Received: October 27, 2019. Revised: November 18, 2019. 214

International Journal of
Intelligent Engineering & Systems

http://www.inass.org/

A New Improved Baum-Welch Algorithm for Unsupervised Learning for Continuous-Time HMM Using Spark

Imad Sassi[1]* **Samir Anter[1]** **Abdelkrim Bekkhoucha[1]**

Computer Science Laboratory of Mohammedia (LIM), FSTM,
Hassan II University, Casablanca, Morocco
* Corresponding author's Email: imadsassi7@gmail.com

Abstract: Hidden Markov Models are widely used for time continuous problems modelling and prediction. This paper presents two new improved algorithms for Gaussian continuous and mixture of Gaussian continuous Hidden Markov Models cases for solving learning problem for large scale multidimensional data. The design of our parallel distributed algorithms is based on Spark, the Big Data framework, thereby we can distribute data over several nodes through Resilient Distributed Datasets which allow to apply, in parallel, a set of operations. The proposed algorithms have two main advantages: a high computational time efficiency and a high scalability as well as an easy integration in Big Data frameworks. The complexity comparison results show great improvements in computational complexity (by a factor of (states number)^2) and execution time. Moreover, the new algorithms might be more effective by reducing the communication costs between the elements of the system involved in the learning task.

Keywords: Big data, Machine learning, Continuous time hidden Markov models, Baum-Welch, Apache spark, Parallel distributed implementation.

1. Introduction

Currently, with the explosion of data volume generated and collected from different sources especially sensors, social networks, mobile devices and Internet, we live in an era marked by complex characteristics of data [1]. This huge amount of data has opened the door for improved modelling and prediction techniques [2]. Certainly, in the past, classical algorithms have shown their processing speed, efficiency and accuracy, but the digital revolution has changed everything. However, classical algorithms are, generally, less efficient in terms of complexity and execution time. Nowadays, in this era of Big Data, characterized by their huge volume, their high speed of production and diffusion as well as their varied nature, the design and implementation of Machine Learning algorithms has become a tedious task. It must therefore look for new algorithms adapted to Big Data or to review and improve conventional algorithms to adapt them to this new context in order to, effectively, manage and analyze Big Data. With this great panoply of Big Data technologies, we have to think about taking full advantage of the great benefits of these new technologies (i.e. distributed computing by GPU, Hadoop, Spark) with a set of powerful tools for managing and analyzing Big Data (e.g., data collection and data storage, preprocessing, feature selection and extraction) for data analysis and processing especially for large scale multidimensional data in order to reduce the computational cost of data analysis. Hidden Markov Models (HMMs) are widely used for modelling and predicting continuous problems [3]. These algorithms must be improved to give good results especially in a Big Data context [4].

In this work, we present two improved versions of Baum-Welch algorithm [5]. It is based on Spark framework [6], to solve problem of unsupervised learning for continuous-time Hidden Markov Models [7]. Thus, we propose two new algorithms for Gaussian continuous HMM and mixture of Gaussian continuous HMM cases.

Our proposed solution is based on Spark as main framework. These are parallel distributed versions of classical algorithms. To achieve this implementation, we considered a set of concepts under Spark: exploiting Resilient Distributed Datasets (RDDs) properties (i.e., data distribution over several nodes) and putting into practice the basic concepts of MapReduce paradigm (i.e., parallel computing operations) which allow to apply, in parallel, a set of operations (transformations and actions).

Our main contributions are summarized as follows:

• We introduce Hidden Markov Models fundamentals.

We discuss the three main questions of Hidden Markov Models.

• We review the Baum-Welch algorithm for the learning problem of Hidden Markov Models.

• We propose a parallel distributed Baum-Welch for continuous HMM: an improved version of Baum-Welch algorithm for solving unsupervised learning problem for continuous-time HMMs (Gaussian Continuous HMM).

• We propose a parallel distributed Baum-Welch for Continuous HMM with Gaussian mixtures: an improved version of Baum-Welch algorithm for solving unsupervised learning problem for mixture of continuous-time HMMs (Continuous HMM with Gaussian mixtures).

The rest of this paper is organized as follows. We provide an overview of some of the relevant literature review addressing this field of research in Section 2. In Section 3, we introduce notations used in this paper. Section 4 deals with Hidden Markov Models fundamentals. Next, we review Baum-Welch algorithm in Section 5. In Section 6, we present and describe our Parallel Distributed implementation of Baum-Welch algorithm for continuous-time HMM under Spark. We present a comparison of proposed algorithms with classical ones in Section 7. We conclude by a discussion of results and presentation of some conclusions and future directions in Section 8.

2. Related work

In the literature, several theories have been proposed to speed-up Baum-Welch algorithm especially those focusing on achieving parallel and/or distributed implementations of this algorithm.

Among the first important works that have studied this topic that of Mitchell et al. [8] who described a parallel implementation of a Hidden Markov Model (HMM) with Duration Modelling for spoken language recognition on the MasPar MP-1. This implementation exploits the massive parallelism of explicit duration HMMs to overcome many Implementational issues to develop complex models for real-time speech recognition. Another important work was presented by Turin [9] in which he proposed a parallel version of the Baum–Welch algorithm suitable for very large size observation requiring a large memory capacity which can reduces Baum-Welch training time. The proposed algorithm is based on temporal splitting of a training sequence, and it relies on some features of observation sequences, such as continuous repetitions of identical observations. In [10], the authors proposed a novel approach to ASL recognition based on parallel HMMs (PaHMMs) which model the parallel processes independently. Thus, it models the p processes with p independent HMMs with separate output. The recognition algorithm runs in time polynomial in the number of states, and in time linear in the number of parallel processes. The evaluation shows that the presented algorithm in this paper achieves a maximum recognition rate of 87.88% on the sentence level and 96.47% on the sign level.

Since the 2000s, efforts have multiplied and different approaches have emerged.

Anikeev et al. [11] proposed a simple strategy of organizing parallel HMM training, which can be effectively implemented using inexpensive network clusters. The proposed parallel algorithm was implemented for distributed cluster systems using Message-Passing Interface (MPI) standard which can be used in intrusion detection. The proposed parallel implementation of Baum-Welch algorithm for multiple observation sequences is suitable for training huge amount of data. Ma et al. [12] proposed a novel distributed multi-dimensional Hidden Markov Model (DHMM) for the modelling of multiple motion trajectories of objects and their interaction activities in a scene capable of conveying interactions information between multiple trajectories. It derives from this a novel General Forward-Backward (GFB) algorithm suitable for recursive calculation of model parameters. Simulation results show superior performance and higher accuracy of the proposed distributed 2D hidden Markov model. In [13], Liu Studied the parallelism of Baum-Welch algorithm for graphical processing units (GPU) and presented a prototype program for HMM training and classification on the NVIDIA CUDA platform. The proposed CUDA implementation achieves performance of 4.3 GFLOP/s and 200× speedups over CPU implementation. Li et al. [14] presented a general parallel learning framework, Cut-And-Stitch, for training hidden Markov chain models. They propose

a model-specific variant, CAS-HMM for learning hidden Markov models (HMM) which is implemented using OpenMP on two supercomputers and a quad-core commercial desktop. Another solution is described in [15] presenting a C++ library exploiting modern CPUs for constructing and analyzing general hidden Markov models which can be used for parallelizing Baum-Welch algorithm using OPENMP. The performance evaluation shows that the multi-threaded version of the Baum-Welch algorithm presents an impressive decrease in the running time. The speed-up can reach greater than a factor 1:5 for models with more than 400 states, when running it using two threads, and a speed-up close to a factor 3 for models with more than 600 states, when running it using 4 threads and a speed-up close to a factor 5 for models with more 800 states when running it with 8 threads. Hymel et al. [16] presented a parallel implementation under GPUs of Hidden Markov Models for wireless applications. They introduce a new method utilizing GPUs and HMMs to identify modulation schemes within a collected signal. The complexity of the algorithm is reduced from O (TN^2) or O (TMN) for the serial algorithm to O (T log N) for the parallel algorithm. The performances show a significant improvement in speedup which can reach 65x for the Baum-Welch algorithm with 4000 states.

In recent years, there has been another work of Yu et al. [17] who proposed a parallelized Hidden Markov Model to accelerate isolated words speech recognition. They implemented a GPU-accelerated HMM targeted for isolated-word based recognition. It is a new parallelization of continuous HMMs using two high-end GPUs belonging to Nvidia's Kepler architecture. The performance evaluation shows that this implementation can improve performance by 9.2x as compared to an optimized multi-thread CPU version during training stage.

In a recent paper, Bražėnas et al. [18] presented three different EM-based fitting procedures that can take advantage of the parallel hardware like Graphics Processing Units to reduce computational complexity for fitting Markov Arrival Processes with the expectation-maximization (EM) algorithm. The performance evaluation shows that the proposed algorithms are orders of magnitudes faster than the standard serial procedure.

There are a set of issues in previous implementations. The way of storing values of $\alpha_t(i)$, $\beta_t(i)$ and emissions probabilities. The memory allocation is not very efficient. In addition to the problem of data transfer since the time spent to transfer data between system modules and devices grows with the increase of state number of the HMM. There is another major drawback of proposed implementations which consists of their utilization which is application dependent and also depends on the architecture used (e.g., the parallel algorithms of Turin is designed for signal processing applications, Voglar's implementation is highly significant to gesture recognition research, the implementation strategy of Anikeev seems to be more suitable for inexpensive network clusters, rather than for massively parallel computers, the parallel implementation of Liu can only use single core CPUs or GPUs). In addition, the implementations of multiplication are not efficient implementations of matrix multiplication since matrix multiplication is not effectively optimized. For some implementations, the use of OpenMP causes configuration problems and target only shared memory system, it is not suitable for distributed memory systems. Concerning MPI, its use is not suitable for small grain level of parallelism, for example, to exploit the parallelism of multi-core platforms for shared memory multiprocessing [19]. However, the use of GPUs is not effective in treating HMMs with a small state number.

In this paper, we present a solution well adapted to Big Data but which also manages data of small size. It is a highly scalable implementation since we can add multiple nodes in a very simple way. One of the advantages of our implementations is that it manages heterogeneous data (i.e., structured, semi-structured and unstructured data) collected from several different sources even in real time.

Our implementation allows efficient memory management thanks to the use of RDD abstraction. RDDs are fault tolerant by nature. Thus, lost data can be recovered, often quite quickly, without requiring costly replication. It offers a distributed file system with failure and data replication management. During the induction phase for the calculation of $\alpha_t(i)$ and $\beta_t(i)$, the use of memory is optimized since we have opted for the use of vectors instead of matrices since vectors fit in memory on a single machine, while matrices do not [20].

It noticed that the use of a Big Data Framework provides a set of tools for data analysis and management that is easy to use, deploy and maintain. This implementation is not dependent on any particular framework or architecture. The proposed algorithms have a number of advantages compared to other solutions: a high computational time efficiency and a high scalability as well as an easy integration in Big Data frameworks which offer great capability of fast and scalable data processing allowing pre-processing and data cleaning with the powerful tools of Big Data frameworks.

Table 1. Notations of a hidden Markov model

Notation	Meaning
N	number of states in the model ($S = \{S_1, \ldots, S_N\}$)
S_i	$i^{th} state$
M	number of observation symbols
V	set of possible observations ($V = \{v_1, \ldots, v_M\}$)
O	observation sequence ($O = o_1, o_2, \ldots, o_T$)
T	length of observation sequence
π_i	initial state probability
Π	initial state matrix ($\Pi = \{\pi_i\}$)
a_{ij}	transition probability
A	Transition matrix ($A = \{a_{ij}\}$)
$b_j(v_k)$	Observation probability
B	Observation matrix ($B = \{b_j(v_k)\}$)
λ	model parameters, $\lambda = \{A, B, \Pi\}$
o_t	observation in time t
$\alpha_t(i)$	forward variable
$\beta_t(i)$	backward variable
$\gamma_t(i)$	probability of being at state S_i at time t, given λ and O
$\xi_t(i,j)$	probability of being at state S_i at time t, and at state S_j at time $t+1$, given λ and O
μ_j	mean
Σ_j	covariance matrix
μ_{jm}	mean of m^{th} mixture in state S_j
Σ_{jm}	covariance matrix of m^{th} mixture in state S_j
c_{jm}	m^{th} mixture weights in state S_j

3. Notations

In the following table (Table 1), we present notations used in this paper.

4. Hidden Markov Models

In this section, we review theoretical foundations of Hidden Markov Models and discuss the three fundamental problems of HMMs. Consider a discrete time Markov chain with a finite set of states $S = \{S_1, S_2, \ldots, S_N\}$. An HMM is defined by the following compact notation to indicate the complete parameter set of the model $\lambda = (\Pi, \mathrm{A}, \mathrm{B})$ where Π, A and B are the initial state distribution vector, matrix of state transition probabilities and the set of the observation probability distribution in each state, respectively [3,7,21]:

$$\Pi = [\pi_1, \pi_2, \ldots, \pi_N],\ \pi_i = \Pr\{q_1 = S_i\}, \quad (1)$$

$$A = \{a_{ij}\}, a_{ij} = Pr\{q_{t+1} = S_j \mid q_t = S_i\}, \quad (2)$$

for $1 \le i, j \le N,\ S_i, S_j \in S,\ t \in [1,2,\ldots,T]$

The observation at time t, o_t, may be a discrete symbol (Discrete HMMs (DHMMs [22]) case, $o_t = v_k$, $v_k \in V = v_1, v_2, \ldots, v_M$, or continuous, $o_t \in \mathbb{R}^k$. The observation matrix B is defined by $B = \{b_j(o_t)\}$, where $b_j(o_t)$ is the state conditional probability of the observation o_t defined by:

$$b_j(o_t) = \Pr\{\, o_t = v_k \mid q_t = S_j \,\}, \quad (3)$$

for $1 \le j \le N, 1 \le k \le M$

For a continuous observation (Continuous HMMs (CHMMs) case [7]), $b_j(o_t)$ is defined by a finite mixture of any log-concave or elliptically symmetric probability density function (pdf), e.g. Gaussian pdf, with state conditional observation mean vector μ_j and state conditional observation covariance matrix Σ_j, so B may be defined as:

$$B = \{\mu_j, \Sigma_j\}, i = 1,2,\ldots,N \quad (4)$$

The model parameters constraints for $1 \le i, j \le N$ are

$$\sum_{i=1}^{N} \pi_i = 1, \sum_{j=1}^{N} a_{ij} = 1, a_{ij} \ge 0, \quad (5)$$

$$\sum_{k=1}^{M} b_j(o_t = v_k) = 1 \ or \int_{-\infty}^{+\infty} b_j(o_t) \mathrm{d}o_t = 1 \quad (6)$$

In general, at each instant of time t, the model is in one of the states S_i, $1 \le i \le N$. It outputs o_t according to a discrete probability (in the DHMM case) or according to a continuous density function (in the CHMM case) $b_j(o_t)$ and then jumps to state S_j, $1 \le j \le N$ with probability a_{ij}. The state transition matrix defines the structure of the HMM.

There are three main questions we are interested in about HMM. First, the evaluation problem in which we look for the probability $Pr\{O|\lambda\}$ that the given observations $O = o_1, o_2, \ldots, o_T$ are generated by the model λ with a given HMM. Second, the decoding problem in which we look for the most likely state sequence in the given model λ that produced the given observations $O = o_1, o_2, \ldots, o_T$. Third, the learning problem in which we look for how we can adjust the model parameters $\{A, B, \Pi\}$ in order to maximize $Pr\{O|\lambda\}$ given a model λ and a sequence of observations $O = o_1, o_2, \ldots, o_T$.

5. Baum-Welch algorithm

The most important problem about HMMs is the learning problem or parameter estimation. To resolve this problem, Baum-Welch algorithm, known as Forward-Backward algorithm, is the most used. It is a special case of the Expectation-Maximisation (i.e., EM) algorithm [23]. In an HMM, the observations can be discrete or continuous. In this paper, we are interested in continuous-time HMM case. We treat Gaussian continuous observation and mixtures of Gaussian continuous observation cases.

The Baum-Welch algorithm for Gaussian continuous observation takes as input an initial model ($\lambda = (A, \mu_j, \Sigma_j, \Pi)$) and a sequence of observations ($O = o_1, o_2, \ldots, o_T$) and estimates the transition matrix A and the observation matrix B in function of mean (μ_j) and covariance matrix (Σ_j) that maximize the probability for the given observations. The iterations terminate when a convergence criterion is meet.

The Baum-Welch algorithm for Gaussian continuous observation can be represented as follows:

$$\alpha_1(j) = \pi_j b_j(o_1),\ 1 \le j \le N \tag{7}$$

$$\alpha_{t+1}(j) = b_j(o_{t+1}) \sum_{i=1}^{N} \alpha_t(i) a_{ij}, \quad 1 \le j \le N, 1 \le t \le T-1 \tag{8}$$

$$Pr\{O|\lambda\} = \sum_{i=1}^{N} \alpha_T(i) \tag{9}$$

$$\beta_T(j) = 1, 1 \le j \le N \tag{10}$$

$$\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j) a_{ij} b_j(o_{t+1}), \quad 1 \le i \le N, 1 \le t \le T-1 \tag{11}$$

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\Pr\{O|\lambda\}},\quad 1 \le i \le N, 1 \le t \le T \tag{12}$$

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} \beta_{t+1}(j) b_j(o_{t+1})}{\Pr\{O|\lambda\}}, \quad 1 \le i, j \le N, 1 \le t \le T-1 \tag{13}$$

$$\overline{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}, 1 \le i, j \le N \tag{14}$$

$$\overline{\mu}_j = \frac{\sum_{t=1}^{T} \gamma_t(j) o_t}{\sum_{t=1}^{T} \gamma_t(j)}, 1 \le j \le N, 1 \le t \le T \tag{15}$$

$$\overline{\Sigma}_j = \frac{\sum_{t=1}^{T} \gamma_t(j)(o_t - \mu_j)(o_t - \mu_j)^T}{\sum_{t=1}^{T} \gamma_t(j)}, \quad 1 \le j \le N, 1 \le t \le T \tag{16}$$

$$\overline{\pi}_i = \alpha_1(i)\beta_1(i), 1 \le i \le N \tag{17}$$

The Baum-Welch algorithm with mixtures of Gaussian continuous observation takes as input an initial model ($\lambda = (A, c_{jm}, \mu_{jm}, \Sigma_{jm}, \Pi)$) and a sequence of observations ($O = o_1, o_2, \ldots, o_T$) and estimates the transition matrix A and the observation matrix B in function of mean of m^{th} mixture μ_{jm}, covariance matrix of m^{th} mixture Σ_{jm} and m^{th} mixture weights c_{jm}. The iterations terminate when a convergence criterion is meet. Thus, the Baum-Welch algorithm with mixtures of Gaussian continuous observation is as follows:

$$\alpha_1(j) = \pi_j b_j(o_1), 1 \le j \le N \tag{18}$$

$$\alpha_{t+1}(j) = b_j(o_{t+1}) \sum_{i=1}^{N} \alpha_t(i) a_{ij}, \quad 1 \le j \le N, 1 \le t \le T-1 \tag{19}$$

$$Pr\{O|\lambda\} = \sum_{i=1}^{N} \alpha_T(i) \tag{20}$$

$$\beta_T(j) = 1, 1 \le j \le N \tag{21}$$

$$\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j) a_{ij} b_j(o_{t+1}) \quad 1 \le i \le N, 1 \le t \le T-1 \tag{22}$$

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\Pr\{O|\lambda\}}, 1 \le i \le N, 1 \le t \le T \tag{23}$$

$$\xi_t(j,m) \leftarrow \frac{\alpha_t(i) a_{ij} c_{jm} g_{jm}(o_t) \beta_{t+1}(j)}{\Pr\{O|\lambda\}}, \quad 1 \le i, j \le N, 1 \le t \le T-1 \tag{24}$$

$$\overline{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}, 1 \le i \le N, 1 \le j \le N \tag{25}$$

$$\overline{c}_{jm} = \frac{\sum_{t=1}^{T} \xi_t(j,m) o_t}{\sum_{t=1}^{T} \gamma_t(j)}, \quad 1 \le j \le N, 1 \le t \le T \tag{26}$$

$$\overline{\mu}_{jm} = \frac{\sum_{t=1}^{T} \xi_t(j,m) o_t}{\sum_{t=1}^{T} \gamma_t(j)}, \quad 1 \le j \le N,\ 1 \le t \le T \tag{27}$$

$$\overline{\Sigma}_{jm} = \frac{\sum_{t=1}^{T} \xi_t(j,m)(o_t - \mu_{jm})(o_t - \mu_{jm})^T}{\sum_{t=1}^{T} \xi_t(j,m)}, \quad 1 \le j \le N, 1 \le t \le T \tag{28}$$

$$\overline{\pi}_i = \alpha_1(i)\beta_1(i), 1 \le i \le N \tag{29}$$

6. Parallel distributed implementation of Baum-Welch algorithm for continuous time HMM on Spark

The design of our algorithms is based on Spark as

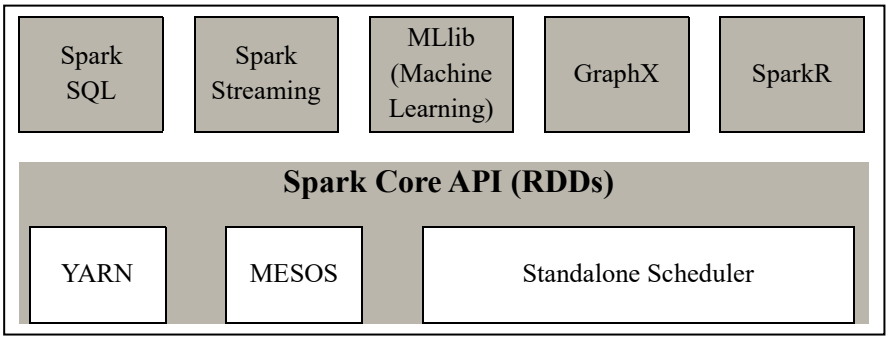

Figure.1 Spark architecture

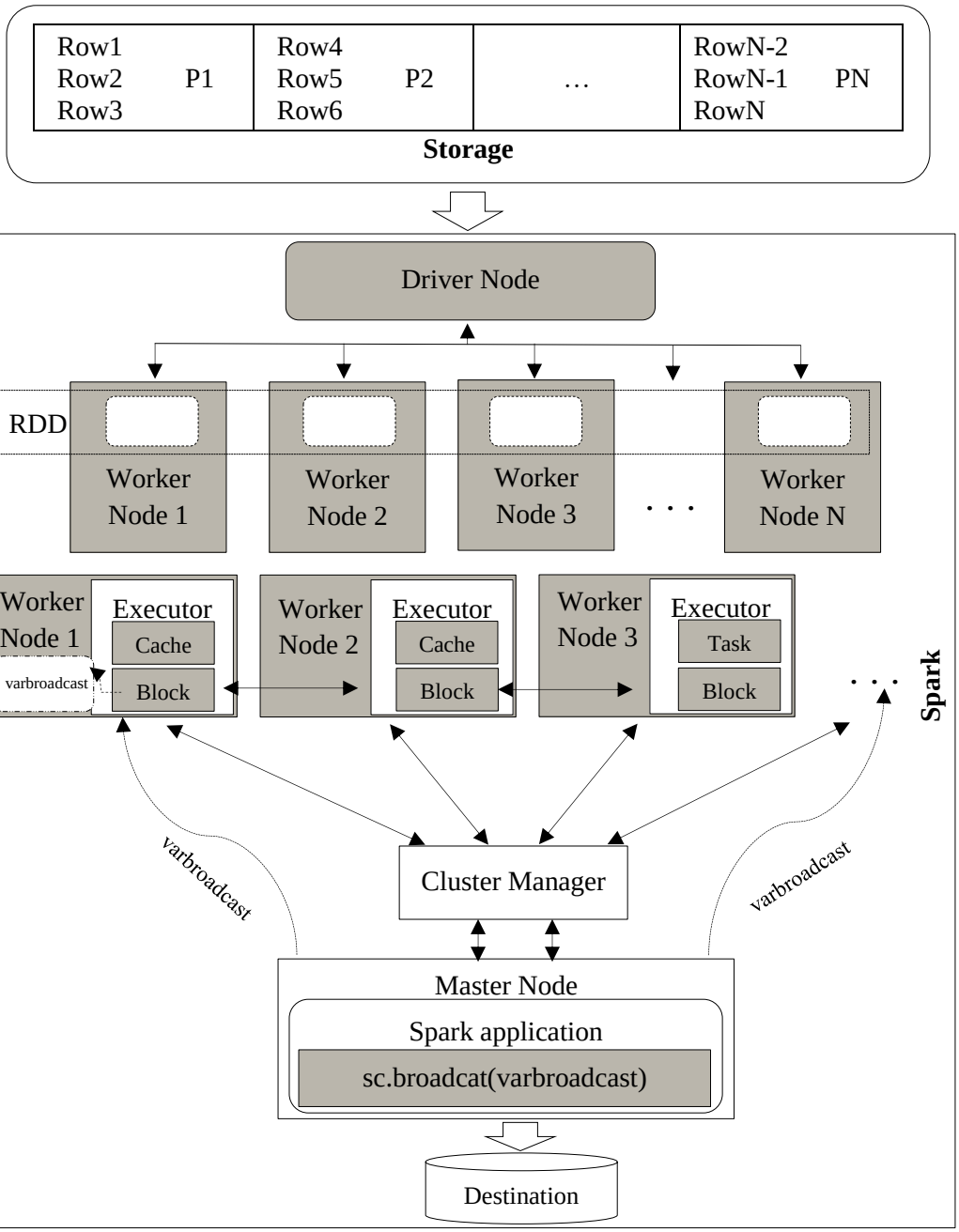

Figure.2 Main Spark's concepts used in the proposed implementation

main framework. Apache Spark is an open source Big Data processing framework that allows to run large-scale analytics applications in batch and real time processing modes in a distributed manner (cluster computing). Spark supports In-memory processing, boosting the performance of Big Data analytics applications. However, it also allows conventional disk processing when data sets are too large for available system memory.

The Spark ecosystem has several tools (Fig. 1): Spark cluster manager (includes Apache Mesos [24], Apache Yarn [25] and built-in Standalone cluster manger), Spark for batch processing, Spark Streaming [26] for the continuous processing of data streams, MLlib [27] for Machine Learning, GraphX [28] for graph calculations, Spark SQL [29] which is an SQL-like implementation of data query. Moreover, it integrates perfectly with the Hadoop ecosystem [30] (including HDFS [31]).

A Spark application contains several components whether in using Spark on a single machine or on a cluster of hundreds or thousands of nodes. A Spark application consists of a single Driver (responsible for distributing the tasks on the various executors. It is the driver that executes the method of applications), the Master, the Cluster Manager (responsible for instantiating the different workers), and a set of Executors processes scattered across nodes on the cluster, which run on worker nodes, or Workers (each worker instantiates an executor responsible for executing the various calculation tasks).

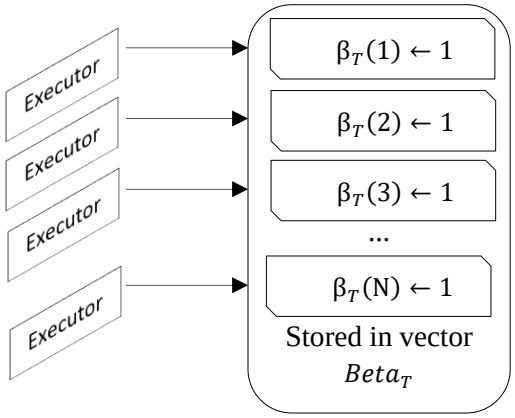

Figure.3 Initialization step in parallel of backward variable

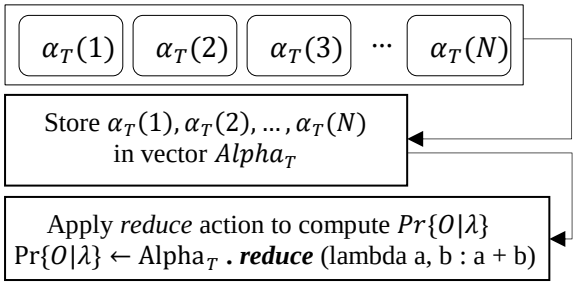

Figure.4 Computation of the probability $Pr\{O \mid \lambda\}$

For example, the initialization of the backward variable will be performed using N executors in parallel and then storing the $\beta_T(j)$ values in the $Beta_T$ vector as shown in Fig. 3.

To compute the probability $Pr\{O \mid \lambda\}$, we apply the Spark's action, *reduce* on all the elements of the vector $Alpha_T$ without using an iteration on N which makes it possible to reduce the computational complexity of O(N) to O(1).

To achieve this implementation, we exploited three key concepts: Spark's Resilient Distributed Datasets (RDDs) [32], for distributing data over many blocks, MapReduce paradigm [33] to achieve the parallel computation and broadcast variables to reduce communication cost (Fig. 2).

The main technical innovation offered by Apache Spark is the concept of Resilient Distributed Datasets (RDDs). The RDD are an abstraction of programming. They represent an immutable collection of objects that can be distributed on a cluster. They are fault-tolerant and provide parallel data structures that allow users to explicitly store intermediate data in memory, control their partitioning to optimize data storage and manipulate data using a set of operators. Operations on RDDs can be distributed across the cluster and executed in a parallel batch process, allowing for fast, scalable parallel processing. RDDs support two types of operations: transformations (e.g., map, filter) and actions (e.g., reduce, collect). As Hadoop, Spark relies on a distributed storage system (e.g., HDFS) to store the input and output data of the jobs submitted by users. However, unlike Hadoop, Spark allows RDDs to be cached in the memory and therefore intermediate data between different iterations of a job can be reused efficiently. This reduces the number of costly disk Input/Output accesses to the distributed storage system. This memory-resident feature of Spark is particularly essential for some Big Data applications such as iterative Machine Learning algorithms which intensively reuse the results across multiple iterations of a MapReduce job.

MapReduce is a programming paradigm that enables parallel distributed processing of large sets of data, converting them into another set of data (i.e., map function), and then combining and reducing those output sets of data into smaller sets of data (i.e., reduce function). MapReduce was designed to take big data and use parallel distributed computing to turn big data into little- or regular-sized data. The MapReduce paradigm allows to apply RDDs

transformations which include several MapReduce-like operations (e.g., map, reduce, collect).

Running a Spark operation on a remote cluster node uses several functions. This operation is usually done in such a way that a different copy of variables is used in the functions. These particular variables are copied to different machines and updates to these variables are not propagated to the driver program. So, the use of read-write shared variables in tasks is inefficient. Nevertheless, Spark provides two types of shared variables: broadcast variables and accumulators. Broadcast Variables are an another very useful concept in this implementation whose objective is to reduce the communications cost. They allow to keep a read-only secure variable cached on different nodes, instead of sending only one copy with each of the necessary tasks. A calculation Spark operation first begins with the variable broadcast send to each node concerned by the associated task. Then each node caches it locally in a serialized form. Hence, to run a scheduled task, instead of getting values from the Driver, these are extracted locally from the cache. So, broadcasting does not really mean that a given object is not transmitted at all on the network. But unlike normal variables, broadcast variables are always read-only and can only be sent once.

In what follows, we present the improved algorithms (i.e., parallel distributed Baum-Welch for continuous time HMMs).

Given a sequence of observations and an initial model λ (A, μ_j, Σ_j, Π), our proposed improved algorithm (Parallel Distributed Baum-Welch Algorithm) for solving unsupervised continuous-time HMMs learning problem is presented in Algorithm 1.

Baum-Welch algorithm for mixture of continuous-time HMMs is widely used in several application domains such as artificial intelligence, pattern recognition, speech recognition, signal processing, biological sequence analysis, robotics and finance. Given a sequence of observations and an initial model λ (A, c_{jm}, μ_{jm}, Σ_{jm}, Π), the proposed version of Baum-Welch algorithm (Parallel Distributed Baum-Welch for Mixture Continuous HMM) for solving unsupervised continuous time HMMs with Gaussian mixtures learning problem is presented in Algorithm 2.

7. Comparisons results

In this section, we investigate the complexities of the proposed algorithms and compare them, step by step, with the existing algorithms. The factor *cst* represents the communication cost between the elements of the system. For the time complexity, the comparison was made by determining the number of significant operations that the algorithm does (e.g., assignment, iterations, sum). Then the space complexity is measured by calculating the required memory consumption of the algorithm (e.g., number

Table 2. Time complexity comparison

	Classical Baum-Welch	**Parallel distributed Baum-Welch**
Forward variable initialization $\alpha_1(j)$	$O(N)$	***cst***.$O(1)$
calculation of $\alpha_{t+1}(i)$	$O(N^2(T-1))$	***cst***.$O(T-1)$
calculation of $Pr\{O \mid \lambda\}$	$O(N)$	***cst***.$O(1)$
Backward variable initialization $\beta_T(j)$	$O(N)$	***cst***.$O(1)$
calculation of $\beta_t(j)$	$O(N^2(T-1))$	***cst***.$O(T-1)$
calculation of $\gamma_t(i)$	$O(NT)$	***cst***.$O(T)$
calculation of $\xi_t(i,j)$	$O(N^2(T-1))$	***cst***.$O(T-1)$
calculation of $\overline{a_{ij}}$	$O(N^2)$	***cst***.$O(1)$
calculation of $\overline{\mu_j}$	$O(N)$	***cst***.$O(1)$
calculation of $\overline{\mu_{jm}}$	$O(NM)$	***cst***.$O(1)$
calculation of $\overline{\Sigma_j}$	$O(N)$	***cst***.$O(1)$
calculation of $\overline{\Sigma_{jm}}$	$O(NM)$	***cst***.$O(1)$
calculation of $\overline{c_{jm}}$	$O(NM)$	***cst***.$O(1)$

Table 3. Space complexity comparison

	Classical Baum-Welch	**Parallel distributed Baum-Welch**
Forward variable calculation	$O(NT)$	$O(NT)$
Backward variable calculation	$O(NT)$	$O(NT)$
calculation of $\gamma_t(i)$	$O(NT)$	$O(NT)$
calculation of $\xi_t(i,j)$	$O(N^2(T-1))$	$O(N^2(T-1))$
calculation of $\overline{a_{ij}}$	$O(N^2)$	$O(N^2)$
calculation of $\overline{\mu_j}$	$O(N)$	$O(N)$
calculation of $\overline{\mu_{jm}}$	$O(NM)$	$O(1)$
calculation of $\overline{\Sigma_j}$	$O(N)$	$O(N)$
calculation of $\overline{\Sigma_{jm}}$	$O(NM)$	$O(NM)$
calculation of $\overline{c_{jm}}$	$O(NM)$	$O(NM)$
calculation of $\overline{\pi_i}$	$O(N)$	$O(N)$

Algorithm 1: Parallel distributed Baum-Welch Algorithm under Spark (Gaussian Continuous HMM)

Input: Initial model $\lambda = (A,B,\Pi)$, a sequence of observations $O = o_1, o_2, \ldots, o_T$

Output: Optimal Model parameters: $\overline{\boldsymbol{A}} = \{\boldsymbol{a_{ij}}\}, \overline{\boldsymbol{\Pi}} = \{\boldsymbol{\pi_i}\}, \overline{\boldsymbol{B}} =$ {mean $\overline{\boldsymbol{\mu_j}}$ and variance $\overline{\sum_{\boldsymbol{j}}}$}

1: **for** each executor$_j$ of *N* executors **do**
2: **Parallel do**
3: $\alpha_1(j) \leftarrow \pi_j \; b_j(o_1)$ { $j \in [1, 2, 3, \ldots, N]$}
4: **end for**
5: **for** t ← 1 **to** *T-1* **do**
6: **for** each executor$_{i,j}$ of *N*N* executors **do**
7: **Parallel do**
8: *calculate* $(map)\ \alpha_t(i)a_{ij}$ and store $\alpha_t(i)$ in Alpha$_t$ { $i, j \in [1, 2, 3, \ldots, N]$}
9: **end for**
10: **end for**
11: Pr$\{O|\lambda\}$ ← Alpha$_T$ **.** ***reduce*** (lambda a, b : a + b)
12: **for** each executor$_j$ of *N* executors **do**
13: **Parallel do**
14: $\beta_T(j) \leftarrow 1$ { $j \in [1, 2, 3, \ldots, N]$}
15: **end for**
16: **for** t ← *T-1* **downto** *1* **do**
17: **for** each executor$_{i,j}$ of *N*N* executors **do**
18: **Parallel do**
19: *calculate* $\beta_{t+1}(j)a_{ij}b_j(o_{t+1})$ *and store* $\beta_t(j)$ in Beta$_t$ { $i, j \in [1, 2, 3, \ldots, N]$}
20: **end for**
21: **end for**
22: **for** each executor$_{t,i}$ of *T*N* executors **do**
23: **Parallel do**
24: *calculate* $\gamma_t(i) \leftarrow (\alpha_t(i)\beta_t(i))/\Pr\{O|\lambda\}$ *and store* $\gamma_t(i)$ in Gamma$_t$
25: **end for**
26: **for** each executor$_{t,i,j}$ of *(T-1)*N*N* executors **do**
27: **Parallel do**
28: *calculate* $\xi_t(i,j) = (\alpha_t(i)a_{ij}\beta_{t+1}(j)b_j(o_{t+1}))/\Pr\{O|\lambda\}$ *and store* $\xi_t(i,j)$ in
29: **end for**
33: **for** each executor$_{i,j}$ of *N*N* executors **do**
31: **Parallel do**
32: $\overline{\boldsymbol{a_{ij}}} \leftarrow \boldsymbol{sum}(\xi_t(i,j))/\boldsymbol{sum}\ (\gamma_t(i))$ { $i, j \in [1, 2, 3, \ldots, N]$; $t \in [1, 2, 3, \ldots, T-1]$}
33: **end for**
34: **for** each executor$_i$ of *N* executors **do**
35: **Parallel do**
36: $\overline{\boldsymbol{\pi_i}} \leftarrow \gamma_1(i)$ { $i \in [1, 2, 3, \ldots, N]$}
37: **end for**
38: **for** each executor$_j$ of *N* executors **do**
39: **Parallel do**
40: $\overline{\boldsymbol{\mu_j}} \leftarrow \boldsymbol{sum}(\gamma_t(j)o_t)/\boldsymbol{sum}\ (\gamma_t(j))$ { $j \in [1, 2, 3, \ldots, N]$; $t \in [1, 2, 3, \ldots, T]$}
41: **end for**
42: **for** each executor$_j$ of *N* executors **do**
43: **Parallel do**
44: $\overline{\sum_{\boldsymbol{j}}} = \boldsymbol{sum}(\gamma_t(j)(o_{t-}\mu_j)(o_{t-}\mu_j)^T)/\boldsymbol{sum}\ (\gamma_t(j))$ { $j \in [1, 2, 3, \ldots, N]$; $t \in [1, 2, 3, \ldots, T]$}
45: **end for**
46: **set** $\lambda \leftarrow \bar{\lambda}$ and Go to **22** unless some *convergence criterion* is met
47: **return** $\overline{\boldsymbol{A}}, \overline{\boldsymbol{\Pi}}, \overline{\boldsymbol{B}}\{\overline{\boldsymbol{\mu_j}}, \overline{\sum_{\boldsymbol{j}}}\}$

Algorithm 2: Parallel distributed Baum-Welch Algorithm under Spark (Continuous HMM with Gaussian mixtures)

Input: Initial model $\lambda = (A,B,\Pi)$, a sequence of observations $O = o_1, o_2, \ldots, o_T$

Output: Optimal Model parameters: $\overline{\boldsymbol{A}} = \{\boldsymbol{a_{ij}}\}, \overline{\boldsymbol{\Pi}} = \{\boldsymbol{\pi_i}\}, \overline{\boldsymbol{B}}\{$ mean of m^{th} mixture $\overline{\boldsymbol{\mu_{jm}}}$, covariance of m^{th} mixture $\overline{\sum_{\boldsymbol{jm}}}$ and $\overline{\boldsymbol{c_{jm}}}$ m^{th} mixture weights$\}$

for each executor$_j$ of *N* executors **do**
Parallel do
  $\alpha_1(j) \leftarrow \pi_j \; b_j(o_1)$ $\{ j \in [1, 2, 3, \ldots, N]\}$
end for
for t ← 1 **to** *T-1* **do**
for each executor$_{i,j}$ of *N*N* executors **do**
Parallel do
  $calculate\ (map)\ \alpha_t(i)a_{ij}$ and store $\alpha_t(i)$ in Alpha_t $\{ i, j \in [1, 2, 3, \ldots, N]\}$
  $sum\ (reduce)$ of $\alpha_t(i)a_{ij}$, then multiple by $b_j(o_{t+1})$ $\{i, j \in [1, 2, 3, \ldots, N]\}$
end for
end for
$\Pr\{O|\lambda\} \leftarrow \text{Alpha}_T$ **.** ***reduce*** (lambda a, b : a + b)
for each executor$_j$ of *N* executors **do**
Parallel do
  $\beta_T(j) \leftarrow 1$ $\{ j \in [1, 2, 3, \ldots, N]\}$
end for
for t ← *T-1* **downto** *1* **do**
  for each executor$_{i,j}$ of *N*N* executors **do**
    Parallel do
   $calculate\ \beta_{t+1}(j)a_{ij}b_j(o_{t+1})\ and\ store\ \beta_t(j)$ in Beta_t $\{ i, j \in [1, 2, 3, \ldots, N]\}$
  end for
end for
for each executor$_{t,i}$ of *T*N* executors **do**
Parallel do
  $calculate\ \gamma_t(i)$
$\leftarrow \left(\alpha_t(i)\beta_t(i)\right) / \Pr\{O|\lambda\}\ and\ store\ \gamma_t(i)$ in Gamma_t $\{ i \in [1, 2, 3, \ldots, N]\ ; t \in [1, 2, 3, \ldots, T]\}$
end for
for each executor$_{t,i,j}$ of *(T-1)*N*N* executors **do**
Parallel do
  $calculate\ \gamma_t(i,j) \leftarrow (\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)) / \Pr\{O|\lambda\}\ and\ store\ \gamma_t(i,j)$ in
  $Gamma2_t$ $\{ i, j \in [1, 2, 3, \ldots, N];\ t \in [1, 2, 3, \ldots, T-1]\}$
end for
for each executor$_{t,j,m}$ of *(T-1)*N*M* executors **do**
Parallel do
  $calculate\ \xi_t(j,m) \leftarrow \boldsymbol{sum}\left(\alpha_t(i)a_{ij}c_{jm}g_{jm}(o_t)\beta_{t+1}(j)\right) / \Pr\{O|\lambda\}\ store\ \xi_t(j,m)$ in Xi_t
  $\{ i, j \in [1, 2, 3, \ldots, N];\ t \in [1, 2, 3, \ldots, T-1]\}$
end for
for each executor$_{i,j}$ of *N*N* executors **do**
Parallel do
  $\overline{\boldsymbol{a_{ij}}} \leftarrow \boldsymbol{sum}(\gamma_t(i,j)) / \boldsymbol{sum}\ (\gamma_t(i))$ $\{ i, j \in [1, 2, 3, \ldots, N];\ t \in [1, 2, 3, \ldots, T-1]\}$
end for
for each executor$_{j,m}$ of *N*M* executors **do**
Parallel do
  $\overline{\boldsymbol{c_{jm}}} \leftarrow \boldsymbol{sum}(\xi_t(j,m)) / \boldsymbol{sum}\ (\gamma_t(i))\{ i, j \in [1, 2, 3, \ldots, N];\ t \in [1, 2, 3, \ldots, T-1]\}$

41: **end for**
42: **for** each executor$_{j,m}$ of *N*M* executors **do**
43: **Parallel do**
44: $\overline{\boldsymbol{\mu_{jm}}} \leftarrow \boldsymbol{sum}(\xi_t(j,m)o_t)/\boldsymbol{sum}\,(\xi_t(j,m))\ \{\, i,j \in [1,2,3,\dots,\mathrm{N}]; t \in [1,2,3,\dots,\mathrm{T}-1]\}$
45: **end for**
46: **for** each executor$_{j,m}$ of *N*M* executors **do**
47: **Parallel do**
48: $\overline{\sum_{\boldsymbol{jm}}} \leftarrow \boldsymbol{sum}(\xi_t(j,m)(o_{t-}\overline{\boldsymbol{\mu_{jm}}})(o_{t-}\overline{\boldsymbol{\mu_{jm}}})^{\boldsymbol{T}})/\boldsymbol{sum}\left(\xi_t(j,m)\right)\ \{\, j \in [1,2,3,\dots,\mathrm{N}];\ t \in [1,2,3,\dots,\mathrm{T}-1]\}$
49: **end for**
50: **set** $\lambda \leftarrow \bar{\lambda}$ and Go to **22** unless some *convergence criterion* is met
51: **return** $\overline{\boldsymbol{A}}, \overline{\boldsymbol{\Pi}}, \overline{\boldsymbol{B}}\{\overline{\boldsymbol{c_{jm}}}, \overline{\boldsymbol{\mu_{jm}}}, \overline{\boldsymbol{\sum_{jm}}}\}$

of variables, size of vectors).

In all steps of the algorithms, our parallel distributed Baum-Welch algorithm under Spark show excellent performance especially in time complexity as shown in Table 2. While, in terms of spatial complexity, there is no difference (See Table 3).

8. Discussion and conclusion

We have presented parallel distributed versions of Baum-Welch algorithm for Gaussian continuous HMM and mixture of Gaussian continuous HMM. Our proposed solution is based on Spark as main framework. It is an improvement of Baum-Welch algorithm for continuous-time Hidden Markov Models that allows Big Data processing. To achieve this implementation, we have exploited the enormous advantages of this framework and have considered a set of concepts under Spark: exploiting Resilient Distributed Datasets (RDDs) properties (i.e., data distribution over several nodes), putting into practice the basic concepts of MapReduce paradigm (i.e., parallel computing operations) which allow to apply, in parallel, a set of operations (transformations and actions). Spark integrates a set of tools for Streaming, SQL, Machine Learning and Graphs in addition to powerful preprocessing tools (e.g., features extraction, transformation and selection, dimensionality reduction) thanks to MLlib the Spark's Machine Learning library.

Through this implementation, we proposed an efficient and fast solution (computational cost reduced by a factor of N^2): the complexity of the initialization step of both forward and backward variables reduced from $O(N)$ to $O(1)$ and the computation complexity of the variables $\alpha_t(i)$, $\beta_t(i)$, $\gamma_t(i)$ and $\xi_t(i,j)$ is reduced by $O(N^2(T-1))$ at $O(T-1)$.

Comparisons with classical algorithms show a great improvement in computational complexity and execution time. The improved algorithms can produce results faster than previous versions. Thus, by following the proposed algorithms, we managed to achieve our objectives: optimize complexity, reduce execution time and provide a solution to unsupervised learning problem for continuous-time HMM especially when dealing with Big Data (e.g., large number of states, large number of observations).

In sum, the proposed algorithms have several advantages compared to other solutions: a high computational time efficiency and a high scalability as well as an easy integration in Big Data frameworks which offer great capability of fast and scalable data processing allowing pre-processing and data cleaning with the powerful tools of Big Data frameworks.

Further work will focus on how to reduce latency caused by the use of RDDs that are stored in a stack of instructions and are not available or transformed until an action is executed (i.e., Lazy Evaluation). The approach can be perfected in the next steps by using novel learning techniques. Other metrics can also be used to allow a good evaluation of our algorithms.

References

[1] I. Lee, "Big data: Dimensions, evolution, impacts, and challenges", *Business Horizons*, Vol. 60, No. 3, pp.293-303, 2017.
[2] A. L'heureux, K. Grolinger, H. F. Elyamany, and M. A. Capretz, "Machine learning with big data: Challenges and approaches", *IEEE Access*, Vol. 5, pp.7776-7797, 2017.
[3] L.R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition", In: *Proc. of the IEEE*, Vol. 77, No. 2, pp.257-286, 1989.
[4] K. Kambatla, G. Kollias, V. Kumar, and A. Grama, "Trends in big data analytics", *Journal of Parallel and Distributed Computing*, Vol. 74, No. 7, pp.2561-2573, 2014.

[5] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains", *The annals of mathematical statistics*, Vol. 41, No. 1, pp.164-171, 1970.

[6] M. Zaharia, M. Chowdhury, M. J., Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets", *HotCloud*, Vol. 10, No. 10-10, pp.95, 2010.

[7] B. H. Juang and L. Rabiner, "Mixture autoregressive hidden Markov models for speech signals", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 33, No. 6, pp.1404-1413, 1985.

[8] C. D. Mitchell, M. P. Harper, L. H. Jamieson, and R. A. Helzerman, "A parallel implementation of a hidden Markov model with duration modeling for speech recognition", *Digital Signal Processing*, Vol. 5, No. 1, pp.43-57, 1995.

[9] W. Turin, "Unidirectional and parallel Baum-Welch algorithms", *IEEE Transactions on Speech and Audio Processing*, Vol. 6, No. 6, pp.516-523, 1998.

[10] C. Vogler and D. Metaxas, "Parallel hidden markov models for american sign language recognition", In: *Proc. of the Seventh IEEE International Conf. on Computer Vision*, pp.116-122, 1999.

[11] M. Anikeev and O. Makarevich, "Parallel implementation of Baum–Welch algorithm", In: *Proc. of the International Conf. on Computer Science and Information Technologies*, pp.197-200, 2006.

[12] X. Ma, D. Schonfeld, and A. Khokhar, "Distributed multi-dimensional hidden Markov model: theory and application in multiple-object trajectory classification and recognition", In: *Proc. of the Conf. of Multimedia Content Access: Algorithms and Systems II. International Society for Optics and Photonics*, pp.68200O, 2008.

[13] C. H. Liu, "cuHMM: a CUDA implementation of hidden Markov model training and classification", *The Chronicle of Higher Education*, pp.1-13, 2009.

[14] L. Li, B. Fu, and C. Faloutsos, "Efficient Parallel Learning of Hidden Markov Chain Models on SMPs", *IEICE Transactions on Information and Systems*, Vol. 93, No. 6, pp.1330-1342, 2010.

[15] A. Sand, C. N. Pedersen, T. Mailund, and A. T. Brask, "HMMlib: A C++ library for general hidden Markov models exploiting modern CPUs", In: *Proc. of the Ninth International Conf. on Parallel and Distributed Methods in Verification, and Second International Conf. on High Performance Computational Systems Biology*, pp.126-134, 2010.

[16] S. Hymel, I. Akbar, and J. F. Reed, "Parallel implementation of Hidden Markov Models for wireless applications", In: *Proc. of the SDR 11 Technical Conf. and Product Exposition*, pp.94-101, 2011.

[17] L. Yu, Y. Ukidave, and D. Kaeli, "GPU-Accelerated HMM for speech recognition", In: *Proc. of the 43rd International Conf. on Parallel Processing Workshops*, pp.395-402, 2014.

[18] M. Bražėnas, G. Horváth, and M. Telek, "Parallel algorithms for fitting Markov arrival processes", *Performance Evaluation*, Vol. 123, pp.50-67, 2018.

[19] J. L Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf", *Procedia Computer Science*, Vol. 53, pp.121-130, 2015.

[20] Z. R. Bosagh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, S., E. Sparks, A. Staple, and M. Zaharia, "Matrix computations and optimization in apache spark", In: *Proc. of the 22nd ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pp.31-38, 2016.

[21] L. R. Rabiner and B. H. Juang, "An introduction to hidden Markov models", *IEEE ASSP Magazine*, Vol. 3, No. 1, pp.4-16, 1986.

[22] I. L. Macdonald and W. Zucchini, *Hidden Markov and other models for discrete-valued time series*, Vol. 110, CRC Press, 1997.

[23] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm", *Journal of the Royal Statistical Society: Series B (Methodological)*, Vol. 39, No. 1, pp.1-22, 1977.

[24] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center", In: *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation*, Vol. 11, No. 2011, pp.22-22, 2011.

[25] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator", In: *Proc. of the 4th annual Symposium on Cloud Computing*, pp.5, 2013.

[26] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale". In: *Proc. of the*

twenty-fourth ACM symposium on operating systems principles, pp.423-438, 2013.

[27] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, and D. Xin, "Mllib: Machine learning in apache spark", *The Journal of Machine Learning Research*, Vol. 17, No. 1, pp.1235-1241, 2016.

[28] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework", In: *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation*, pp.599-613, 2014.

[29] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, M. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark", In: *Proc. of the 2015 ACM SIGMOD international Conf. on management of data*, pp.1383-1394, 2015.

[30] T. White, *Hadoop: The definitive guide*, "O'Reilly Media, Inc.", 2012.

[31] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system", In: *Proc. of 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, Vol. 10, pp.1-10, 2010.

[32] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", In: *Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation*, pp.2-2, 2012.

[33] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters", *Communications of the ACM*, Vol. 51, No 1, pp.107-113, 2008.