Multi-level optimization of the canonical polyadic tensor decomposition at large-scale: Application to the stratification of social networks through deflation

Annabelle Gillet, Éric Leclercq, Nadine Cullot

Multi-level optimization of the canonical polyadic tensor decomposition at large-scale: Application to the stratification of social networks through deflation

Annabelle Gillet, Éric Leclercq, Nadine Cullot

LIB EA 7534
University of Burgundy Franche Comté
Dijon, France

Abstract

Tensors are multi-dimensional mathematical objects that allow to model complex relationships and to perform decompositions for analytical purpose. They are used in a wide range of data mining applications. In social network analysis, tensor decompositions give interesting insights by taking into consideration multiple characteristics of data. However, the power-law distribution of such data forces the decomposition to reveal only the strong signals that hide information of interest having a lighter intensity. To reveal hidden information, we propose a method to stratify the signal, by gathering clusters of similar intensity in each stratum. It is an iterative process, in which the CANDECOMP/PARAFAC (CP) decomposition is applied and its result is used to deflate the tensor, i.e., by removing from the tensor the clusters found with the decomposition. As the CP decomposition is computationally demanding, it is also necessary to optimize its algorithm, to apply it on large-scale data with a reasonable execution time, even with the several executions needed by the iterative process of the stratification. Therefore, we propose an algorithm that uses both dense and sparse data structures and that leverages coarse and fine grained optimizations in addition to incremental computations in order to achieve large scale CP tensor decomposition. Our implementation outperforms the baseline of large-scale CP decomposition libraries by several orders of magnitude. We validate our stratification method and our optimized algorithm on a Twitter dataset about COVID vaccines.

Keywords: Tensor decomposition, Data mining, Multi-dimensional analytics

1. Introduction

Tensors are powerful mathematical objects, which bring capabilities to model multi-dimensional data [1]. They are used in multiple analytics frameworks, such as Tensorflow [2], PyTorch [3], Theano [4], TensorLy [5], where their ability to represent various models is a great advantage. Furthermore, associated with powerful decompositions, they can be used as data mining tools to reveal the hidden value of Big Data. Tensor decompositions are used for various purposes such as dimensionality reduction, noise elimination, identification of latent factors, pattern discovery, ranking, recommendation and data completion. They are applied in a wide range of applications, including genomics [6], analysis of health records [7] and graph mining [8]. Papalexakis et al. in [9] review major usages of tensor decompositions in data mining applications.

The analysis of social networks benefits also greatly from tensor decompositions [10, 11]. For example, it can serve as a tool to detect communities, while including several characteristics such as a temporal

Email address: annabelle.gillet@u-bourgogne.fr, eric.leclercq@u-bourgogne.fr, nadine.cullot@u-bourgogne.fr (Annabelle Gillet, Éric Leclercq, Nadine Cullot)

dimension or the use of keywords. However, the power-law distribution of most of the social network interactions [12] leads analytics algorithms to often extract strong signals that hide other information of interest. Indeed, some short-lasting subjects or some opinions that are discussed by a small group of users might be undetected when using traditional analytics techniques, if the group is smaller than the groups discussing of the major topics. So, algorithms have to be adapted to tackle this issue.

Furthermore, as tensors model multi-dimensional data, their global size varies exponentially depending on the number and size of their dimensions, making them sensitive to large-scale issues. Most of tensor libraries that include decompositions work with tensors of limited size, and do not consider the large-scale challenge. For example, some intermediate structures needed in the algorithms produce a data explosion, such as the Khatri-Rao product in the CANDECOMP/PARAFAC decomposition [13] (CP, also called canonical polyadic decomposition). Thus, analyzing Big Data with tensors requires optimization techniques and suitable implementations, able to scale up. These optimizations are directed towards different computational aspects, such as the memory consumption, the execution time or the scaling capabilities, and can follow different principles, such as coarse grained optimizations, fine grained optimizations or incremental computations.

In this article, we focus on the CP decomposition that allows to factorize a tensor into smaller and more usable sets of vectors [14], and which is largely adopted in exploratory analyzes. We propose to extend the CP decomposition with a stratification method, that reveals interesting signals of lighter intensity. To do so, we rely on deflation [15], that consists in removing from the original tensor the strong signals found by the decomposition, and reitering the process with the new tensor. We validate our stratification method on a real world Twitter dataset on COVID-19. Nevertheless, as it requires multiple executions of the decomposition, the efficiency of the algorithm used is critical.

To cope with the need of performance, we also propose an optimized algorithm to achieve large scale CP decomposition, that uses dense or sparse data structures depending on what suits best each step, and that leverages incremental computation, coarse and fine grained optimizations to improve the algorithm. We provide an implementation, named MuLOT[1], in Scala using Spark 3.0 that outperforms the state of the art of large-scale tensor CP decomposition libraries.

The rest of the article is organized as follows: section 2 presents an overview of tensors including the CP decomposition and an empirical study of the behavior of the CP decomposition, section 3 introduces a state of the art of uses of CP decomposition and tensor manipulation libraries, section 4 describes our stratification method based on deflation, section 5 describes our scalable and optimized algorithm, as well as the experiments we ran to compare our algorithm to other large-scale CP decomposition libraries, section 6 presents a study on real data performed with our stratification method and finally section 7 concludes.

2. Overview of tensors and CP decomposition

In this section, we present the notion of tensor along with major operators used in the CP decomposition, and we perform an empirical study in order to intuit how the decomposition works and how the results can be interpreted.

2.1. Background of tensors

Tensors are general abstract mathematical objects which can be considered according to various points of view such as a multi-linear application, or as the generalization of matrices to multiple dimensions. We will use the definition of a tensor as an element of the set of the functions from the product of N sets $I_n, n = 1, \ldots, N$ to $\mathbb{R} : \boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, where N is the number of dimensions of the tensor or its order or its mode. Table 1 summarizes the notations used in this article. We adopt the notations of [1].

Tensor operations, by analogy with operations on matrices and vectors, are multiplications, transpositions, matricizations (or unfolding) and decompositions (also named factorizations). We only highlight the

[1]The implementation is open source and available on Github, along with experimental evaluations to validate its efficiency especially at large scale : `https://github.com/AnnabelleGillet/MuLOT`

most significant operators on tensors which are used in our algorithm. The reader can consult [14, 1] for an overview of the major operators.

Symbol	Definition	Symbol	Definition
$\boldsymbol{\mathcal{X}}$	A tensor	$\circ$	Outer product
$\mathbf{M}$	A matrix	$\otimes$	Kronecker product
$\mathbf{v}$	A column vector	$\circledast$	Hadamard product
a	A scalar	$\oslash$	Hadamard division
$\mathbf{m}_i$	The i^{th} column vector of the matrix $\mathbf{M}$	$\odot$	Khatri-Rao product
		$\dagger$	Pseudo inverse
$m_{i,j}$	The element of the matrix $\mathbf{M}$ at the i^{th} line and j^{th} column	$\boldsymbol{\mathcal{X}}_{(n)}$	Matricization of the tensor $\boldsymbol{\mathcal{X}}$ on mode-n
$x_{i_1,\ldots,i_N}$	The element of the N-order tensor $\boldsymbol{\mathcal{X}}$ at the indexes $(i_1,\ldots,i_N)$		

Table 1: Symbols and operators used

The **mode-n matricization** of a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ noted $\boldsymbol{\mathcal{X}}_{(n)}$ produces a matrix $\mathbf{M} \in \mathbb{R}^{I_n \times \Pi_{j \neq n} I_j}$, where:

$$m_{i_n,j} = x_{i_1,\ldots,i_n,\ldots,i_N} \text{ with } j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^{N} (i_k - 1) \prod_{\substack{m=1 \\ m \neq n}}^{k-1} I_m$$

The **outer product** between a tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and another tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_M}$ noted $\boldsymbol{\mathcal{Y}} \circ \boldsymbol{\mathcal{X}}$ produces a tensor $\boldsymbol{\mathcal{Z}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N \times J_1 \times J_2 \times \cdots \times J_M}$ in which the elements are equal to:

$$z_{i_1,i_2,\ldots,i_N,j_1,j_2,\ldots,j_M} = y_{i_1,i_2,\ldots,i_N} x_{j_1,j_2,\ldots,j_M}$$

The **Hadamard product** between two matrices of same size $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{I \times J}$ noted $\mathbf{A} \circledast \mathbf{B}$ is an element-wise multiplication that produces a matrix $\mathbf{C} \in \mathbb{R}^{I \times J}$:

$$c_{i,j} = a_{i,j} b_{i,j}$$

The **Kronecker product** between two matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{K \times L}$ noted $\mathbf{A} \otimes \mathbf{B}$ produces a matrix $\mathbf{C} \in \mathbb{R}^{(IK) \times (JL)}$, in which every elements of $\mathbf{A}$ are multiplied by the matrix $\mathbf{B}$:

$$c_{m,n} = a_{i,j} b_{k,l} \text{ where } m = k + (i-1)K \text{ and } n = l + (j-1)L$$

The **Khatri-Rao product** between two matrices having the same number of columns $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{K \times J}$ is noted $\mathbf{A} \odot \mathbf{B}$ and performs the column-wise Knrocker product that results in a matrix $\mathbf{C} \in \mathbb{R}^{(IK) \times J}$. The elements are computed by:

$$c_{m,j} = a_{i,j} b_{k,j} \text{ where } m = k + (i-1)K$$

The **pseudo inverse** of a matrix $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$ is $\mathbf{A}^\dagger = \mathbf{V}\boldsymbol{\Sigma}^\dagger\mathbf{U}^T$, in which $\mathbf{U}$ represents the left singular vectors, $\mathbf{V}$ represents the right singular vectors, and $\boldsymbol{\Sigma}$ has the singular values on its diagonal. $\boldsymbol{\Sigma}^\dagger$ can be obtained by replacing each singular value σ_k of $\boldsymbol{\Sigma}$ by $1/\sigma_k$.

The canonical polyadic decomposition allows to factorize a tensor into smaller and more exploitable sets of vectors [16, 17]. Given a N-order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and a rank $R \in \mathbb{N}$, the CP decomposition factorizes the tensor $\boldsymbol{\mathcal{X}}$ into N column-normalized factor matrices $\mathbf{A}^{(i)} \in \mathbb{R}^{I_i \times R}$ for $i = 1,\ldots,N$ with their scaling factors $\boldsymbol{\lambda} \in \mathbb{R}^R$ as follows:

$$\boldsymbol{\mathcal{X}} \simeq [\![\boldsymbol{\lambda}, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!] = \sum_{r=1}^{R} \lambda_r \mathbf{a}_r^{(1)} \circ \mathbf{a}_r^{(2)} \circ \cdots \circ \mathbf{a}_r^{(N)}$$

Algorithm 1 CP-ALS

Require: Tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and target rank R
1: Initialize $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$, with $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$
2: **repeat**
3: **for** $n = 1, \ldots, N$ **do**
4: $\mathbf{V} \leftarrow \mathbf{A}^{(1)T}\mathbf{A}^{(1)} \circledast \ldots \circledast \mathbf{A}^{(n-1)T}\mathbf{A}^{(n-1)} \circledast \mathbf{A}^{(n+1)T}\mathbf{A}^{(n+1)} \circledast \ldots \circledast \mathbf{A}^{(N)T}\mathbf{A}^{(N)}$
5: $\mathbf{A}^{(n)} \leftarrow \boldsymbol{\mathcal{X}}_{(n)}(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)})\mathbf{V}^{\dagger}$
6: normalize columns of $\mathbf{A}^{(n)}$
7: $\boldsymbol{\lambda} \leftarrow$ norms of $\mathbf{A}^{(n)}$
8: **end for**
9: **until** < convergence >

where $\mathbf{a}_r^{(i)}$ are columns of $\mathbf{A}^{(i)}$.

Several algorithms have been proposed to compute the CP decomposition [18], we focus on the alternating least squares (ALS) one (algorithm 1). At line 1, the initialization can be done by randomly filling the factor matrices. Then, the iterative phase of the algorithm begins until a convergence criterion is met (lines 2 to 9). During the iterative phase, two major steps are repeated for each dimension n: 1) $\mathbf{V} \in \mathbb{R}^{R \times R}$ is computed by multiplying the transpose of each factor matrix by the same factor matrix except for the factor matrix of the dimension n, and by performing the Hadamard product on all these results of multiplication (line 4); and 2) updating the factor matrix corresponding to the dimension n by multiplying the matricized tensor $\boldsymbol{\mathcal{X}}$ by the result of the Khatri-Rao product between all the factor matrices except the one of the dimension n and by the pseudo inverse of $\mathbf{V}$ (line 5). The columns of the updated factor matrix are then normalized and stored in $\boldsymbol{\lambda}$ (lines 6 and 7).

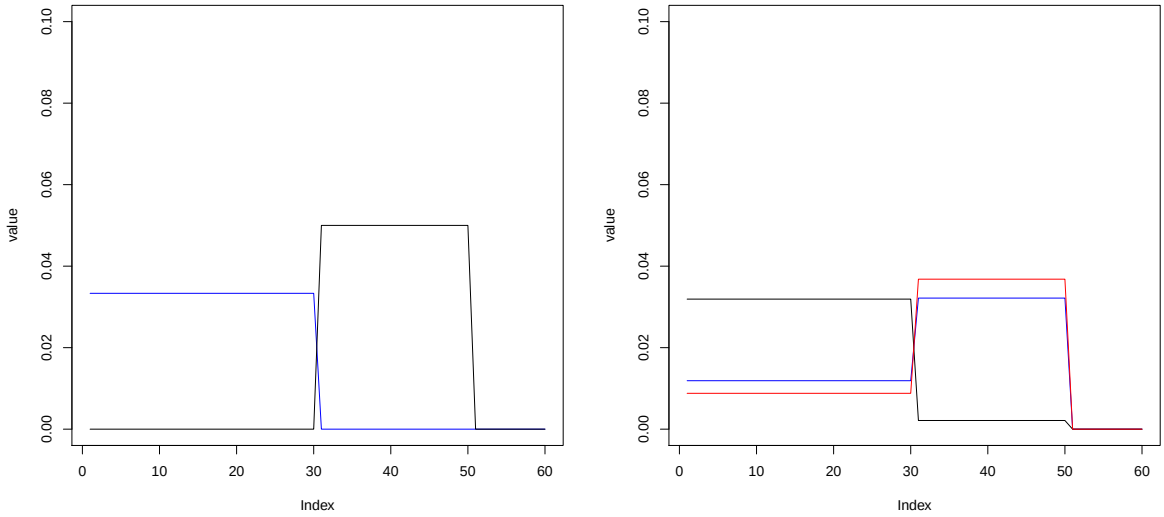
The Matricized Tensor Times Khatri-Rao Product (MTTKRP, line 5 of the algorithm 1) is often the target of optimizations, because it involves the tensor matricized of size $\mathbb{R}^{I_n \times J}$, with $J = \prod_{j \neq n} I_j$, as well as the result of the Khatri-Rao product of size $\mathbb{R}^{J \times R}$. It is thus computationally demanding and uses a lot of memory to store the dense temporary matrix resulting of the Khatri-Rao product [19]. For example, by considering that a value is stored as a double on 1 byte, for a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{10\,000 \times 5\,000 \times 2\,000 \times 10}$ with a sparsity of 10^{-7} that could represent the users, the hashtags they use, a temporal dimension and the country of the users, the tensor can be stored with $10\,000 \times 5\,000 \times 2\,000 \times 10 \times 10^{-7} = 100\,000$B = 100kB. However, as the factor matrices are dense, the Khatri-Rao product produces a dense matrix of size $\mathbb{R}^{10\,000 \times 100\,000\,000}$ for the first dimension (for the other dimensions the number of lines and columns changes but the overall size stays the same) that needs $10\,000 \times 100\,000\,000 = 1\,000\,000\,000\,000$B = 1 000GB of storage.

2.2. Empirical study

In order to better understand the capabilities and the limits of the CP decomposition, we conduct an empirical study on simple tensors to illustrate its behavior on an interpretable result.

We build a 3-order tensor $\boldsymbol{\mathcal{X}}$ of size $60 \times 60 \times 60$, that contains three clusters that do not overlap: one of size $30 \times 30 \times 30$, the second of size $20 \times 20 \times 20$ and the last one of size $10 \times 10 \times 10$. We run the CP decomposition on this tensor, with rank-2 and -3. The results of $\mathbf{A}^{(1)}$ are shown in figure 1, in which the highest values represent the elements that participate the most to the rank (only the results for the first dimension are drawn, but they are identical for the other dimensions given the shape of the clusters). They allow to deduce that the optimal rank is two, because the third rank does not bring more information than the two previous ranks.

The third cluster is not found: even if its values are the same as for the other clusters, its smaller size leads to a lower participation in the signal of the global tensor. We try to reveal it by first applying a deflation technique. A tensor deflation consists in removing some elements of it, often depending on the result of a decomposition. We remove the cluster of size $30 \times 30 \times 30$ to reduce the global intensity of the signal, and execute a rank-2 decomposition on the deflated tensor (figure 2). The third cluster is now taken into consideration by the decomposition.



(a) Rank-2

(b) Rank-3

Figure 1: CP decomposition on a tensor of size $60 \times 60 \times 60$ with three clusters, the x-axis represents the indexes of the elements, the y-axis represents the value of each element and each line corresponds to a rank of the decomposition

Another interesting behaviour of the decomposition is the probability of a rank-1 decomposition to find a given cluster among others of different signal intensity. To better understand this, we build a 3-order tensor of size $60 \times 60 \times 60$ with three non-overlapping clusters of size $20 \times 20 \times 20$, one with each element having 5 as value, the second with each element having 10 as value and the last with each element having 15 as value. All of these clusters are detected by a rank-3 CP decomposition. We execute a rank-1 CP decomposition 1000 times on the tensor, and record which cluster is found at each run (table 2). The weight of the cluster in the tensor seems to play a role in its probability of finding it among the others with a rank-1 CP decomposition.

Value of elements in the cluster	Apparition frequency
15	62%
10	32%
5	6%

Table 2: Probability of apparition of each cluster with a rank-1 CP decomposition

Based on these phenomena, we can assume that: 1) a deflation can be necessary to find clusters that are also valuable but hidden by some strong signals; and 2) when the rank of the decomposition is not high enough, it will find some of the existing clusters and most of the time it will be clusters with a high signal intensity in the tensor. By taking into consideration these observations, we propose a stratification method that consists in progressively revealing clusters of lighter signal intensity, in order to find hidden but interesting information in data.

3. State of the art

This section first reviews the use of the CP decomposition as analytics tool, as well as extension of the CP decomposition to apply it on particular use cases. The second part of the section describes major tensor libraries, in order to identify what is missing to execute the CP decomposition at large scale.

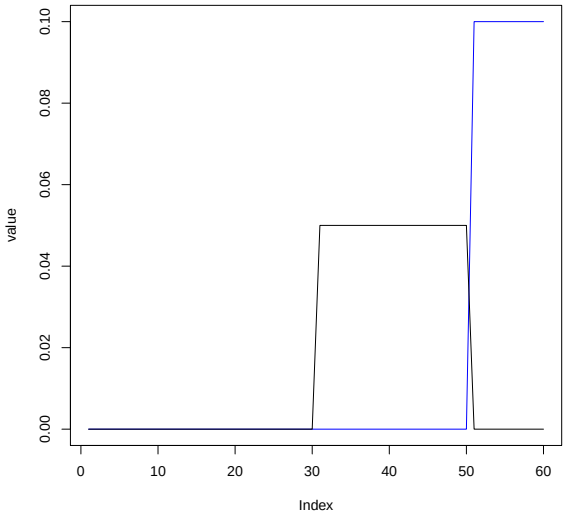


Figure 2: Rank-2 CP decomposition on a deflated tensor of size $60 \times 60 \times 60$ with two clusters

3.1. The CANDECOMP/PARAFAC decomposition as analytics tool

The CP decomposition is a useful tool in data analytics. Gauvin et al. [20] applied it on a primary school dataset that contains the interactions among the students. They were able to reveal clusters that gather students of a same class thanks to the decomposition, and to get a temporal signature of these clusters. Indeed, the results also showed clusters containing students of different classes, and the temporal signature helped to see that these clusters appeared only during the break and lunch hours, when students mixed themselves with each other. Similar experiments have been conducted on the Enron dataset [21] that contains email interactions among employees, the MovieLens or the DARPA1998 [22] datasets, that contains respectively movies watched by users and network traffic information. The experiment on DARPA1998 aims to detect anomalies (e.g., network attacks). Fernandes el al. [11] recently described different applications of tensor decompositions on social network data, including clustering and anomaly detection. They also detailed some methods based on the CP decomposition that aim to extend it.

Some works are based on the CP decomposition, and extend it to try to improve its results in given use cases. Araujo et al. with Com2 [15] seek to find comet communities in social networks, namely communities that appear during a period of time and that can appear again later. To do so, they compute a rank-1 CP decomposition, they evaluate the consistency of the community with a Minimum Description Length (MDL) measure and they deflate the tensor from the found community to repeat the process. Their goal is to avoid having to choose a rank to perform the decomposition by computing successive rank-1 decomposition, but, as seen in the section 2.2, this method reveals clusters that would not have been found in an initial decomposition with the perfect rank, and it is hard to detect when the clusters have little or no importance. Sheikholeslami and Giannakis propose EgoTen [23]. It relies on a special modeling of the network in the tensor, by storing the adjacency matrix of each user rather than the interactions among the users. By doing so, they focus on discovering communities than can be overlapping, and each user has a membership proportion for its different communities. However, with this method it is hard to add some contextual information, as for example the keywords used by the users.

The CP decomposition has already proven its usefulness to analyze social network data, especially to find communities. Nevertheless, methods often detect the major communities and neglect those of lighter signal intensity that can be hidden due to the power law distribution of such data. Thus, it is necessary to propose a method that can reveal hidden clusters that can present an interest for the analyzes. However,

the performance of the algorithm to do so must be taken into consideration, in order to be able to process large amount of data.

3.2. Tensor libraries

Several tensor libraries have been proposed. They can be classified in three categories: 1) the mathematical libraries; 2) the libraries which use tensors to convey data among tasks; and 3) specialized libraries, which implement few tensorial operators. Only the most representative libraries are presented in this section, Psarras et al. [24] have recently published a wider overview.

Mathematical libraries are often written in Matlab or R, and aim to make available tensorial operators. However, they have often limited performances when the size of the tensors increases. rTensor[2] provides users with standard operators to manipulate tensors in R language including tensor decompositions, but does not support sparse tensors. Tensor Algebra COmpiler (TACO) [25] provides optimized tensor operators in C++, from vector multiplication to MTTKRP, by relying on compressed formats such as Compressed Sparse Row (CSR) or Compressed Sparse Fiber (CSF). High-Performance Tensor Transpose [26] is a C++ library only for tensor transpositions, thus it lacks lots of useful operators. It uses micro-kernels to parallelize operations. TensorToolbox[3] is a Matlab library relying on Kolda's work. It proposes several tensorial operators including Tucker and CP decomposition.

Libraries that use tensors to convey data among tasks of a workflow are often used in machine learning applications. TensorFlow [2] and PyTorch [3] are two Python libraries, that are used in machine learning and deep learning contexts. But these libraries have only few tensorial operators, and mostly focus on developing classic methods of machine learning. NumPy [27] is another Python library, that proposes multi-dimensional arrays manipulation operators. TensorLy [5] allows to switch its backend library (such as TensorFlow or PyTorch), but is closer to the mathematical libraries because it includes more tensorial operators than its backend libraries alone, including tensor decompositions.

Specialized libraries implement only a specific operator or algorithm. Some of these libraries can be used at large scale. We focus on those that implement tensor decompositions. HaTen2 [13] is a Hadoop implementation of the CP and Tucker decompositions using the map-reduce paradigm. It was later improved with BigTensor [22]. It was one of the first library allowing to perform decompositions at large scale, but the use of Hadoop requires to store data as files in HDFS, and thus forces to preprocess data before being able to run the decomposition and before manipulating the results. SamBaTen [28] proposes an incremental CP decomposition for evolving tensors. The authors developed a Matlab and a Spark implementations. However, it can only compute the decomposition on 3-order tensors, and the first computation of the decomposition is a costly operation. Gudibanda et al. in [29] developed a CP decomposition for coupled tensors using Spark (i.e., different tensors having a dimension in common). ParCube [30] is a parallel Matlab implementation of the CP decomposition. However it can only take 3- or 4-order tensors, and cannot distribute operations. CSTF [31] is based on Spark and proposes a distributed CP decomposition. Nevertheless, it relies mainly on sparse structures, thus it limits the applicable optimizations.

As a conclusion, the study of the state of the art shows some limitations of the proposed solutions. The mathematical libraries often take into consideration only small tensors. The libraries that use tensors to convey data among tasks have a limited vision of tensors (i.e., only as multi-dimensional arrays), and do not support all the functionalities of tensors. Specialized libraries are often only prototypes.

More generally, libraries focus on a specific type of optimization, and use only sparse structures to handle the sparsity of large tensors. This is a bottleneck to performance, as they do not consider all the characteristics of the algorithm (i.e., the factor matrices are dense). Furthermore, they are not really data centric, as they manipulate tensors only with integer indexes, for dimensions and for values of dimensions. Thus it reduces greatly the user-friendliness as the mapping between meaningful values (e.g., user name

[2] `http://jamesyili.github.io/rTensor/`
[3] `https://www.tensortoolbox.org/`

or timestamp) and indexes has to be supported by users. The Hadoop implementations need a particular input format, thus necessitate data transformations to execute the decomposition and to interpret the results, leading to laborious prerequisites and increasing the risk of mistakes when working with the results. Table 3 summarizes the characteristics of specialized libraries able to handle large scale tensors, and compares them with ours.

4. Stratification of social network data through tensor deflation

As demonstrated in the section 2.2, the CP decomposition will only find clusters that produce a strong signal in a tensor. However, with social network data and their power-law distribution, the most important clusters produce a signal strong enough to hide many other clusters of interest, but with a lighter intensity. So, the objective of our stratification method is to find these hidden clusters, by producing several strata that gather clusters with a similar signal intensity. Each new stratum reveals clusters with a signal lighter than those of the previous stratum. To do so, the method can be summarized in two steps: 1) computing the CP decomposition on the tensor to find the clusters; and 2) deflating the tensor to remove the strongest signal found by the decomposition. These two steps are repeated, and the resulting clusters of each iteration are gathered into a stratum (figure 3). We detail these steps below.

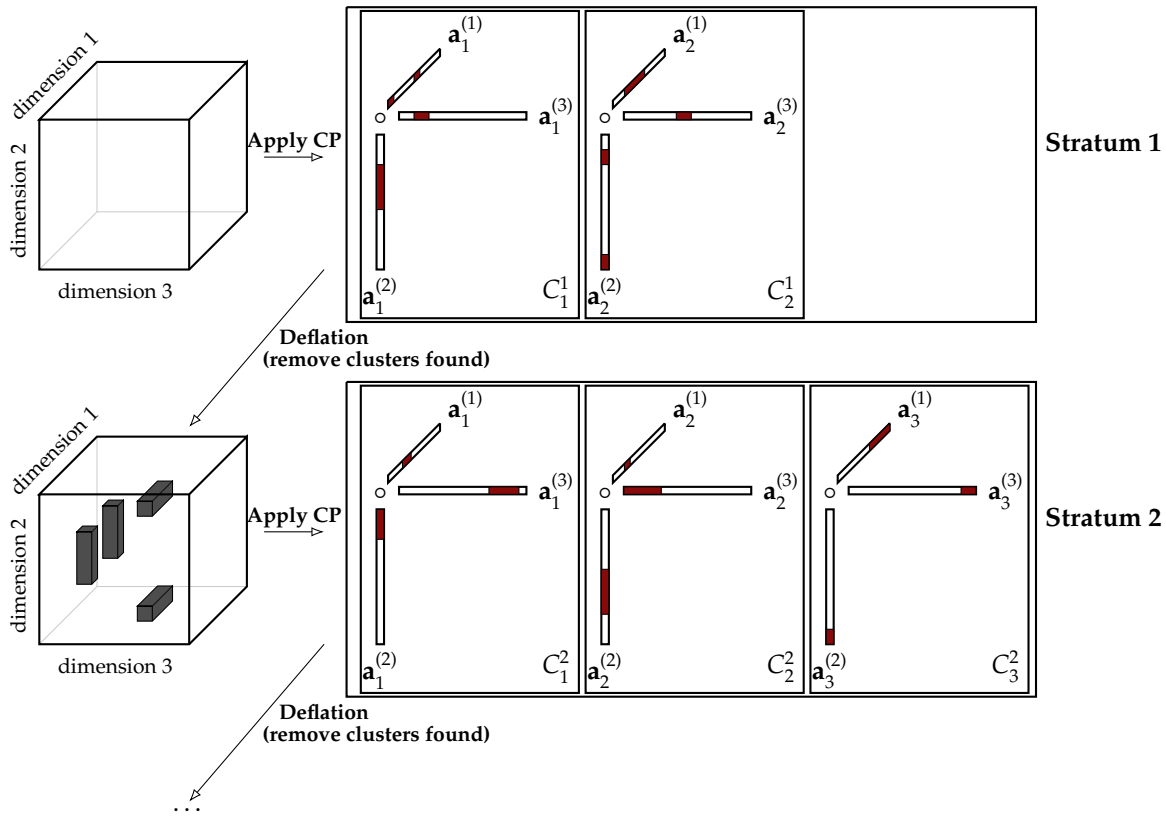


Figure 3: The stratification method (the black blocks represent the clusters removed from the tensor with the deflation, the red part of each vector are the elements of $D_r^{s,n}$)

The first step of the stratification method, applied on a N-order tensor $\mathcal{X}$, consists in executing the CP decomposition on the tensor. To choose the rank of the decomposition, the CORe CONsistency DIAgnostic (CORCONDIA) [32] can be used. It evaluates if increasing the rank will bring new information or not. The result of a rank-R^s CP decomposition on a tensor $\mathcal{X}^s$ is used to create a stratum $S^s_{R^s}$, such as:

Library	Language/ Framework	Number of dimensions	Input format	Indexes of dimensions	Optimization	Code available
HaTen2 [13]	Java/Hadoop	n	CSV stored in HDFS	Integer	Use of sparsity	✓
BigTensor [22]	Java/Hadoop	n	CSV stored in HDFS	Integer	Use of sparsity	✓
SamBaTen [28]	Scala/Spark 2.2	3	Specific Spark structure	Integer	Use of sparsity	✓
CSTF [31]	Scala/Spark 1.5	3	CSV file	Integer	Use of sparsity	✓
ParCube [30]	Matlab	3 or 4	Tensor from Tensor Toolbox	Integer	Use of sparsity, parallel only	✓
[29]	Spark	not specified	not specified	Integer	Use of sparsity	
This paper	**Scala/Spark 3.0**	n	**Spark Dataframe**	**Meaningful values (String, Integer, etc.)**	**Use of sparsity, density and incremental computations**	✓

Table 3: Comparison of large scale tensor libraries to compute the CP decomposition

$$S^s_{R^s} = (\boldsymbol{\mathcal{X}}^s, \{C^s_r \text{ for } r = 1 \dots R^s\})$$

with s the index of the stratum. As the stratification method is an iterative process, the first execution of the decomposition is done on the input tensor, so $\boldsymbol{\mathcal{X}}^1 = \boldsymbol{\mathcal{X}}$.

The R^s clusters C^s_r that compose a stratum contain, for each dimension, a set $D^{s,n}_r$ of the indices of the elements of the dimension n that contribute to the cluster. A cluster is defined by:

$$C^s_r = (D^{s,1}_r, \dots, D^{s,N}_r)$$

And a set $D^{s,n}_r$ is defined by:

$$D^{s,n}_r = \{i \text{ such as } i \in [1, I_n] \text{ and } a^{(n)}_{i,r} \geq f(\mathbf{a}^{(n)}_r) \text{ with } \mathbf{A}^{(n)} \in CP_{R^s}(\boldsymbol{\mathcal{X}}^s)\}$$

$CP_{R^s}(\boldsymbol{\mathcal{X}}^s)$ indicates the execution of the CP decomposition with rank-R^s on a tensor $\boldsymbol{\mathcal{X}}^s$. The function f is used to evaluate if the value of an element given by the decomposition is high enough to consider that the element contributes to the cluster. For example, by relying on the null hypothesis that if all elements contributed equally to the cluster they would all have the same value in the vector, the function can be the average of the vector.

The first stratum is created from the input tensor. In order to discover other strata, the tensor must be deflated to remove the clusters having a strong signal intensity. To do so, the clusters of the stratum obtained at the previous iteration are used. The values of the tensor that are indexed on all the dimensions by elements that are part of a same cluster in the previous stratum are removed from the tensor such as:

$$\boldsymbol{\mathcal{X}}^{s+1}_{i_1,\dots,i_N} = \begin{cases} 0, & \text{if } i_1 \in D^{s,1}_r, \dots, i_N \in D^{s,N}_r \text{ with } r \in [1, R^s] \\ \boldsymbol{\mathcal{X}}^s_{i_1,\dots,i_N}, & \text{otherwise} \end{cases}$$

The deflation step allows to reduce the maximal signal intensity of the tensor, and thus to reveal clusters having a lighter signal intensity when executing a CP decomposition on the deflated tensor. The first stratum shows the obvious clusters contained in data, and the latter stratum reveals clusters that are harder to detect but that still present an interest to better understand the content of the data. However, as the stratification method is an iterative process that can be executed on large real data (e.g., social network data, biological networks), the CP decomposition must be optimized to produce a result within an acceptable execution time.

5. Distributed, scalable and optimized ALS for Apache Spark

Optimizing the ALS algorithm for the CP tensor decomposition induces several challenges, that gain importance proportionally to the size of the data.

First, the **data explosion of the MTTKRP** is a serious computational bottleneck (line 5 of algorithm 1), that can overflow memory, and prohibit to work with large tensors, even if they are extremely sparse. Indeed, the matrix produced by the Khatri-Rao has $J \times R$ non-zero elements, with $J = \Pi_{j \neq n} I_j$, for an input tensor of size $\mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$. We propose to optimize carefully this operation, in order to avoid the data explosion and to improve significantly the execution time (see algorithm 3).

The main operations in the ALS algorithm, i.e., the updates of the factor matrices, **are not themselves parallelizable** (lines 4 and 5 of algorithm 1). In such a situation, it is profitable to think of other methods to take advantage of parallelism, that could be applied on fine grained operations. For example, leveraging parallelism for matrix multiplications is an optimization that can be applied in many situations. This also eases the reuse of such optimizations, without expecting specific characteristics from the algorithm (see section 5.2).

The **nature of data structures used in the CP decomposition are mixed**: tensors are often sparse, while factor matrices are dense. Their needs to be efficiently implemented diverge, so rather than sticking globally

to sparse data structures to match the sparsity of tensors, each structure should take advantage of their particularities to improve the whole execution (see section 5.1). To the best of our knowledge, this strategy has not been explored by others.

The **stopping criterion** can also be a bottleneck. In distributed implementations of the CP ALS, the main solutions used to stop the algorithm are to repeat the 3 main steps (lines 4 to 7) for a fixed number of iterations, or to compute the Frobenius norm on the difference between the input tensor and the tensor reconstructed from the factor matrices. The first solution severely lacks in precision, and the second is computationally demanding as it involves outer products between all the factor matrices. However, an other option is available to check the convergence, and consists in measuring the similarity of the factor matrices between two iterations, as suggested in [1, 14]. It is a very efficient solution at large-scale, as it combines precision and light computations (see section 5.3).

Finally, the implementation should facilitate the **data loading**, and should avoid data transformations only needed to fit the expected input of the algorithm. It should also produce easily interpretable results, and minimize the risk of errors induced by laborious data manipulations (see section 5.4). The study of the state of the art of tensor libraries showed that tensors are often used as multi-dimensional arrays, that are manipulated through their indexes, even if they represent real world data. The mapping between indexes and values is delegated to the user, although being an error-prone step. As it is a common task, it should be handled by the library.

To tackle these challenges, we leverage three optimization principles to develop an efficient decomposition: coarse grained optimization, fine grained optimization, and incremental computation. The coarse grained one relies on specific data structures and capabilities of Spark to efficiently distribute operations. The incremental computation is used to avoid to compute the whole Hadamard product at each iteration. The fine grained optimization is applied on the MTTKRP to reduce the storage of large amount of data and costly computations. For this, we have extended Spark's matrices with the operations needed for the CP decomposition. In addition, we choose to use an adapted converging criteria, efficient at large-scale. For the implementation of the algorithm, we take a data centric point of view to facilitate the loading of data and the interpretation of the results. By doing so, our CP decomposition implementation is able to process tensors with billions of elements (i.e., non zero entries) on a mid-range server, and small and medium size tensors can be processed in a short time on a low-end personal computer.

5.1. Distributed and scalable matrix data structures

A simple but **efficient sparse matrix storage structure** is COO (COOrdinate storage) [33, 34]. The `CoordinateMatrix`, available in the `mllib` package of Spark [35], is one of these structures, that stores only the coordinates and the value of each existing element in a `RDD` (Resilient Distributed Datasets). **It is well suited to process sparse matrices**, such as a matricized tensor.

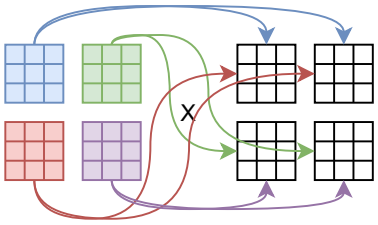


Figure 4: Blocks mapping for a multiplication between two `BlockMatrix`

Another useful structure is the `BlockMatrix`. It is composed of multiple blocks containing each a fragment of the global matrix. Operations can be parallelized by executing it on each sub-matrix. For binary operations such as multiplication, only blocks from each `BlockMatrix` that will be associated are sent to each other, and the result is then aggregated if needed (see figure 4). **It is thus an efficient structure for dense matrices**, such as the factor matrices, and allows distributed computations to process all blocks.

Unfortunately, only some basic operations are available for `BlockMatrix`, as for example the multiplication or addition. The more complex ones, including the Hadamard and Khatri-Rao products, are missing. We have extended Spark `BlockMatrix` with more advanced operations, that keep the coarse grained optimization logic of the multiplication. We also added new operations, that involve `BlockMatrix` and `CoordinateMatrix` to take advantage of the both structures for our optimized MTTKRP (see below).

5.2. Mixing three principles of optimization

Tensors have generally a high level of sparsity. In the CP decomposition, they only appear under their matricized form, thus they are naturally manipulated as a `CoordinateMatrix` data structure in our implementation. On the other hand, the factor matrices $\mathbf{A}$ of the CP decomposition are dense, because they hold information for each index of each dimension. They greatly benefit from the capabilities of the extended `BlockMatrix` we developed. By using the most suitable structure for each part of the algorithm, we leverage specific optimizations that can speed up the whole algorithm.

Algorithm 2 CP-ALS adapted to Spark

Require: Tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and target rank R
1: Initialize $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$, with $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$
2: $\mathbf{V} \leftarrow \mathbf{A}^{(1)T}\mathbf{A}^{(1)} \circledast \ldots \circledast \mathbf{A}^{(N)T}\mathbf{A}^{(N)}$
3: **repeat**
4: **for** $n = 1, \ldots, N$ **do**
5: $\mathbf{V} \leftarrow \mathbf{V} \oslash \mathbf{A}^{(n)T}\mathbf{A}^{(n)}$
6: $\mathbf{A}^{(n)} \leftarrow MTTKRP(\boldsymbol{\mathcal{X}}_{(n)}, (\mathbf{A}^{(N)}, \ldots, \mathbf{A}^{(n+1)}, \mathbf{A}^{(n-1)}, \ldots, \mathbf{A}^{(1)}))\mathbf{V}^{\dagger}$
7: $\mathbf{V} \leftarrow \mathbf{V} \circledast \mathbf{A}^{(n)T}\mathbf{A}^{(n)}$
8: normalize columns of $\mathbf{A}^{(n)}$
9: $\boldsymbol{\lambda} \leftarrow$ norms of $\mathbf{A}^{(n)}$
10: **end for**
11: **until** < convergence >

Besides using and improving Spark's matrices according to the specificities of data, we also have introduced fine grained optimization and incremental computation into the algorithm to avoid costly operations in terms of memory and execution time. These improvements are synthesized in algorithm 2 and explained below.

First, to avoid computing $\mathbf{V}$ completely at each iteration for each dimension, we propose to do it incrementally. Before iterating, we calculate the Hadamard product for all $\mathbf{A}$ (line 2 of the algorithm 2). At the beginning of the iteration, $\mathbf{A}^{(n)T}\mathbf{A}^{(n)}$ is element-wise divided from $\mathbf{V}$, giving the expected result at this step (line 5 of the algorithm 2). At the end of the iteration, the Hadamard product between the new $\mathbf{A}^{(n)T}\mathbf{A}^{(n)}$ and $\mathbf{V}$ prepares $\mathbf{V}$ for the next iteration (line 7 of the algorithm 2).

The MTTKRP part (line 6 of the algorithm 2) is sensitive to improvement, as stated in section 2. Indeed, by focusing on the elements of the result rather than on the operation, it can be easily reordered. For example, if we multiply a 3-order tensor matricized on dimension 1 with the result of $\mathbf{A}^{(3)} \odot \mathbf{A}^{(2)}$, we can notice that, in the result, the indexes of the dimensions in the tensor $\boldsymbol{\mathcal{X}}$ correspond directly to those in the matrices $\mathbf{A}^{(3)}$ and $\mathbf{A}^{(2)}$. This behavior is represented below in an example simplified with only one rank:

$$\begin{bmatrix} x_{1,1,1} & x_{1,2,1} & x_{1,1,2} & x_{1,2,2} \\ x_{2,1,1} & x_{2,2,1} & x_{2,1,2} & x_{2,2,2} \end{bmatrix} \times \begin{bmatrix} a^{(2)}_{1,1}a^{(3)}_{1,1} \\ a^{(2)}_{2,1}a^{(3)}_{1,1} \\ a^{(2)}_{1,1}a^{(3)}_{2,1} \\ a^{(2)}_{2,1}a^{(3)}_{2,1} \end{bmatrix} =$$

$$\begin{bmatrix} x_{1,1,1}a^{(2)}_{1,1}a^{(3)}_{1,1} + x_{1,2,1}a^{(2)}_{2,1}a^{(3)}_{1,1} + x_{1,1,2}a^{(2)}_{1,1}a^{(3)}_{2,1} + x_{1,2,2}a^{(2)}_{2,1}a^{(3)}_{2,1} \\ x_{2,1,1}a^{(2)}_{1,1}a^{(3)}_{1,1} + x_{2,2,1}a^{(2)}_{2,1}a^{(3)}_{1,1} + x_{2,1,2}a^{(2)}_{1,1}a^{(3)}_{2,1} + x_{2,2,2}a^{(2)}_{2,1}a^{(3)}_{2,1} \end{bmatrix}$$

Thus, rather than computing the full Khatri-Rao product and performing the multiplication with the matricized tensor, we apply a fine grained optimization that takes advantage of the mapping of indexes, and that anticipates the construction of the final matrix. For each entry of the `CoordinateMatrix` of the matricized tensor (i.e., all non-zero values), we select in each factor matrix $\mathbf{A}$ which element will be used, and compute elements of the final matrix (algorithm 3). As this optimization allows to skip the temporary storage of the dense result of the Khatri-Rao product of size $\mathbb{R}^{I_n \times J}$ with $J = \prod_{j \neq n} I_j$, it is a major gain in memory space.

Algorithm 3 Detail of the MTTKRP

Require: The index of the factor matrix n, the matricized tensor $\boldsymbol{\mathcal{X}}_{(n)} \in \mathbb{R}^{I_n \times J}$ with $J = \Pi_{j \neq n} I_j$ and $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(n-1)}, \mathbf{A}^{(n+1)}, \ldots, \mathbf{A}^{(N)}$, with $\mathbf{A}^{(i)} \in \mathbb{R}^{I_i \times R}$

Initialize $\mathbf{A}^{(n)}$ with zeros, with $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$
for each (x, y, v) in $\boldsymbol{\mathcal{X}}_{(n)}$ with x, y the coordinates and v the value **do**
  for $r = 1, \ldots, R$ **do**
    $value \leftarrow v$
    for each $\mathbf{A}^{(i)}$ with $i \neq n$ **do**
      $c \leftarrow$ extract $\mathbf{A}^{(i)}$ coordinate from y
      $value \leftarrow value \times a^{(i)}_{c,r}$
    end for
    $a^{(n)}_{x,r} \leftarrow a^{(n)}_{x,r} + value$
  end for
end for

5.3. Stopping criterion

To evaluate the convergence of the algorithm and decide when to stop iterating, a majority of CP decomposition implementations uses the Frobenius norm on the difference between the original tensor and the reconstructed tensor from the factor matrices. However, at large-scale the reconstruction of the tensor from the factor matrices is an expensive computation, even more than the naive MTTKRP. Waiting for a predetermined number of iterations is not very effective to avoid unnecessary iterations. Thus, other stopping criteria such as the evaluation of the difference between the factor matrices with those of the previous iteration [1, 14] are much more interesting, as they work on smaller chunks of data. To this end, we use the Factor Match Score (FMS) [36] to measure the difference between factor matrices of the current iteration ($[\![\boldsymbol{\lambda}, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!]$) and those of the previous iteration ($[\![\hat{\boldsymbol{\lambda}}, \hat{\mathbf{A}}^{(1)}, \hat{\mathbf{A}}^{(2)}, \ldots, \hat{\mathbf{A}}^{(N)}]\!]$). The FMS is defined as follows:

$$FMS = \frac{1}{R} \sum_{r=1}^{R} \left(1 - \frac{\xi - \hat{\xi}}{max(\xi, \hat{\xi})}\right) \prod_{n=1}^{N} \frac{\mathbf{a}_r^{(n)T} \hat{\mathbf{a}}_r^{(n)}}{\|\mathbf{a}_r^{(n)}\| . \|\hat{\mathbf{a}}_r^{(n)}\|}$$

where $\xi = \lambda_r \prod_{n=1}^{N} \|\mathbf{a}_r^{(n)}\|$ and $\hat{\xi} = \hat{\lambda}_r \prod_{n=1}^{N} \|\hat{\mathbf{a}}_r^{(n)}\|$.

The use of the FMS allows us to significantly improve the performance of the decomposition on large tensors, without sacrificing precision.

5.4. Data centric implementation

Our implementation of the CP decomposition, in addition to being able to run with any number of dimensions, is data centric: it takes a Spark `DataFrame` as input to execute the CP directly on real data. Thus, it benefits from Spark capabilities to retrieve data directly from various datasources, as most of DBMS are supported as well as file formats.

A specific column of the `DataFrame` contains the values of the tensor and all the other columns contain the values for each dimension. The CP operators returns a map associating the original names of the dimensions to a new `DataFrame` with three columns for each dimension: the dimension's values, the rank,

and the value computed by the CP decomposition. By using a `DataFrame` as input, we allow the use of any type as dimensions' values. For example, users could create a `DataFrame` with four columns: username, hashtag, time and value, with username and hashtag being of type `String` in order to easily interpret the decomposition result. This avoids having to handle an intermediate data structure containing the mapping between indexes and real values, and thus reduces the risk of mistakes when transforming data.

5.5. *Performance study*

To validate our algorithm, we have run experiments[4] on tensors produced by varying the size of dimensions and the sparsity, on a Dell PowerEdge R740 server (Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 20 cores, 256GB RAM). We compare our execution time to those of the baseline of distributed CP tensor decomposition libraries available: HaTen2 [13], BigTensor [22], SamBaTen [28] and CSTF [31]. Hadoop 2.6.0 was used to execute HaTen and BigTensor. TensorLy [5] is used as reference of non-distributed library, with its default backend (NumPy). The measured times do not include the loading of data to be as neutral as possible, as for some libraries it requires the creation of a specific file to perform the decomposition.

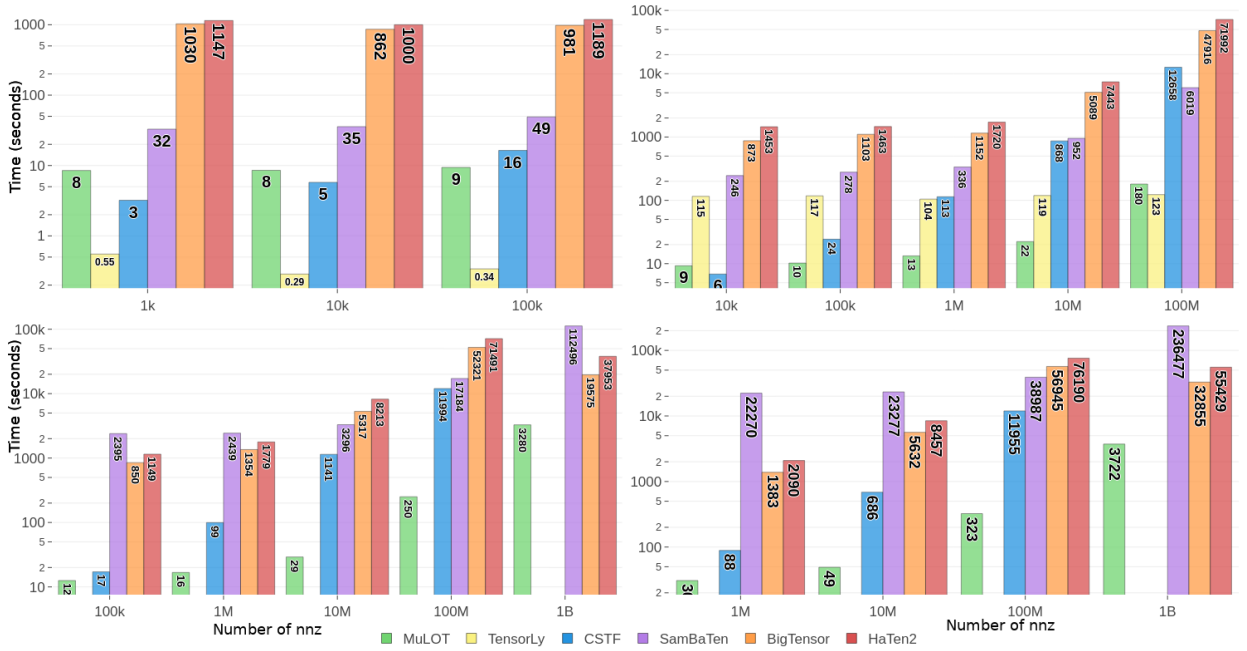


Figure 5: Execution time for tensors with 3 dimensions of size 100 (top-left), 1 000 (top-right), 10 000 (bottom-left) and 100 000 (bottom-right). CSTF produces an Out Of Memory exception for tensors with 1B elements, and TensorLy cannot load data with dimensions of size 10 000 and 100 000

Tensors were created randomly with 3 dimensions of the same size, from 100 to 100k. The sparsity ranges from 10^{-1} to 10^{-9}, and tensors were created only if the number of non-zero elements is superior to $3\times size$ and inferior or equal to 1B (with dimensions of size 100 and 1 000, tensors can only have respectively 10^6 and 10^9 non-zero elements at most, with a sparsity up to 10^{-1} they cannot reach 1B elements, but respectively 10^5 and 10^8 non-zero elements). As the libraries do not use the same convergence criteria, we used a fixed number of iterations. We have run the CP decomposition for 5 iterations, and have measured the execution time. Results are shown in figure 5.

Our implementation clearly outperforms the state of the art, with speed-up reaching several orders of magnitude. CSTF keeps up concerning the execution time of small tensors, but is no match for large

[4]The source code of the experiments and the tool used to create tensors are available at `https://github.com/AnnabelleGillet/MuLOT/tree/main/experiments`

tensors, and cannot compute the decomposition for tensors with 1B elements. Execution times of MuLOT are nearly linear following the number of non-zero elements. The optimization techniques applied show efficient results even for very large tensors of billion elements, with a maximum execution time for a 3-order tensor with dimensions of size 100k of 62 minutes, while the closest, BigTensor, takes 547 minutes. The execution time for small tensors with dimensions of size 100 is acceptable as it takes less than 10 seconds, but compared to TensorLy it could be improved with a non-distributed implementation. However, TensorLy does not take into consideration the sparsity of the tensors as shown by the execution time on tensors with dimensions of size 1 000, and it cannot process tensors with dimensions of size 10 000 or greater.

The differences of result with the CSTF library are interesting to study. As we both use Spark, it allows to see the benefits of our algorithm. Even if the Spark version of CSTF is older (1.5 for CSTF and 3.0 for our implementation), it was already depending on efficient math libraries (breeze[5], that itself depends on BLAS, LAPACK and ARPACK), thus the improved execution time comes mainly from our optimized algorithm. Indeed, CSTF does not use a dense structure for the factor matrices, and cannot apply operations that would benefit from dense structures. Furthermore, CSTF recomputes the $\mathbf{V}$ at each iteration, while we do it incrementally. It also uses the Frobenius norm as stopping criterion, that is less efficient than the Factor Match Score that we use.

6. Analysis of the Twitter dataset related to COVID vaccines

We have experimented our stratification method in the context of Cocktail[6], an interdisciplinary research project aiming to study trends and weak signals in discourses about food and health on Twitter. We focus on french tweets revolving around COVID-19 vaccines, harvested with Hydre, a high performance system to collect, store and analyze tweets [37]. The corpus contains 55 315 877 tweets from the period of December 1st 2020 to October 30th 2021. The code and the anonymized dataset are available[7].

To apply the stratification method, we have built a 3-order tensor $\mathcal{UHT}$, with one dimension containing the users, one the hashtags, and one the time with a granularity of 1 day. The values of the tensor represent the number of tweets produced by a given user, that contained a given hashtag at a given day. Only the tweets that have been retweeted at least 10 times are kept. The most representative ranks of each stratum are shown in figure 6 (stratum 1), figure 7 (stratum 2) and figure 8 (stratum 3). In these figures, the 20 users and hashtags that have the highest values in the rank are shown (the users have been anonymized), and the time is plotted for the whole period.

In a rank of the stratum 1, we can see a strong concern for the vaccination topic in general, with hashtags such as "Astrazeneca", "Vaccination" or "Pfizer". It is present almost at anytime of the corpus, with a spike in March/April, when the vaccination started in France.

In a rank of the stratum 2, we can detect a topic about long COVID, in which we can find the hashtags "ApresJ20" (after day 20), "UnDonPourLeCovidLong" (a donation for the long COVID) or "CovidLongEnfants" (kids long COVID). Although being an important topic, it is not as present in the dataset as the vaccination discourse.

In the last stratum, we can find a rank about the evaluation of the BTS (the french equivalent of the Higher National Diploma), that has been switched from final tests to continuous assessment. It was a temporary issue that appeared during the exam period, around May 2021. Another rank is close to a conspiracy discourse about the origin of COVID, with hashtags such as "CCPVirus", "UnrestrictedBioWeapon" or "CCP_Is_Terrorist". It can be seen as an isolated discourse that is not propagated into the dataset.

To conclude this experiment, the stratification method is able to reveal interesting signals that have lighter intensity in each new stratum. In the stratum 3, it even shows topics that can be considered as weak signals given the volume and the extended time period of the dataset. These results have been validated by social scientists of the Cocktail project.

[5] https://github.com/scalanlp/breeze

[6] https://projet-cocktail.fr/

[7] https://github.com/AnnabelleGillet/MuLOT/tree/main/experiments/Stratification

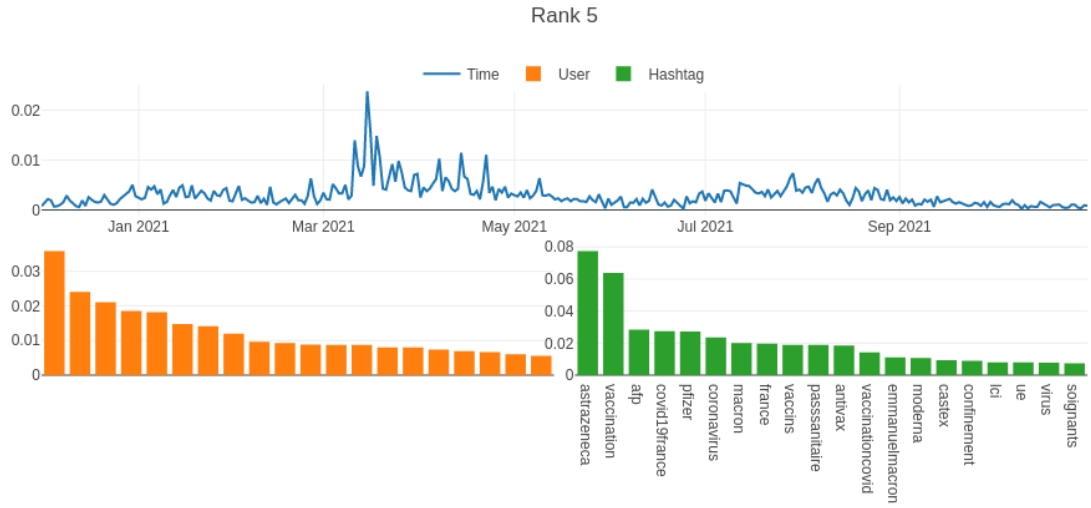


Figure 6: Stratum 1 of the stratification

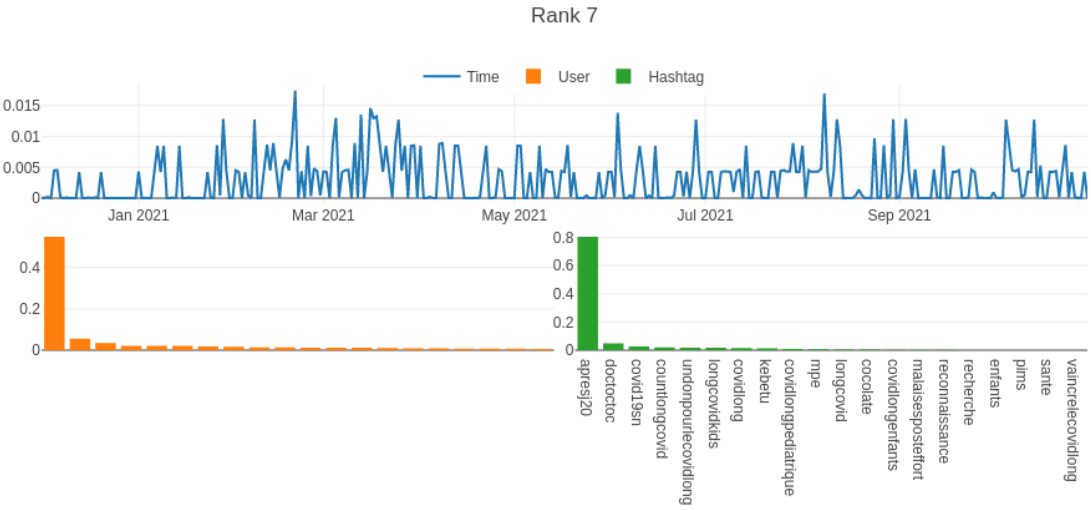


Figure 7: Stratum 2 of the stratification

7. Conclusion

We have proposed a stratification method based on tensor deflation, allowing to find hidden signals in data while keeping information about the level of intensity of the signal. Furthermore, as this method requires several execution of the CP decomposition, we have also proposed an optimized algorithm to execute the CP decomposition at large-scale. We have validated this algorithm with a Spark implementation, and showed that it outperforms the state of the art by several orders of magnitude. We applied our stratification method along with our CP algorithm on a Twitter dataset about COVID vaccines, in which we found discourses of different intensity, including some that could be qualified as weak signal.

We plan to continue our work on tensor decompositions by 1) developing other tensor decompositions such as Tucker, HOSVD or DEDICOM; and 2) studying the impact of the choice of the norm for the scaling

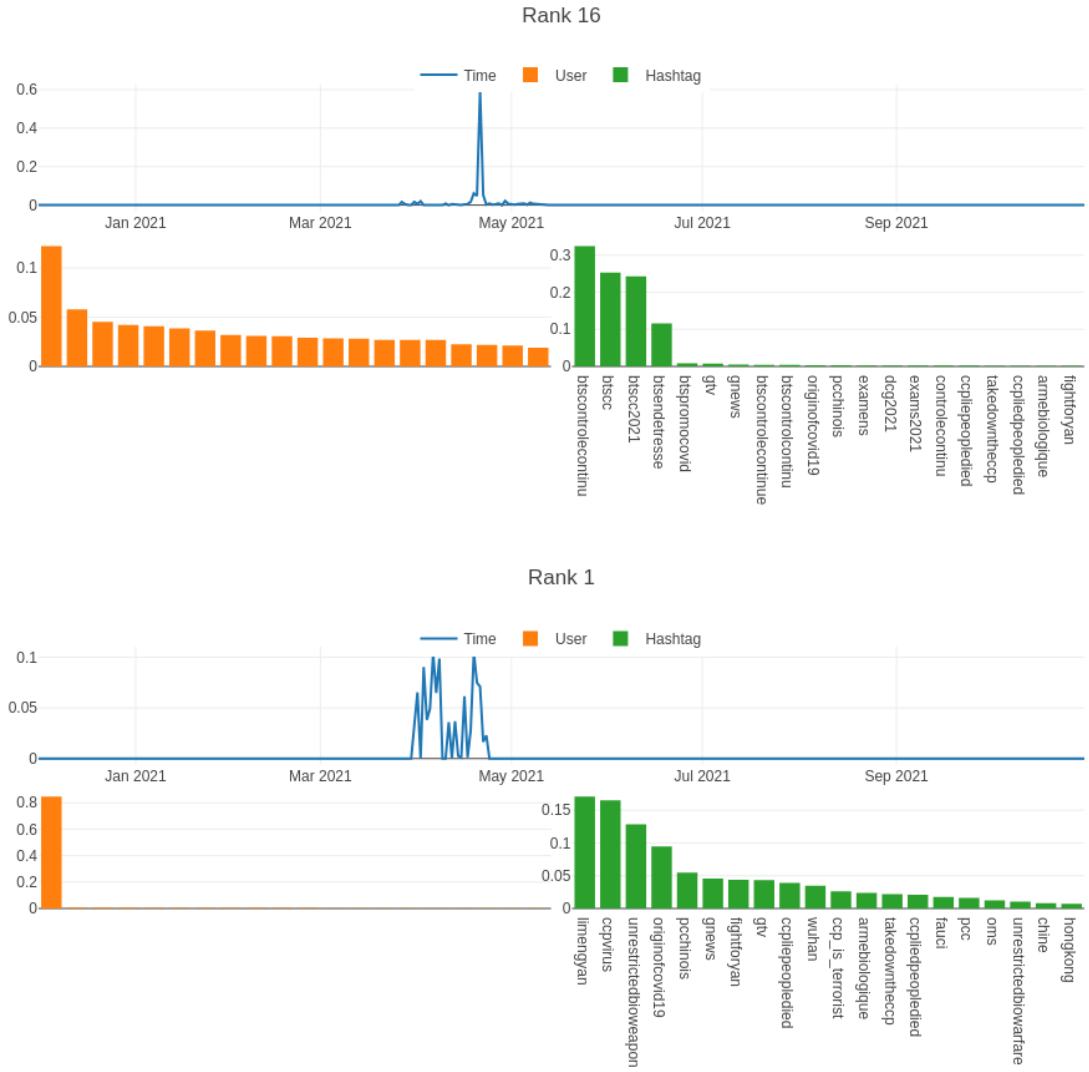


Figure 8: Stratum 3 of the stratification

of the factor matrices.

Acknowledgments This work is supported by ISITE-BFC (ANR-15-IDEX-0003) coordinated by G. Brachotte, CIMEOS Laboratory (EA 4177), University of Burgundy.

References

[1] A. Cichocki, R. Zdunek, A. H. Phan, S. Amari, Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation, John Wiley & Sons, 2009.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: A system for large-scale machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation, 2016, pp. 265–283.

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., PyTorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems, 2019, pp. 8024–8035.

[4] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, et al., Theano: A Python framework for fast computation of mathematical expressions, arXiv:1605.02688.
[5] J. Kossaifi, Y. Panagakis, A. Anandkumar, M. Pantic, Tensorly: Tensor learning in python, The Journal of Machine Learning Research 20 (1) (2019) 925–930.
[6] V. Hore, A. Viñuela, A. Buil, J. Knight, M. I. McCarthy, K. Small, J. Marchini, Tensor decomposition for multiple-tissue gene expression experiments, Nature genetics 48 (9) (2016) 1094–1100.
[7] K. Yang, X. Li, H. Liu, J. Mei, G. Xie, J. Zhao, B. Xie, F. Wang, Tagited: Predictive task guided tensor decomposition for representation learning from electronic health records, in: Proc. of the Thirty-First AAAI Conference on Artificial Intelligence, 2017.
[8] J. Sun, D. Tao, C. Faloutsos, Beyond streams and graphs: dynamic tensor analysis, in: ACM SIGKDD International Conference on Knowledge Discovery and Data mining, ACM, 2006, pp. 374–383.
[9] E. E. Papalexakis, C. Faloutsos, N. D. Sidiropoulos, Tensors for data mining and data fusion: Models, applications, and scalable algorithms, Transactions on Intelligent Systems and Technology (TIST) 8 (2) (2016) 16.
[10] E. E. Papalexakis, L. Akoglu, D. Ience, Do more views of a graph help? community detection and clustering in multi-graphs, in: International Conference on Information Fusion, IEEE, 2013, pp. 899–905.
[11] S. Fernandes, H. Fanaee-T, J. Gama, Tensor decomposition for analysing time-evolving social networks: An overview, Artificial Intelligence Review 54 (4) (2021) 2891–2916.
[12] L. A. Adamic, B. A. Huberman, A. Barabási, R. Albert, H. Jeong, G. Bianconi, Power-law distribution of the world wide web, science 287 (5461) (2000) 2115–2115.
[13] I. Jeon, E. E. Papalexakis, U. Kang, C. Faloutsos, Haten2: Billion-scale tensor decompositions, in: International Conference on Data Engineering, IEEE, 2015, pp. 1047–1058.
[14] T. G. Kolda, B. W. Bader, Tensor decompositions and applications, SIAM review 51 (3) (2009) 455–500.
[15] M. Araujo, S. Papadimitriou, S. Günnemann, C. Faloutsos, P. Basu, A. Swami, E. E. Papalexakis, D. Koutra, Com2: fast automatic discovery of temporal ('comet') communities, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2014, pp. 271–283.
[16] R. A. Harshman, et al., Foundations of the PARAFAC procedure: Models and conditions for an" explanatory" multimodal factor analysis, Tech. rep. (1970).
[17] S. Rabanser, O. Shchur, S. Günnemann, Introduction to tensor decompositions and their applications in machine learning, arXiv preprint arXiv:1711.10781.
[18] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, C. Faloutsos, Tensor decomposition for signal processing and machine learning, Transactions on Signal Processing 65 (13) (2017) 3551–3582.
[19] A.-H. Phan, P. Tichavskỳ, A. Cichocki, Fast alternating ls algorithms for high order candecomp/parafac tensor factorizations, Transactions on Signal Processing 61 (19) (2013) 4834–4846.
[20] L. Gauvin, A. Panisson, C. Cattuto, Detecting the community structure and activity patterns of temporal networks: a non-negative tensor factorization approach, PloS one 9 (1) (2014) e86028.
[21] E. E. Papalexakis, C. Faloutsos, N. D. Sidiropoulos, Parcube: Sparse parallelizable tensor decompositions, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2012, pp. 521–536.
[22] N. Park, B. Jeon, J. Lee, U. Kang, Bigtensor: Mining billion-scale tensor made easy, in: ACM International on Conference on Information and Knowledge Management, 2016, pp. 2457–2460.
[23] F. Sheikholeslami, G. B. Giannakis, Identification of overlapping communities via constrained egonet tensor decomposition, IEEE Transactions on Signal Processing 66 (21) (2018) 5730–5745.
[24] C. Psarras, L. Karlsson, J. Li, P. Bientinesi, The landscape of software for tensor computations, arXiv preprint arXiv:2103.13756.
[25] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, S. Amarasinghe, The tensor algebra compiler, OOPSLA (2017) 1–29.
[26] P. Springer, T. Su, P. Bientinesi, Hptt: a high-performance tensor transposition c++ library, in: ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, 2017, pp. 56–62.
[27] S. Van Der Walt, S. C. Colbert, G. Varoquaux, The NumPy array: a structure for efficient numerical computation, Computing in Science & Engineering 13 (2) (2011) 22.
[28] E. Gujral, R. Pasricha, E. E. Papalexakis, Sambaten: Sampling-based batch incremental tensor decomposition, in: International Conference on Data Mining, SIAM, 2018, pp. 387–395.
[29] A. Gudibanda, T. Henretty, M. Baskaran, J. Ezick, R. Lethin, All-at-once decomposition of coupled billion-scale tensors in apache spark, in: High Performance extreme Computing Conference, IEEE, 2018, pp. 1–8.
[30] E. E. Papalexakis, C. Faloutsos, N. D. Sidiropoulos, ParCube: Sparse parallelizable CANDECOMP-PARAFAC tensor decomposition, ACM Transactions on Knowledge Discovery from Data (TKDD) 10 (1) (2015) 1–25.
[31] Z. Blanco, B. Liu, M. M. Dehnavi, Cstf: Large-scale sparse tensor factorizations on distributed platforms, in: Proceedings of the 47th International Conference on Parallel Processing, 2018, pp. 1–10.
[32] R. Bro, H. A. Kiers, A new efficient method for determining the number of components in parafac models, Journal of Chemometrics: A Journal of the Chemometrics Society 17 (5) (2003) 274–286.
[33] N. Ahmed, N. Mateev, K. Pingali, P. Stodghill, A framework for sparse matrix code synthesis from high-level specifications, in: SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, IEEE, 2000, pp. 58–58.
[34] N. Goharian, A. Jain, Q. Sun, Comparative analysis of sparse matrix algorithms for information retrieval, computer 2 (2003) 0–4.
[35] R. Bosagh Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, M. Zaharia, Matrix computations and optimization in apache spark, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 31–38.
[36] E. C. Chi, T. G. Kolda, On tensors, sparsity, and nonnegative factorizations, SIAM Journal on Matrix Analysis and Applications 33 (4) (2012) 1272–1299.

[37] A. Gillet, É. Leclercq, N. Cullot, Lambda+, the renewal of the lambda architecture: Category theory to the rescue, in: International Conference on Advanced Information Systems Engineering (CAiSE), Springer, 2021, pp. 381–396.