel

6.1.3 HMM Initialization and Training. In phase (3) of Section 4.2.1, an HMM is initialized with a single state and the symbols of the observations contained in sequences. After identifying sequences in phase (2), these are used in phase (3) of Section 4.2.1 to train an HMM with the Baum-Welch algorithm. For initializing an HMM and then training it, I made use of an open-source HMM implementation

[1]The code of my HMM Hadoop Map-Reduce script is available at www.github.com/DanyEle/Hadoop_HMM

M = Amount of datasets
N = amount of mappers
$F_0,\ldots, F_{M-1}$ = Datasets' filenames and sizes
$D_0,\ldots, D_{M-1}$ = Loaded datasets by decreasing size
$E_0,\ldots, E_{M-1}$ = Sequences identified
G = Sorted sequences
$G_0,\ldots, G_{N-1}$ = Partitions of sorted Sequences
H = θ-frequent sequences

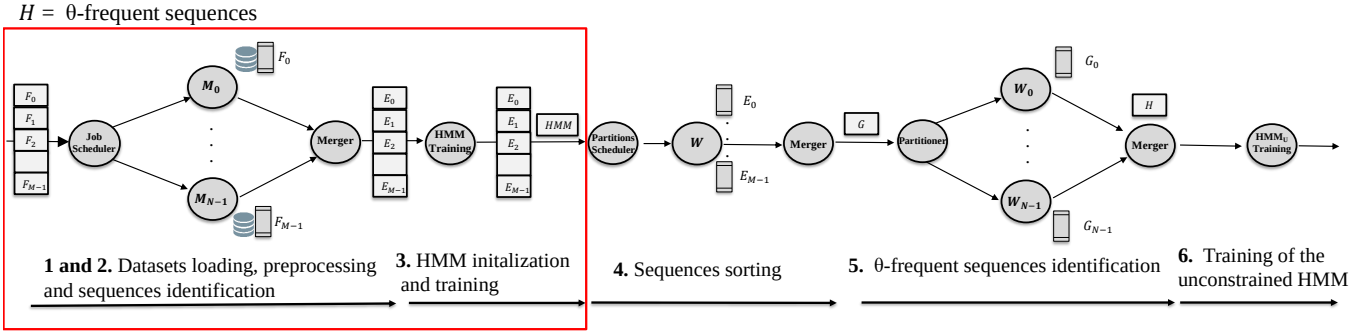

Figure 1: Scheme of the full parallelization of the *Initialization phase* for HMM initialization and training, implemented in the R-based script of [8]. Only the part highlighted in red was actually implemented in the Hadoop-based version of my script and it is shown in more detail in Figure 2.

M = Amount of datasets
N = amount of mappers
$F_0,\ldots, F_{M-1}$ = Log files
$E_0,\ldots, E_{M-1}$ = Sequences identified
$M_0,\ldots, M_{N-1}$= Mappers

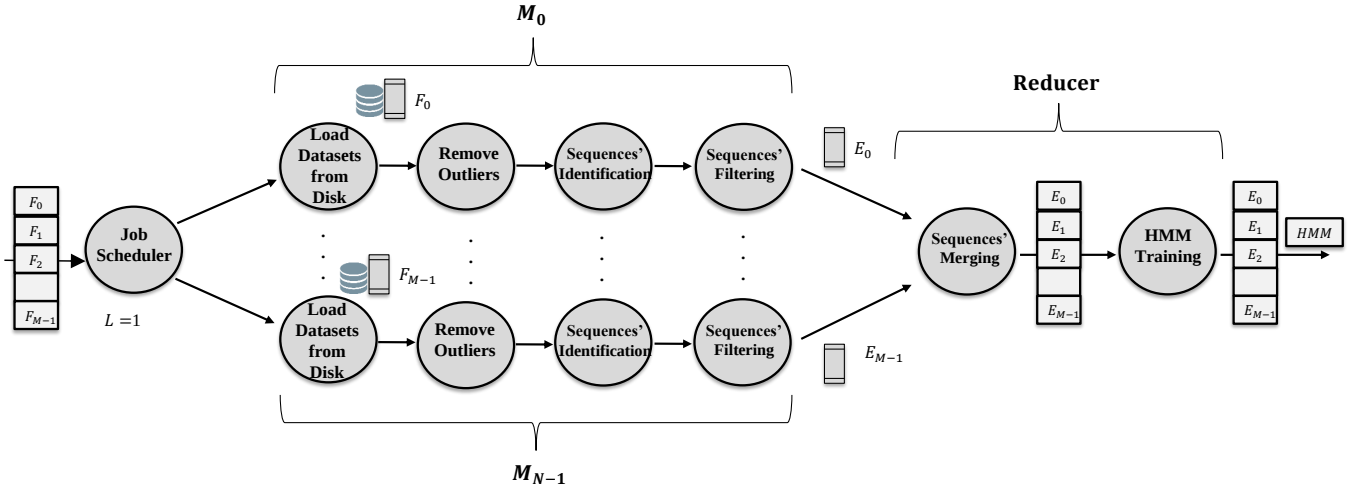

Figure 2: Scheme of the Map-Reduce parallelization employed for phases (1), (2) and (3) of the *Initialization phase* implemented in my Hadoop-based script.

in Java by Haifeng Li [6], whose code was tuned to fit my data format.

6.2 Version 1 - File name as Mapper key

- **Map Phase**: In an initial version, I made use of the log file name as *key* and all the sequences identified in the file were output as *values*, as shown in Figure 3. Also, the whole file content was fully loaded into a single String before being processed.
- **Reduce Phase**: In the reduce phase, all the sequences associated to all keys received by the reducer are merged into one single set of sequences. In this reduce phase, an HMM is initialized with one single state and the set of sequences

received from all mappers is used to train the newly created HMM with the Baum-Welch algorithm. In this reduce phase, received sequences are also output for further processing in the following phase, as described in phase (4) of the *Initialization Phase*.

Issues: The Version 1 implementation version rose a Java heap space error when processing more than two files at the mappers because each file content was stored into a string and the whole file acted as a key being passed from the mapper to the reducer, leading to very unefficient heap space management.

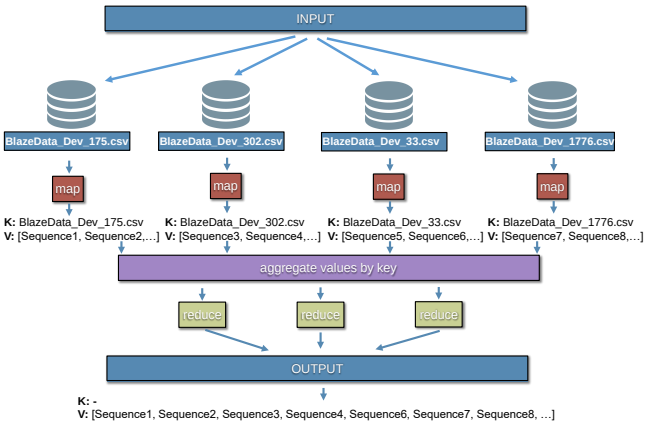

Figure 3: Overview of version 1 of the Map-Reduce parallelization described in 6.2.

6.3 Version 2 - Sequence ID as Mapper key

- **Map Phase**: The sequence ID is now the *key* of the map phase, and the messages in the sequence represent the *value* output by the map phase. However, also in this implementation, the whole file was still stored in a string before being processed.
- **Reduce Phase**: The reducer now takes as keys the sequences' sequence IDs, whereas the values received at the reducer consist of all messages lying in a certain sequence, which are gathered into a single set. The HMM is initialized and trained as described in Section 6.2, and the received sequences, together with their sequence IDs, are output for further processing in phase (4) of the Initialization phase. Possibly, if no HMM were to be trained in this phase, multiple reducers could be used.

  Issues: The Version 2 implementation improved over the performance attained by Version 1, described in Section 6.2, allowing to process tens of log files in parallel, provided each log files was no larger than 1 MB of size. If larger files were attempted to be parsed, these would cause a Java heap space error. The reason for such issue was tracked down to be the loading of the full log file content into a String.

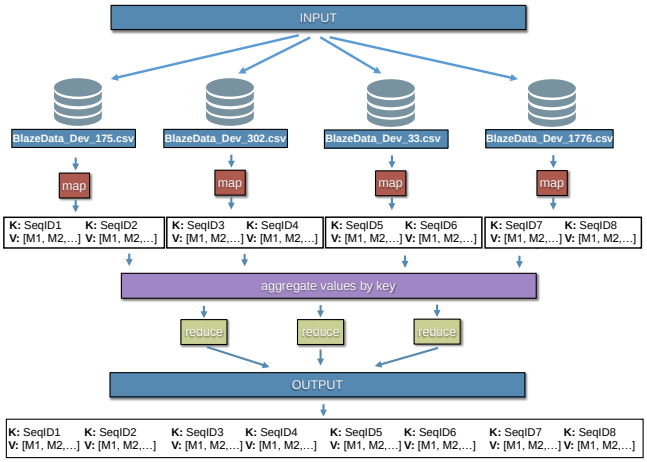

Figure 4: Overview of version 2 of the Map-Reduce parallelization described in 6.3.

6.4 Version 3 - Optimized File Parsing

- **Map Phase**: I optimized the file-parsing process to avoid loading a full log file into a string by parsing the log file in binary format character-by-character and keeping in memory only the very essential information required to process a sequence. From a theoretical point of view, the program has been transformed into a *data-stream* parallel program in the map phase, in which the stream is represented by every single line of the file being parsed . Furthermore, this code properly exploits the streaming-based data access pattern of Hadoop. The steps of this new data-stream approach are depicted in Algorithm 2.

```
Data: logFile in binary format
curLine ← ∅
while not at end of this log file do
    curCharacter ← (char) logFile[i]
    if curCharacter == '\n' then
        //end of the current line has been reached
        process(curLine)
    else
        //end of the current line not yet reached.
        curLine ← curLine + curCharacter
    end
    i ← i + 1
end
```

Algorithm 2: Algorithm for the efficient parsing of the sequences within a log file in $\Theta(n)$ time and requiring $O(1)$ space .

Algorithm 2 parses the log file character by character and runs in $\Theta(n)$ time. Its space occupancy is $O(1)$, as the only data being stored is the line currently being parsed (curLine), the character currently being parsed (curCharacter), along with other singleton variables for keeping track of the sequenceID, the last valid timestamp seen and the duration of the current sequence (which have been omitted in the pseudocode for sake of clarity). This $O(1)$ space occupancy

is a large improvement over the $O(n)$ space occupancy of Version 1 and 2, in which the whole file was loaded into a string. Such optimization is based on the intuition that we do not actually need to keep the whole log file in memory all the time, but simply need to keep track of the most updated information needed to process the current sequence adequately whenever the end of a line is reached. The sequence-wise processing of files in the map phase was maintained as that of Version 2, so the *key* and the *value* of the map phase are the same as those of Version 2 described in Section 6.3.

- **Reduce Phase**: See the reduce phase of Version 2 in Section 6.3.

7 EXPERIMENTAL EVALUATION

In this section, I compare the performance of phases (1), (2) of the *Initialization Phase* of the Hadoop Map-Reduce script for phases (1) and (2) of the program. Phase (3), involving the creation and training of an HMM, was not evaluated because both the R-based and the Hadoop-based versions are dependent on an external library, whose performance would substantially impact the overall runtime. In fact, the R-based version for HMM training is way slower than the Hadoop-based version and this fact would potentially skew the experimental results.

7.1 Initial approach - Fixed data, variable mappers

In an initial attempt at evaluating the performance of the Hadoop Map-Reduce program, I tried to keep the data fixed (i.e. all 145 datasets) and changed the amount of mappers processing the data. This approach was possible for the R-based version, but was not possible in the Hadoop-based version: despite my efforts at changing the amount of mappers processing data at the same time in Hadoop, no solution was found to change the amount of mappers in a Hadoop program, as this amount depends on the amount of files being passed as input. No way was found to change the amount of virtual cores (vcores) either, as Linux commands to set the amount of cores allocated to a certain Hadoop script did not have any effect (`coreset - c`). Therefore, a different approach was adopted to evaluate the Hadoop-based version, which is described in Section 7.2.

7.2 Final approach - Variable data, variable mappers

Hadoop Version: Following the observations drawn in Section 7.1, I looked for other approaches to evaluate the Hadoop-based version with a varying amount of mappers. Namely, I noticed that the amount of mappers in Hadoop was equal to the amount of map tasks submitted because of the File-wise `inputSplitFormat` adopted. Consequently, I decided to vary the amount of map tasks (i.e.: log files being given as input to the program), which would automatically imply a certain amount of mappers in the Hadoop-based version.

$$n \text{ map tasks} \Rightarrow n \text{ mappers}$$

	Local Laptop	**Xeon Phi**
CPU	Intel core i7-8650u 1.8 GHz, 8 cores	Intel Xeon Phi x20 (Knights Landing) 1.3GHz, 256 cores
RAM	16 GBs	46 GBs

Table 2: Hardware configurations used to test the Hadoop-based version and the R-based version of the program for phases (1) and (2) of the *Initialization phase*.

R-based Version: In the R-based version, I manually changed the amount of mappers based on the amount of map tasks (i.e. : log files) being passed as input to the R-based program.

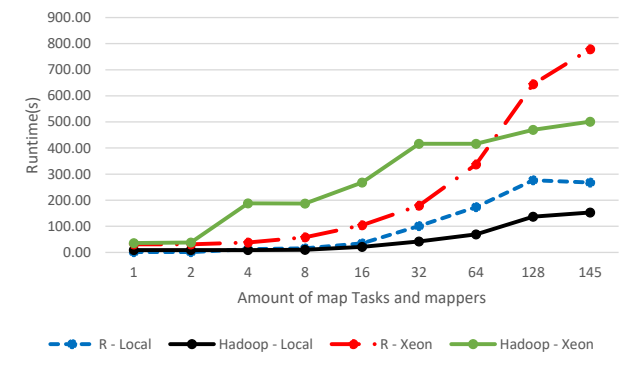

Figure 5: Runtime of the R-based program and the Hadoop-based program tested on the hardware configuration outlined in Table 2

7.3 Performance Comparison

Finally, I evaluated the *R-based program* and the *Hadoop-based program* on the hardware configurations shown in Table 2. Figure 5 reports the runtimes obtained by running the *R-based version* and the *Hadoop-based version* with $n \in [1, 2, 4, 8, 16, 32, 64, 128, 145]$, where n is the amount of log files passed and the amount of mappers used. Both the R-Based program and the Hadoop-based program were tested on the same log files and performed the same actions (i.e: phase (1) and (2) of the Initialization Phase), with Hadoop running in pseudo-distributed mode.

Xeon Phi: From Figure 5, it is remarkable but unsurprising to note that the runtime of the Hadoop-based version overcomes the performance of the R-based version when applied to a large amount of log files ($n \geq 128$) on the Xeon Phi machine. The motivation for this fact is two-fold: on the one hand, the poorer performance of Hadoop with $n < 128$ is due to the large overhead of Hadoop's HDFS in distributing and handling files. On the other hand, this result is imputable to the large sequential computation involved in each mapper when assigning sequence IDs, which is upper bounded by the mapper that is assigned the largest log file. Such sequential computation is run slowly by the 1.3 GHz CPU of the Xeon Phi. Furthermore, Hadoop is a framework optimized for achieving high throughput with big data, though this is generally balanced by a higher latency.

Local Machine: Both the Hadoop-based version and the R-based version running on my local laptop outperform the performance of the same versions ran on the Xeon Phi by a significant amount. This is because the 1.8 GHz clock of my CPU is about 28% faster than the 1.3 GHz clock of the Xeon Phi; furthermore, the local Intel Core i7 is more suitable for regular sequential computations than the Xeon Phi CPU, which is specifically optimized for handling parallel computations that exploit all 256 cores intensively. In my Hadoop-based program and in my R-based program, instead, not all 256 cores of the Xeon Phi can be fully and intensively exploited all the time because of two reasons. 1) Only 145 log files exist and one mapper (and core) is assigned to the processing of a single log file. 2) Log Files are characterized by a high variance in size, consequently leading to a high variance in the number of messages contained therein, as shown in Figure 6. Therefore, a worse performance is attained when running the R-based version and the Hadoop-based version on the Xeon Phi than on my local machine because of the strong impact of the sequential computation.

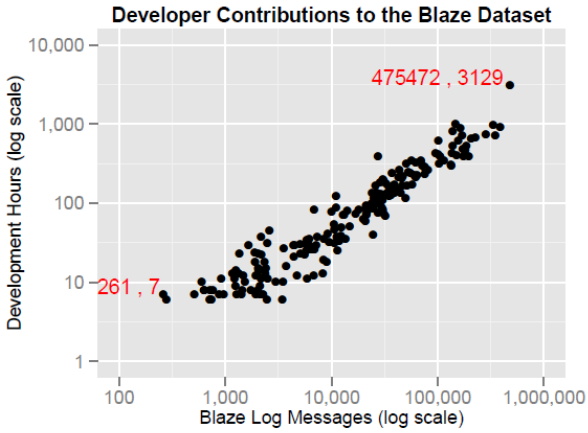

Figure 6: The collection interval and number of messages for each of the 145 developers' log files in the dataset.

The high variance in log files' size reflects onto the variable processing time required to process each file's content and does not allow to fully utilize all cores available all the time. (i.e.: with 128 cores, very small files will be processed straight away and the cores assigned to their processing will stay idle thereafter).

Figure 8 and Figure 7 show the utilization of the Xeon Phi's cores when running respectively the Hadoop-based version and the R-based version. It is worth noting that the Hadoop-based version uses very little resources in each CPU, whereas the R-based version makes a very intensive usage of the underlying resources. Yet, the Hadoop-based version outperforms the R-based version on both hardware benchmarks when running with $n \geq 128$.

7.4 Ideal and Effective Service time

The ideal service time $T_{id}(n)$ of the Hadoop program running with n mappers and n map tasks is defined in Formula 5 of Section 5. Namely, $T_{id}(n)$ represents the ideal time we would obtain if

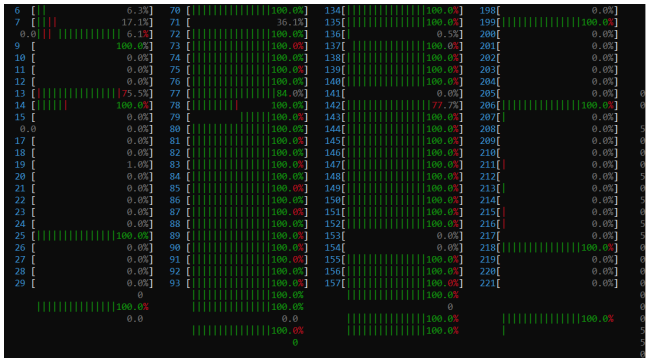

Figure 7: The CPU usage of the R-based implementation running on the Xeon PHI machine with n = 128

Figure 8: The CPU usage of the Hadoop-based implementation running on the Xeon PHI machine with n = 128

perfect scalability were achieved when increasing the amount of workers and map tasks. $T_{eff}(n)$ represents the effective service time, namely the actual service time obtained when running the job with n mappers and map tasks. The ideal and effective service time of the different phases is s shown in Table 3 when running the Hadoop program on the local machine by averaging three runs, whereas Table 4 shows the ideal and effective service time when running the Hadoop program on the Xeon Phi by averaging three runs.

By observing Figure 9 and Figure 10, we notice that the effective service time of the Hadoop version closely matches the ideal service time both when running in the local machine and on the Xeon Phi, especially when increasing the amount of map tasks and mappers beyond 128. This is explained by the fact that Hadoop is a framework specifically built for achieving excellent scalability and high throughput when handling large quantities of data..

8 CONCLUSIONS

In the present report, a Hadoop Map-Reduce script for creating and training Hidden Markov Models was presented. Firstly, an introduction to the problem at hand was given in Section 1 and 2. Then, the related work inherent to the construction of HMMs was presented in Section 3 and background information related to the

n	$T^{Effective}_{Mapper(n)}$	$T^{Ideal}_{Mapper(n)}$	$T_{Reducer(n)}$	$T^{Effective}_{Total(n)}$	$T^{Ideal}_{Total(n)}$
1	8	0.98	1	9	1.98
2	8	1.96	1	9	2.96
4	10	3.92	1	11	4.92
8	12	7.83	1	13	8.83
16	19	15.67	1	20	16.67
32	35	31.34	2	37	33.34
64	67	62.68	3	70	65.68
128	133	125.35	3	136	128.35
145	142	142.00	5	147	147.00

Table 3: Effective Service time of the Mappers ($T^{Effective}_{Mapper(n)}$), Ideal Service time of the mappers ($T^{Ideal}_{Mapper(n)}$), time spent in the reducer phase($T_{Reducer(n)}$), Total Effective service time ($T^{Effective}_{Total(n)} = T_{Reducer(n)} + T^{Effective}_{Mapper(n)}$), Total Ideal Service Time ($T^{Ideal}_{Total(n)} = T^{Ideal}_{Mapper(n)} + T_{Reducer(n)}$) for the Hadoop program ran on the local machine. All times are given in seconds.

n	$T^{Effective}_{Mapper(n)}$	$T^{Ideal}_{Mapper(n)}$	$T_{Reducer(n)}$	$T^{Effective}_{Total(n)}$	$T^{Ideal}_{Total(n)}$
1	26	4.86	2	28	6.86
2	26	9.72	2	28	11.72
4	30	19.45	5	35	24.45
8	50	38.90	5	55	43.90
16	87	77.79	6	93	83.79
32	148	155.59	9	157	164.59
64	332	311.17	12	344	323.17
128	655	622.34	18	673	640.34
145	705	705.00	19	724	724.00

Table 4: Effective service time for the mappers, ideal service time of the mappers, time spent in the reducer phase, total effective service time, total ideal service time for the Hadoop program ran on the Xeon Phi. The same formulas' meaning applies as those of Table 3.

very procedure to construct an HMM in an iterative manner was presented in Section 4. In Section 5, the theoretical methodology for parellelizing the construction of Hidden Markov Models by means of a Map-Reduce program was presented. Instead, the the Hadoop Java-based implementation details were described in Section 6. Finally, in Section 7, the performance of the Hadoop Map-Reduce program was compared with a pre-existing R-based version of the program for different sizes of the dataset. Experimental evaluation on dedicated parallel hardware and on a commodity laptop showed that the performance of the Hadoop implementation overcame that of the R-based implementation in case the dataset used is large, whereas on a commodity laptop the Hadoop-based implementation outperformed the R-based implementation for any dataset size. The effective service time of the Hadoop version was also shown to be very close to the ideal one, and the Hadoop implementation proved to utilize the underlying CPU resources in a more efficient way than the R-based version, leading to a consequent lower energy consumption.

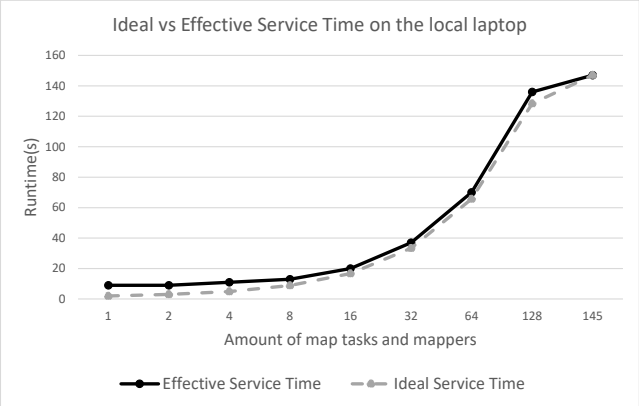

Figure 9: The ideal vs the effective service time of the Hadoop program when run on the local machine for $n \in [1, 2, 4, 8, 16, 32, 64, 128, 145]$

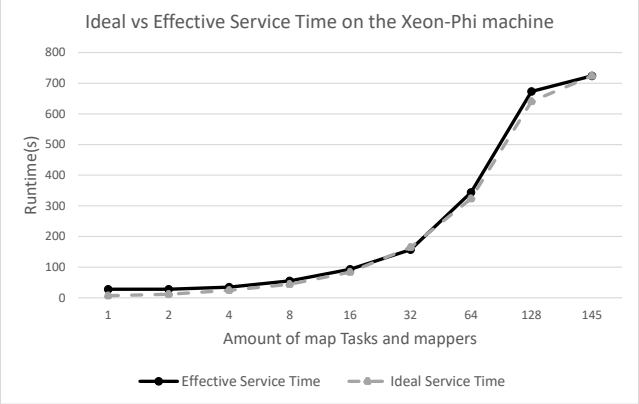

Figure 10: The ideal vs the effective service time of the Hadoop program when run on the Xeon Phi machine for $n \in [1, 2, 4, 8, 16, 32, 64, 128, 145]$

9 INSTRUCTIONS TO RUN THE HMM MAP-REDUCE PROGRAM

(1) Log into the XeonPhi machine via SSH.
(2) Navigate to my home folder:
`cd /home/spm18-gadler`
(3) Start Hadoop:
`./start_hadoop.sh`
(4) Adjust the amount of datasets to be used by the HMM Map-Reduce program by adding or removing datasets from the folder "/home/spm18- gadler/ABB_Logs"
(5) Load the datasets in the ABB_Logs folder onto Hadoop's HDFS:
`hadoop fs -put ABB_Logs ./`
(6) Navigate to the code project folder:
`cd HMM_MapReduce`
(7) Actually run the HMM Map Reduce program, passing the folder with the input log files on the HDFS to the run_program.sh

script:

```
./run_program.sh ABB_Logs
```

REFERENCES

[1] 2007. *Exploring event correlation for failure prediction in coalitions of clusters.* https://doi.org/10.1145/1362622.1362678

[2] Saulius Astromskis, Andrea Janes, and Michael Mairegger. 2015. A Process Mining Approach to Measure How Users Interact with Software: An Industrial Case Study. In *Proceedings of the 2015 International Conference on Software and System Process (ICSSP 2015)*. ACM, New York, NY, USA, 137–141. https://doi.org/10.1145/2785592.2785612

[3] Kostadin Damevski. 2013. ABB Visual Studio Developer Interaction Dataset. https://abb-iss.github.io/DeveloperInteractionLogs/. (2013).

[4] Kostadin Damevski, Hui Chen, David Shepherd, and Lori Pollock. 2016. Interactive Exploration of Developer Interaction Traces Using a Hidden Markov Model. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 126–136. https://doi.org/10.1145/2901739.2901741

[5] Barbara Russo Daniele Gadler, Andrea Janes. 2017. R Scripts for the creation of AIHMMs and IIHMMs. https://github.com/danyele/HMM. (2017).

[6] Haifeng Li et al. 2019. Smile - Statistical Machine Intelligence and Learning Engine. (Feb. 2019). https://haifengl.github.io/

[7] Errin W. Fulp, Glenn A. Fink, and Jereme N. Haack. 2008. Predicting Computer System Failures Using Support Vector Machines. In *Proceedings of the First USENIX Conference on Analysis of System Logs (WASL'08)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1855886.1855891

[8] D. Gadler, M. Mairegger, A. Janes, and B. Russo. 2017. Mining Logs to Model the Use of a System. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 334–343. https://doi.org/10.1109/ESEM.2017.47

[9] Szymon Jaroszewicz. 2010. Using interesting sequences to interactively build Hidden Markov Models. *Data Mining and Knowledge Discovery* 21, 1 (01 Jul 2010), 186–220. https://doi.org/10.1007/s10618-010-0171-0

[10] Daniel Jurafsky and James H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

[11] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and Challenges in Log Analysis. *Commun. ACM* 55, 2 (Feb. 2012), 55–61. https://doi.org/10.1145/2076450.2076466

[12] Lawrence R. Rabiner. 1990. Readings in Speech Recognition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, 267–296. http://dl.acm.org/citation.cfm?id=108235.108253

[13] Barbara Russo, Giancarlo Succi, and Witold Pedrycz. 2015. Mining system logs to learn error predictors: a case study of a telemetry system. *Empirical Software Engineering* 20, 4 (2015), 879–927. https://doi.org/10.1007/s10664-014-9303-2

[14] M. Steinder and A. S. Sethi. 2004. Probabilistic fault localization in communication systems using belief networks. *IEEE/ACM Transactions on Networking* 12, 5 (Oct 2004), 809–822. https://doi.org/10.1109/TNET.2004.836121

[15] Wil van der Aalst. 2012. Process Mining: Overview and Opportunities. *ACM Trans. Manage. Inf. Syst.* 3, 2, Article 7 (July 2012), 17 pages. https://doi.org/10.1145/2229156.2229157

[16] Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[17] Kenji Yamanishi and Yuko Maruyama. 2005. Dynamic Syslog Mining for Network Failure Monitoring. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD '05)*. ACM, New York, NY, USA, 499–508. https://doi.org/10.1145/1081870.1081927