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

Imad Sassi¹*  Samir Anter¹  Abdelkrim Bekkhoucha¹

Computer Science Laboratory of Mohammadia (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 \((\text{states number})^2\)) 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 Implementation 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...
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 multiprocessor [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

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

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 $\lambda = \{\Pi, A, B\}$ where $\Pi$, $A$ 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 $1 \leq i, j \leq N$, $S_i, S_j \in S$, $t \in \{1,2,\ldots,T\}$

The observation at time $t$, $o_t$, may be a discrete symbol (Discrete HMMs (DHMMs) [22]) or 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 $\mu_j$ and state conditional observation covariance matrix $\Sigma_j$, so $B$ may be defined as:

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

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

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

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

In general, at each instant of time $t$, the model is in one of the states $S_i$, $1 \leq i \leq N$. 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 $S_j$, $1 \leq j \leq N$ 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 $\lambda$ with a given HMM. Second, the decoding problem in which we look for the most likely state sequence in the given model $\lambda$ 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 $\lambda$ and a sequence of observations $O = o_1, o_2, \ldots, o_T$. DOI: 10.22266/ijies2020.0229.20
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 \( \lambda = (A, \mu, \Sigma, \Pi) \) 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 \( \mu \) and covariance matrix \( \Sigma \) 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:

\[
\alpha_t(j) = \pi_j b_j(o_t), \quad 1 \leq j \leq N \tag{7}
\]

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

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

\[
\beta_T(j) = 1, \quad 1 \leq j \leq N \tag{10}
\]

\[
\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j) a_{ij}, \quad 0 \leq i \leq N, 1 \leq t \leq T - 1 \tag{11}
\]

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

\[
\xi_{t}(i,j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})}{Pr(O|\lambda)}, \quad 1 \leq i, j \leq N, 1 \leq t \leq T - 1 \tag{13}
\]

\[
\Xi_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{t}(i,j)}{\sum_{t=1}^{T} \gamma_t(i)}, \quad 1 \leq i, j \leq N \tag{14}
\]

\[
\beta_j = \frac{\sum_{i=1}^{N} \gamma_t(i) a_{ij} \mu_j}{\sum_{i=1}^{N} \gamma_t(i)}, \quad 1 \leq j \leq N, 1 \leq t \leq T \tag{15}
\]

\[
\Sigma_j = \frac{\sum_{i=1}^{N} \gamma_t(i)(o_t - \mu_j)(o_t - \mu_j)^T}{\sum_{i=1}^{N} \gamma_t(i)}, \quad 1 \leq j \leq N, 1 \leq t \leq T \tag{16}
\]

\[
\gamma_i = \alpha_t(i) \beta_t(i), \quad 1 \leq i \leq N \tag{17}
\]

The Baum-Welch algorithm with mixtures of Gaussian continuous observation takes as input an initial model \( \lambda = (A, c_{jm}, \mu_j, \Sigma_j, \Pi) \) 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 \( \mu_j \) and covariance matrix \( \Sigma_j \) of \( m \)th mixture \( \Sigma_{jm} \) 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:

\[
\alpha_t(j) = \pi_j b_j(o_t), 1 \leq j \leq N \tag{18}
\]

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

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

\[
\beta_T(j) = 1, 1 \leq j \leq N \tag{21}
\]

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

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

\[
\xi_{t}(i,j) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1})}{Pr(O|\lambda)}, 1 \leq i, j \leq N, 1 \leq t \leq T - 1 \tag{24}
\]

\[
\Xi_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{t}(i,j)}{\sum_{t=1}^{T} \gamma_t(i)}, 1 \leq i, j \leq N \tag{25}
\]

\[
\beta_j = \frac{\sum_{i=1}^{N} \gamma_t(i) a_{ij} \mu_j}{\sum_{i=1}^{N} \gamma_t(i)}, 1 \leq j \leq N, 1 \leq t \leq T \tag{26}
\]

\[
\Sigma_j = \frac{\sum_{i=1}^{N} \gamma_t(i)(o_t - \mu_j)(o_t - \mu_j)^T}{\sum_{i=1}^{N} \gamma_t(i)}, 1 \leq j \leq N, 1 \leq t \leq T \tag{27}
\]

\[
\gamma_i = \alpha_t(i) \beta_t(i), 1 \leq i \leq N \tag{28}
\]

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 manager), 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), 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).

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 $Pr(0 | \lambda)$, 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 variables 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 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 $\lambda (A, \mu_j, \Sigma_j, \Pi)$, 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 $\lambda (A, \mu_j, \Sigma_j, \Pi)$, 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 $\text{cst}$ 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>
<thead>
<tr>
<th>Table 2. Time complexity comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Classical Baum-Welch</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Classical</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward variable initialization $a_i(j)$</td>
<td>$O(N)$</td>
<td>$\text{cst} O(1)$</td>
</tr>
<tr>
<td>Calculation of $\alpha_{t+1}(i)$</td>
<td>$O(N^2(T - 1))$</td>
<td>$\text{cst} O(T - 1)$</td>
</tr>
<tr>
<td>Calculation of $Pr[O \mid \lambda]$</td>
<td>$O(N)$</td>
<td>$\text{cst} O(1)$</td>
</tr>
<tr>
<td>Backward variable initialization $\beta_r(j)$</td>
<td>$O(N)$</td>
<td>$\text{cst} O(1)$</td>
</tr>
<tr>
<td>Calculation of $y_t(i)$</td>
<td>$O(NT)$</td>
<td>$\text{cst} O(T)$</td>
</tr>
<tr>
<td>Calculation of $z_t(i, j)$</td>
<td>$O(N^2(T - 1))$</td>
<td>$\text{cst} O(T - 1)$</td>
</tr>
<tr>
<td>Calculation of $\pi_{ij}$</td>
<td>$O(N^2)$</td>
<td>$\text{cst} O(1)$</td>
</tr>
<tr>
<td>Calculation of $\bar{\mu}_j$</td>
<td>$O(N)$</td>
<td>$\text{cst} O(1)$</td>
</tr>
<tr>
<td>Calculation of $\bar{\Sigma}_j$</td>
<td>$O(NM)$</td>
<td>$\text{cst} O(1)$</td>
</tr>
<tr>
<td>Calculation of $\bar{\Sigma}_{jm}$</td>
<td>$O(NM)$</td>
<td>$\text{cst} O(1)$</td>
</tr>
<tr>
<td>Calculation of $\pi_i$</td>
<td>$O(N)$</td>
<td>$\text{cst} O(N)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3. Space complexity comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Classical Baum-Welch</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Classical</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward calculation</td>
<td>$O(NT)$</td>
<td>$O(NT)$</td>
</tr>
<tr>
<td>Backward calculation</td>
<td>$O(NT)$</td>
<td>$O(NT)$</td>
</tr>
<tr>
<td>Calculation of $y_t(i)$</td>
<td>$O(NT)$</td>
<td>$O(NT)$</td>
</tr>
<tr>
<td>Calculation of $z_t(i, j)$</td>
<td>$O(N^2(T - 1))$</td>
<td>$O(N^2(T - 1))$</td>
</tr>
<tr>
<td>Calculation of $\pi_{ij}$</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>Calculation of $\bar{\mu}_j$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Calculation of $\bar{\Sigma}_j$</td>
<td>$O(NM)$</td>
<td>$O(NM)$</td>
</tr>
<tr>
<td>Calculation of $\bar{\Sigma}_{jm}$</td>
<td>$O(NM)$</td>
<td>$O(NM)$</td>
</tr>
<tr>
<td>Calculation of $\pi_i$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
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: $\tilde{A} = \{a_{ij}\}, \tilde{\Pi} = \{\pi_j\}, \tilde{B} = \{\text{mean } \mu_j \text{ and variance } \Sigma_j\}$

1: for each executor $i$ of $N$ executors do
2: Parallel do
3: $a_t(j) \leftarrow \pi_j b_j(o_t) \{ j \in [1, 2, 3, \ldots, N]\}$
4: end for
5: for $t \leftarrow 1 \text{ to } T$ do
6: for each executor $i, j$ of $N \times N$ 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: $\Pr(O|\lambda) \leftarrow \text{Alpha}_T . \text{reduce} (\text{lambda } a, b : a + b)$
12: for each executor $i$ 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 $t \leftarrow T-1 \text{ downto } 1$ do
17: for each executor $i, j$ of $N \times N$ 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 $i, j$ of $T \times N$ 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 $i, j$ of $(T-1) \times N$ 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
30: for each executor $i, j$ of $N \times N$ executors do
31: Parallel do
32: $\bar{a}_{ij} \leftarrow \text{sum}(\xi_t(i, j))/\text{sum}(\gamma_t(i)) \{ i, j \in [1, 2, 3, \ldots, N]; \ t \in [1, 2, 3, \ldots, T-1]\}$
33: end for
34: for each executor $i$ of $N$ executors do
35: Parallel do
36: $\bar{\pi}_i \leftarrow \gamma_t(i) \{ i \in [1, 2, 3, \ldots, N]\}$
37: end for
38: for each executor $j$ of $N$ executors do
39: Parallel do
40: $\bar{\mu}_j \leftarrow \text{sum}(\gamma_t(j) o_t)/\text{sum}(\gamma_t(j)) \{ j \in [1, 2, 3, \ldots, N]; \ t \in [1, 2, 3, \ldots, T]\}$
41: end for
42: for each executor $i$ of $N$ executors do
43: Parallel do
44: $\bar{\Sigma}_j = \text{sum}(\gamma_t(j)(o_t-\mu_j)(o_t-\mu_j)^T)/\text{sum}(\gamma_t(j)) \{ j \in [1, 2, 3, \ldots, N]; \ t \in [1, 2, 3, \ldots, T]\}$
45: end for
46: set $\lambda \leftarrow \bar{\lambda}$ and Go to 22 unless some convergence criterion is met
47: return $\tilde{A}, \tilde{\Pi}, \tilde{B}[\bar{\mu}_j, \bar{\Sigma}_j]$. 

International Journal of Intelligent Engineering and Systems, Vol.13, No.1, 2020
DOI: 10.22266/ijies2020.0229.20
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: $\hat{\lambda} = \{a_{ij}\}, \hat{\Pi} = \{\pi_i\}, \hat{B}\{\text{mean of } m^{th} \text{ mixture} \}
\mu_{jm}, \text{covariance of } m^{th} \text{ mixture } \sum_{jm} \text{and } c_{jm} m^{th} \text{ mixture weights}$

1: for each executor $j$ of $N$ executors do
2: Parallel do
3: $\alpha_i(j) \leftarrow \pi_j b_j(o_1) \{j \in [1, 2, 3, \ldots, N]\}$
4: end for
5: for $t \leftarrow 1 \text{ to } T-1$ do
6: for each executor $j$ of $N*N$ executors do
7: Parallel do
8: calculate (map) $\alpha_t(i) a_{ij}$ and store $\alpha_t(i)$ in $\text{Alpha}_t \{i, j \in [1, 2, 3, \ldots, N]\}$
9: sum (reduce) of $\alpha_t(i) a_{ij}$, then multiple by $b_j(o_{t+1}) \{i, j \in [1, 2, 3, \ldots, N]\}$
10: end for
11: end for
12: for each executor $j$ of $N*N$ executors do
13: Parallel do
14: $\beta_t(j) \leftarrow 1 \{j \in [1, 2, 3, \ldots, N]\}$
15: end for
16: for $t \leftarrow T-1 \text{ downto } 0$ do
17: for each executor $i,j$ of $T*N$ executors do
18: Parallel do
19: calculate $\beta_{t+1}(j) a_{ij} b_j(o_{t+1})$ and store $\beta_{t+1}(j) i, j \in [1, 2, 3, \ldots, N]$\}
20: end for
21: end for
22: for each executor $i,j$ of $(T-1)*N*N$ executors do
23: Parallel do
24: $\gamma_t(i) \leftarrow (\alpha_t(i) \beta_t(j) \text{Pr}[O|\lambda] \text{ and store } \gamma_t(i) \text{ in } \text{Gamma}_t \{i, j \in [1, 2, 3, \ldots, N]; t \in [1, 2, 3, \ldots, T]\}$
25: end for
26: for each executor $i,j$ of $(T-1)*N*N$ executors do
27: Parallel do
28: calculate $\gamma_t(i,j) \leftarrow (\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)) / \text{Pr}[O|\lambda] \text{ and store } \gamma_t(i,j)$ in $\text{Gamma}_{2,t} \{i, j \in [1, 2, 3, \ldots, N]; t \in [1, 2, 3, \ldots, T-1]\}$
29: end for
30: for each executor $i,j,m$ of $(T-1)*N*M$ executors do
31: Parallel do
32: calculate $\xi_t(i,j,m) \leftarrow \text{sum} (\alpha_t(i) a_{ij} c_{jm} g_{jm}(o_t) \beta_{t+1}(j)) / \text{Pr}[O|\lambda] \text{ store } \xi_t(i,j,m)$ in $\text{Xi}_{t}$
33: end for
34: for each executor $i,j$ of $N*M$ executors do
35: Parallel do
36: $\overline{\alpha}_{i,j} \leftarrow \text{sum}(\gamma_t(i,j))/ \text{sum}(\gamma_t(i)) \{i, j \in [1, 2, 3, \ldots, N]; t \in [1, 2, 3, \ldots, T-1]\}$
37: end for
38: for each executor $j,m$ of $N*M$ executors do
39: Parallel do
40: $\overline{c}_{jm} \leftarrow \text{sum}(\xi_t(i,j,m))/ \text{sum}(\gamma_t(i)) \{i, j \in [1, 2, 3, \ldots, N]; t \in [1, 2, 3, \ldots, T-1]\}$
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 $a_t(i), b_t(i), y_t(i)$ and $ξ_t(i,j)$ 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


