Identifying and Alleviating Concept Drift in Streaming Tensor Decomposition

Ravdeep Pasricha(✉), Ekta Gujral, and Evangelos E. Papalexakis

Department of Computer Science and Engineering, University of California Riverside, 900 University Avenue, Riverside, CA, USA
{rpasr001,egujr001}@ucr.edu, epapalex@cs.ucr.edu

Abstract. Tensor decompositions are used in various data mining applications from social network to medical applications and are extremely useful in discovering latent structures or *concepts* in the data. Many real-world applications are dynamic in nature and so are their data. To deal with this dynamic nature of data, there exist a variety of online tensor decomposition algorithms. A central assumption in all those algorithms is that the number of latent concepts remains fixed throughout the entire stream. However, this need not be the case. Every incoming batch in the stream may have a different number of latent concepts, and the difference in latent concepts from one tensor batch to another can provide insights into how our findings in a particular application behave and deviate over time. In this paper, we define "concept" and "concept drift" in the context of streaming tensor decomposition, as the manifestation of the variability of latent concepts throughout the stream. Furthermore, we introduce *SeekAndDestroy* (The method name is after (and a tribute to) Metallica's song from their first album (who also owns the copyright for the name)), an algorithm that detects concept drift in streaming tensor decomposition and is able to produce results robust to that drift. To the best of our knowledge, this is the first work that investigates concept drift in streaming tensor decomposition. We extensively evaluate *SeekAndDestroy* on synthetic datasets, which exhibit a wide variety of realistic drift. Our experiments demonstrate the effectiveness of *SeekAndDestroy*, both in the detection of concept drift and in the alleviation of its effects, producing results with similar quality to decomposing the entire tensor in one shot. Additionally, in real datasets, *SeekAndDestroy* outperforms other streaming baselines, while discovering novel useful components. Code related to this paper is available at: https://github.com/ravdeep003/conceptDrift.

Keywords: Tensor analysis · Streaming · Concept drift
Unsupervised learning

1 Introduction

Data comes in many shapes and sizes. Many real world applications deal with data that is multi-aspect (or multi-dimensional) in nature. An example of multi-aspect data would be interactions between different users in a social network over

© Springer Nature Switzerland AG 2019
M. Berlingerio et al. (Eds.): ECML PKDD 2018, LNAI 11052, pp. 327–343, 2019.
https://doi.org/10.1007/978-3-030-10928-8_20

period of time. Interactions like who messages whom, who liked whose posts or who shared (re-tweet) whose post. This can be modeled as a three-mode tensor, user-user being two modes of the tensor and time being the third mode, where each data point can be considered as an interaction between two users.

Tensor decomposition has been used in many data mining applications and is an extremely useful tool for finding latent structures in tensor in an unsupervised fashion. There exist a wide variety of tensor decomposition models and algorithms available, interested readers can refer to [9,13] for details. In this paper, our main focus is on CP/PARAFAC decomposition [7] (henceforth refered to as CP for brevity), which decomposes a tensor into a sum of rank-one tensors, each one being a latent factor (or *concept*) in the data. CP has been widely used in many applications, due to its ability to uniquely uncover latent components in a variety of unsupervised multi-aspect data mining applications [13].

In today's world data is not static, data keeps on evolving over time. In real world applications like stock market and e-commerce websites hundred of transaction (if not thousands) takes place every second, or in applications like social media where every second, thousands of new interactions take place forming new communities of users who interact with each other. In this example, we consider each *community* of people within the graph as a *concept.*

There has been a considerable amount of work in dealing with online or streaming CP decomposition [6,11,16], where the goal is to absorb the updates to the tensor in the already computed decomposition, as they arrive, and avoid recomputing the decomposition every time new data arrives. However, despite the already existing work in the literature, a central issue has been left, to the best of our knowledge, entirely unexplored. All of the existing online/streaming tensor decomposition literature assumes that the concepts in the data (whose number is equal to the rank of the decomposition) remains *fixed* throughout the lifetime of the application. What happens if the number of components changes? What if a new component is introduced, or an existing component splits into two or more new components? This is an instance of *concept drift* in unsupervised tensor analysis, and this paper is a look at this problem from first principles.

Our contributions in this paper are the following:

- **Characterizing concept drift in streaming tensors:** We define concept and concept drift in time evolving tensors and provide a quantitative method to measure the concept drift.
- **Algorithm for detecting and alleviating concept drift in streaming tensor decomposition:** We provide an algorithm which detects drift in the streaming data and also updates the previous decomposition without any assumption on the rank of the tensor.
- **Experimental evaluation on real and synthetic data:** We extensively evaluate our method on both synthetic and real datasets and out-perform state of the art methods in cases where the rank is not known a priori and perform on par in other cases.
- **Reproducibility:** Our implementation is made publicly available[1] for reproducibility of experiments.

[1] https://github.com/ravdeep003/conceptDrift.

2 Problem Formulation

2.1 Tensor Definition and Notations

Tensor $\underline{\mathbf{X}}$ is collection of stacked matrices $(\mathbf{X}_1, \mathbf{X}_2, \dots \mathbf{X}_K)$ with dimension $\mathbb{R}^{I \times J \times K}$, where I and J represents rows and columns of matrix and K represents number of views. In other words, a tensor is a higher order abstraction of a matrix. For simplicity, we call the term "dimension" as "mode" of tensor, where "modes" are the numbers of views used to index the tensor. The rank($\underline{\mathbf{X}}$) is the minimum number of rank-1 tensors computed from its latent components which are required to re-produce $\underline{\mathbf{X}}$ as their sum. Table 1 represents the notations used throughout the paper.

Table 1. Table of symbols and their description

Symbols	Definition
$\underline{\mathbf{X}}, \mathbf{X}, \mathbf{x}, x$	Tensor, Matrix, Column vector, Scalar
$\mathbb{R}$	Set of Real Numbers
$\circ$	Outer product
$\|\mathbf{A}\|_F, \|\mathbf{a}\|_2$	Frobenius norm, ℓ_2 norm
$\mathbf{X}(:, r)$	r^{th} column of $\mathbf{X}$
$\odot$	Khatri-Rao product (column-wise Kronecker product [13])

Tensor Batch: A batch is a (N-1)-mode partition of tensor $\underline{\mathbf{X}} \in \mathbb{R}^{I \times J \times K}$ where size is varied only in one mode and other modes remain unchanged. Here, tensor $\underline{\mathbf{X}}_{new}$ is of dimension $\mathbb{R}^{I \times J \times t_{new}}$ and existing tensor $\underline{\mathbf{X}}_{old}$ is of dimension $\mathbb{R}^{I \times J \times t_{old}}$. The full tensor $\underline{\mathbf{X}} = [\underline{\mathbf{X}}_{old}; \underline{\mathbf{X}}_{new}]$ where its temporal mode $K = t_{old} + t_{new}$. The tensor $\underline{\mathbf{X}}$ can be partitioned into horizontal $\underline{\mathbf{X}}$(I,:,:), lateral $\underline{\mathbf{X}}$(:,J,:), and frontal $\underline{\mathbf{X}}$(:,:,K) mode.

CP Decomposition: The most popular and extensively used tensor decompositions is the Canonical Polyadic or CANDECOMP/PARAFAC decomposition, referred to as CP decomposition henceforth. Given a 3-mode tensor $\underline{\mathbf{X}}$ of dimension $\mathbb{R}^{I \times J \times K}$, and rank at most R can be written

$$\underline{\mathbf{X}} = \sum_{r=1}^{R} (a_r \odot b_r \odot c_r) \iff \underline{\mathbf{X}}(i,j,k) = \sum_{r=1}^{R} A(i,r)B(j,r)C(k,r)$$

$\forall\, i \in \{1, 2, \dots, I\}$, $j \in \{1, 2, \dots, J\}$, $k \in \{1, 2, \dots, K\}$ and $\mathbf{A} \in \mathbb{R}^{I \times R}, \mathbf{B} \in \mathbb{R}^{J \times R}$ and $\mathbf{C} \in \mathbb{R}^{K \times R}$. For tensor approximation, we adopted minimizing least square criteria as $\mathcal{L} \approx \min \frac{1}{2} ||\underline{\mathbf{X}} - \mathbf{A}(\mathbf{C} \odot \mathbf{B})^T||_F^2$ where $||\underline{\mathbf{X}}||_F^2$ is the sum of squares of its all elements and $||.||_F$ is *Frobenius* (norm). The CP model is nonconvex in $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$. We refer interested readers to popular surveys [9,13] on tensor decompositions and its applications for more details.

2.2 Problem Definition

Let us consider a social media network like Facebook, where a large number of users ($\approx 684K$) update information every single minute, and Twitter, where about $\approx 100K$ users tweet every minute[2]. Here, we have interactions arriving continuously at high velocity, where each interaction consists of User Id, Tag Ids, Device, and Location information etc. How can we capture such dynamic user interactions? How to identify concepts which can signify a potential newly emerging community, complete disappearance of interactions, or a merging of one or more communities to a single one? When using tensors to represent such dynamically evolving data, our problem falls under "streaming" or "online" tensor analysis. Decomposing streaming or online tensors is challenging task, and concept drift in incoming data makes the problem significantly more difficult, especially in applications where we care about characterizing the concepts in the data, in addition to merely approximating the streaming tensor adequately.

Before we conceptualize the problem that our paper deals with, we define certain terms which are necessary to set up the problem. Consider $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$ be two incremental batches of a streaming tensors of rank R and F respectively. Let $\underline{\mathbf{X}}$ be the initial tensor at time t_0 and $\underline{\mathbf{Y}}$ be the batch of the streaming tensor which arrives at time t_1 such as $t_1 > t_0$. The CP decomposition for these two tensors is given as follows:

$$\underline{\mathbf{X}} \approx \sum_{r=1}^{R} \mathbf{A}(:,r) \circ \mathbf{B}(:,r) \circ \mathbf{C}(:,r) \tag{1}$$

$$\underline{\mathbf{Y}} \approx \sum_{r=1}^{F} \mathbf{A}(:,r) \circ \mathbf{B}(:,r) \circ \mathbf{C}(:,r) \tag{2}$$

Concept: In case of tensors, we define *concept* as one latent component; a sum of R such components make up the tensor. In above equations tensor $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$ has R and F concepts respectively.

Concept Overlap: We define *concept overlap* as the set of latent concepts that are common or shared between two streaming CP decompositions. Consider Fig. 1 where R and F both are equal to three, which means both tensors $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$ have three concepts. Each concept of $\underline{\mathbf{X}}$ corresponds to each concept of $\underline{\mathbf{Y}}$. This means that there are three concepts that overlap between $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$. The minimum and maximum number of concept overlaps between two tensors can be zero and $\min(R, F)$ respectively. Thus, the value of concept overlap lies between 0 and $\min(R, F)$. In Sect. 3 we propose an algorithm for detecting such overlap.

$$0 \leq \text{Concept Overlap} \leq \min(R, F) \tag{3}$$

New Concept: If there exists a set of concepts which are not similar to any of the concepts already present in the most recent tensor batch, we call all such concepts in that set as *new concepts*. Consider Fig. 2(a), where $\underline{\mathbf{X}}$ has two

[2] https://mashable.com/2012/06/22/data-created-every-minute/.

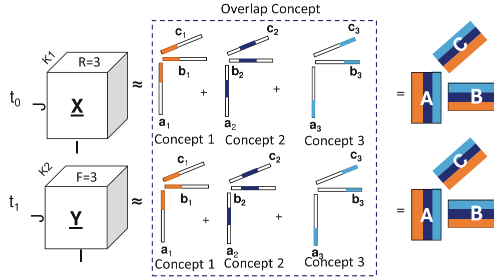


Fig. 1. Complete overlap of concepts

concepts ($R = 2$) and $\underline{\mathbf{Y}}$ has three concepts ($F = 3$). We see that at time t_1 tensor $\underline{\mathbf{Y}}$ batch has three concepts, out of which, two match with tensor $\underline{\mathbf{X}}$ concepts and one concept (namely concept 3) does not match with any concept of $\underline{\mathbf{X}}$. In this scenario we say that concept 1 and 2 are *overlapping concepts* and concept 3 is a *new concept*.

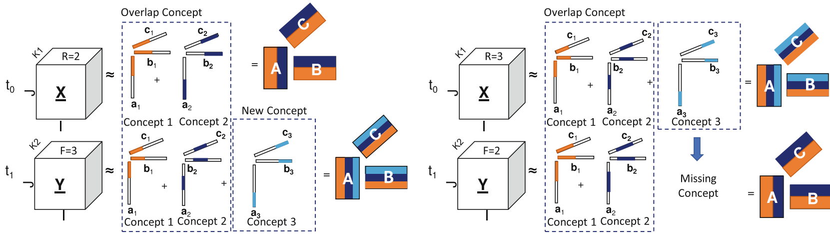


Fig. 2. (a) Concept appears (b) Concept disappears

Missing Concept: If there exists a set of concepts which was present at time t_0, but was missing at future time t_1, we call the concepts in the set *missing concepts*. For example, consider Fig. 2(b), at time t_0, the CP decomposition of $\underline{\mathbf{X}}$ has three concepts, and at time t_1 CP decomposition of $\underline{\mathbf{Y}}$ has two concepts. Two concepts of $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$ match with each other and one concept, present at t_0, is missing at t_1; we label that concept, as *missing concept*.

Running Rank: Running Rank (runningRank) at time t is defined as the total number of unique concepts (or latent components) seen until time t. Running Rank is different from tensor rank of a tensor batch. It may or may not be equal to rank of the current tensor batch. Consider Fig. 1, runningRank at time t_1 is three, since the total unique number of concepts seen until t_1 is three. Similarly runningRank of Fig. 2(b) at time t_1 is three, even though rank of $\underline{\mathbf{Y}}$ is two, since the number unique concepts seen until t_1 is three.

Let us assume rank of the initial tensor batch $\underline{\mathbf{X}}$ at time t_0 is R and rank of the subsequent tensor batch $\underline{\mathbf{Y}}$ at time t_1 is F. Then runningRank at time t_1 is sum of running rank at t_0 and number of new concepts discovered from t_0 to t_1. At time t_0 running rank is equal to initial rank of the tensor batch in this case R.

$$\text{runningRank}_{t_1} = \text{runningRank}_{t_0} + num(\text{newConcept})_{t_1 - t_0} \tag{4}$$

Concept Drift: Concept drift is usually defined in terms of supervised learning [3,14,15]. In [14], authors define concept drift in unsupervised learning as the change in probability distribution of a random variable over time. We define concept drift in the context of latent concepts, which is based on rank of the tensor batch. We first give an intuitive description of concept in terms of running rank, and then define concept drift.

Intuition: Consider running rank at time t_1 be runningRank_{t_1} and running at time t_2 be runningRank_{t_2}. If runningRank_{t_1} is not equal to runningRank_{t_2}, then there is a concept drift i.e. either a new concept has appeared, or a concept has disappeared. However, this definition does not capture every single case. Assume if runningRank_{t_1} is equal to runningRank_{t_2}. In this case, there is no drift only when there is a complete overlap. However there may be concept drift present even if runningRank_{t_1} is equal to runningRank_{t_2}, since a concept might disappear while runningRank remains the same.

Definition: Whenever a new concept appears, a concept disappears, or both from time $t1$ to $t2$, this phenomenon is defined as *concept drift*.

In a streaming tensor application, a tensor batch arrives at regular intervals of time. Before we decompose a tensor batch to get latent concepts, we need to know the rank of the tensor. Finding tensor rank is a hard problem [8] and it is beyond the scope of this paper. There has been considerable amount of work which approximates rank of a tensor [10,12]. In this paper we employ AutoTen [12] to compute a low rank of a tensor. As new advances in tensor rank estimation happen, our proposed method will also benefit.

Problem 1. **Given** (a) tensor $\underline{\mathbf{X}}$ of dimensions $I \times J \times K_1$ and rank R, (b) $\underline{\mathbf{Y}}$ of dimensions $I \times J \times K_2$ of rank F at time t_0 and t_1 respectively as shown in figure 3. Compute $\underline{\mathbf{X}}_{new}$ of dimension $I \times J \times (K_1 + K_2)$ of rank equal to runningRank at time t_1 as shown in equation (5) using factor matrices of $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$.

$$\underline{\mathbf{X}}_{new_{t_1}} \approx \sum_{r=1}^{\text{runningRank}} \mathbf{A}(:,r) \circ \mathbf{B}(:,r) \circ \mathbf{C}(:,r) \tag{5}$$

3 Proposed Method

Consider a social media application where thousands of connections are formed every second, for example, who follows whom or who interacts with whom. These connections formed can be viewed as forming communities. Over a period of time communities disappear, new communities appear or some communities re-appear after sometime. Number of communities at any given point of time is dynamic. There is no way of knowing what communities will appear or disappear in future. When this data stream is captured as a tensor, communities refer to latent concepts and appearing and disappearing of communities over a period of a time is referred to as concept drift. Here we need a dynamic way of figuring out number of communities in a tensor batch rather than assuming constant number of communities in all tensor batches.

To the best of our knowledge, there is no algorithmic approach that detects concept drift in streaming tensor decomposition. As we mentioned in Sect. 1, there has been considerable amount of work [6,11,16] which deals with streaming tensor data and applies batch decomposition on incoming slices and combine the results. But these methods don't take change of rank in consideration, which could reveal new latent concept in the data sets. Even if we know the rank (latent concept) of the complete tensor, the tensor batches of that tensor might not have same rank as the complete tensor.

In this paper we propose *SeekAndDestroy*, a streaming CP decomposition algorithm that does not assume rank is fixed. *SeekAndDestroy* detects the rank of every incoming batch in order to decompose it, and finally, updates the existing decomposition after detecting and alleviating concept drift, as defined in Sect. 2.

An integral part of *SeekAndDestroy* is detecting different concepts and identifying concept drift in streaming tensor. In order to do this successfully, we need to solve following problems:

P1: Finding the rank of a tensor batch.
P2: Finding New Concept, Concept Overlap and Missing Concept between two consecutive tensor batch decomposition.
P3: Updating the factor matrices to incorporate the new and missing concepts along with concept overlaps.

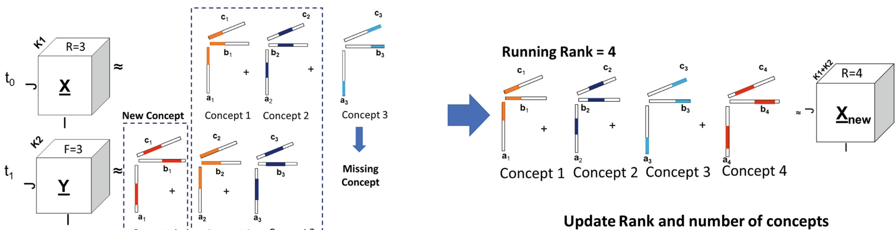


Fig. 3. Problem formulation

Finding Number of Latent Concepts: Finding the rank of the tensor is beyond the scope of this paper, thus we employ AutoTen [12]. Furthermore, in Sect. 4, we perform our experiments on synthetic data where we know the rank (and use that information as given to us by an "oracle") and repeat those experiments using AutoTen, comparing the error between them; the gap in quality signifies room for improvement that *SeekAndDestroy* will reap, if rank estimation is solved more accurately in the future.

Finding Concept Overlap: Given a rank of tensor batch, we compute its latent components using CP decomposition. Consider Fig. 3 as an example. At time t_1, the number of latent concepts we computed is represented by F, and we already had R components before new batch $\underline{\mathbf{Y}}$ arrived. In this scenario, there could be three possible cases: (1) $R = F$ (2) $R > F$ (3) $R < F$.

For each one of the cases mentioned above, there may be new concepts appear at t_1, or concepts disappear from t_0 to t_1, or there could be shared concepts between two decompositions. In Fig. 3. we see that, even though R is equal to F, we have one new concept, one missing concept and two shared/overlapping concepts. Now, at time t_1, we have four unique concepts, which means our runningRank at t_1 is four.

Algorithm 1. *SeekAndDestroy* for Detecting & Alleviating Concept Drift

Input: Tensor $\underline{\mathbf{X}}_{new}$ of size $I \times J \times K_{new}$, Factor matrices $\mathbf{A}_{old}, \mathbf{B}_{old}, \mathbf{C}_{old}$ of size $I \times R$, $J \times R$ and $K_{old} \times R$ respectively, runningRank, mode.

Output: Factor matrices $\mathbf{A}_{updated}, \mathbf{B}_{updated}, \mathbf{C}_{updated}$ of size $I \times$ runningRank, $J \times$ runningRank and $(K_{new} + K_{old}) \times$ runningRank, $\boldsymbol{\rho}$, runningRank.

1: $batchRank \leftarrow getRankAutoten(\underline{\mathbf{X}}_{new}, runningRank)$
2: $[\mathbf{A}, \mathbf{B}, \mathbf{C}, \boldsymbol{\lambda}] = \text{CP}(\underline{\mathbf{X}}_{new}, batchRank)$.
3: $\mathbf{colA}, \mathbf{colB}, \mathbf{colC} \leftarrow$ Compute Column Normalization of $\mathbf{A}, \mathbf{B}, \mathbf{C}$.
4: $\mathbf{normMatA}, \mathbf{normMatB}, \mathbf{normMatC} \leftarrow$ Absorb $\boldsymbol{\lambda}$ and Normalize $\mathbf{A}, \mathbf{B}, \mathbf{C}$.
5: $rhoVal \leftarrow colA\ .*\ colB\ .*\ colC$
6: $[\text{newConcept}, \text{conceptOverlap}, overlapConceptOld] \leftarrow$
$findConceptOverlap(\mathbf{A}_{old}, \mathbf{normMatA})$
7: **if** newConcept **then**
8: runningRank $\leftarrow$ runningRank + len(newConcept)
9: **Aupdated** $\leftarrow \left[\mathbf{A}_{old}\ \mathbf{normMatA}(:, \text{newConcept})\right]$
10: **Bupdated** $\leftarrow \left[\mathbf{B}_{old}\ \mathbf{normMatB}(:, \text{newConcept})\right]$
11: **Cupdated** $\leftarrow$ update $\mathbf{C}$ depending on the New Concept, Concept Overlap, overlapConceptOld indices and runningRank
12: **else**
13: **Aupdated** $\leftarrow \mathbf{A}_{old}$
14: **Bupdated** $\leftarrow \mathbf{B}_{old}$
15: **Cupdated** $\leftarrow$ update $\mathbf{C}$ depending on the Concept Overlap, overlapConceptOld indices and runningRank
16: **end if**
17: Update ρ depending on the New Concept and Concept Overlap indices
18: **if** newConcept or (len(newConcept) + len(conceptOverlap) < runningRank) **then**
19: Concept Drift Detected
20: **end if**

In order to discover which concepts are shared, new, or missing we use the *Cauchy-Schwarz inequality* which states for two vectors $\mathbf{a}$ and $\mathbf{b}$ we have $\mathbf{a}^T\mathbf{b} \leq ||\mathbf{a}||_2||\mathbf{b}||_2$. Algorithm 2 provides the general outline of technique used in finding concepts. It takes a column-normalized matrices $\mathbf{A_{old}}$ and $\mathbf{A_{batch}}$ of size $I \times R$ and $I \times batchRank$ respectively as input. We compute the dot product for all permutations of columns between two matrices, as shown below

$$\mathbf{A}^T_{\mathbf{old}}(:, col_i) \cdot \mathbf{A_{batch}}(:, col_j)$$

col_i and col_j are the respective columns. If the computed dot product is higher than the threshold value, the two concepts match, and we consider them as shared/overlapping between $\mathbf{A_{old}}$ and $\mathbf{A_{batch}}$. If the dot product between a column in $\mathbf{A_{batch}}$ and with all the columns in $\mathbf{A_{old}}$ has a value less than the threshold, we consider it as a new concept. This solves problem **P2**. In the experimental evaluation, we demonstrate the behavior of *SeekAndDestroy* with respect to that threshold.

SeekAndDestroy: This is our overall proposed algorithm, which detects concept drift between the two consecutive tensor batch decompositions, as illustrated in Algorithm 1 and updates the decomposition in a fashion robust to the drift. *SeekAndDestroy* takes factor matrices ($\mathbf{A_{old}}$, $\mathbf{B_{old}}$, $\mathbf{C_{old}}$) of previous tensor batch (say at time t_0), running rank at t_0($\mathbf{runningRank}_{t_0}$) and new tensor batch ($\underline{\mathbf{X}}_{new}$) (say at time t_1) as inputs. Subsequently, *SeekAndDestroy* computes the tensor rank for the batch (**batchRank**) for $\underline{\mathbf{X}}_{new}$ using AutoTen.

Using the estimated rank **batchRank**, *SeekAndDestroy* computes the CP decomposition of $\underline{\mathbf{X}}_{new}$, which returns factor matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$. We normalize the columns of A, B, C to unit ℓ_2 norm and we store the normalized matrices into **normMatA**, **normMatB**, and **normMatC**, as shown by lines 3–4 of Algorithm 1. Both $\mathbf{A_{old}}$ and normalized matrix $\mathbf{A}$ are passed to $findConceptOverlap$ function as described above. This returns the indexes of new concept and indexes of overlapping concepts from both matrices. Those indexes inform *SeekAndDestroy*, while updating the factor matrices, where to append the overlapped concepts. If there are new concepts, we update A and B factor matrices simply by adding new columns from normalized factor matrices of $\underline{\mathbf{X}}_{new}$ as shown in lines 9–10 of Algorithm 1. Furthermore, we update the running rank by adding number of new concept discovered to the previous running rank. If there is only overlapping concepts and no new concepts, then $\mathbf{A}$ and $\mathbf{B}$ factor matrices does not change.

Updating Factor Matrix C: In this paper, for simplicity of exposition, we are focusing on streaming data that are increasing only on one mode. However, our proposed method readily generalizes to cases where more than one modes grow over time.

In order to update the "evolving" factor matrix ($\mathbf{C}$ in our case), we use a different technique from the one used to update $\mathbf{A}$ and $\mathbf{B}$. If there is a new concept discovered in **normMatC** then

$$\mathbf{C}_{updated} = \begin{bmatrix} \mathbf{C}_{old} & zeroCol \\ zerosM & \mathbf{normMatC}(:, newConcept) \end{bmatrix} \tag{6}$$

where $zeroCol$ is of size $K_{old} \times len(newConcept)$, $zerosM$ is of size $K_{new} \times R$ and $\mathbf{C}_{updated}$ is of size $(K_{old} + K_{new}) \times \text{runningRank}$.

If there are overlapping concepts, then we update $\mathbf{C}$ accordingly as shown below; in this case $\mathbf{C}_{updated}$ is again of size $(K_{old} + K_{new}) \times \text{runningRank}$.

$$\mathbf{C}_{updated} = \begin{bmatrix} \mathbf{C}_{old}(:, overlapConceptOld) \\ \mathbf{normMatC}(:, conceptOverlap) \end{bmatrix} \tag{7}$$

If there are missing concepts we append an all-zeros matrix (column vector) to those indexes.

The Scaling Factor ρ: When we reconstruct the tensor from updated factor (normalized) matrices, we need a way to re-scale the columns of those factor matrices. In our approach we compute element wise product on normalized columns of factor matrices ($\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$) of $\underline{\mathbf{X}}_{new}$ as shown in line 5 of Algorithm 1. We use the same technique as the one used in updating C matrix, in order to match the values between two consecutive intervals, and we add this value to previously computed values. If it is a missing concept, we simply add zero to it. While reconstructing the tensor we take the average of vector over the number of batches received and we re-scale the components as follows

$$\underline{\mathbf{X}}_r = \sum_{r=1}^{\text{runningRank}} \rho_r \mathbf{A}_{\text{upd.}}(:, r) \circ \mathbf{B}_{\text{upd.}}(:, r) \circ \mathbf{C}_{\text{upd.}}(:, r).$$

4 Experimental Evaluation

We evaluate our algorithm on the following criteria:

Q1: Approximation Quality: We compare *SeekAndDestroy*'s reconstruction accuracy against state-of-the-art streaming baselines, in data that we generate synthetically so that we observe different instances of concept drift. In cases where *SeekAndDestroy* outperforms the baselines, we argue that this is due to the detection and alleviation of concept drift.

Q2: Concept Drift Detection Accuracy: We evaluate how effectively *SeekAndDestroy* is able to detect concept drift in synthetic cases, where we control the drift patterns.

Q3: Sensitivity Analysis: As shown in Sect. 3, *SeekAndDestroy* expects the matching threshold as a user input. Furthermore, its performance may depend on the selection of the batch size. Here, we experimentally evaluate *SeekAndDestroy*'s sensitivity along those axes.

Algorithm 2. Find Concept Overlap

Input: Factor matrices $\mathbf{A}_{old}$, $\mathbf{normMatA}$ of size $I \times R$, $I \times batchRank$ respectively.
Output: newConcept, conceptOverlap, overlapConceptOld
1: $THRESHOLD \leftarrow 0.6$
2: **if** $R == batchRank$ **then**
3: Generate all the permutations for [1:R]
4: **foreach** *permutation* **do**
| Compute dot product of $\mathbf{A}_{old}$ *and* $\mathbf{normMatA}(:, \mathbf{permutation})$
end
5: **else if** $R > batchRank$ **then**
6: Generate all the permutations(1:R, batchRank)
7: **foreach** *permutation* **do**
| Compute dot product of $\mathbf{A}_{old}(:, permutation)$ *and* $\mathbf{normMatA}$
end
8: **else if** $R < batchRank$ **then**
9: Generate all the permutations (1:batchRank, R)
10: **foreach** *permutation* **do**
| Compute dot product of $\mathbf{A}_{old}$ *and* $\mathbf{normMatA}(:, \mathbf{permutation})$
end
11: **end if**
12: Select the best permutation based on the maximum sum.
13: If dot product value of a column is less than threshold its a New Concept
14: If dot product value of a column is more than threshold then its a Concept Overlap.
15: Return column index's of New Concept and Concept Overlap for both matrices

Q4: Effectiveness on Real Data: In addition to measuring *SeekAndDestroy*'s performance in real data, we also evaluate its ability to identify useful and interpretable latent concepts in real data, which elude other streaming baselines.

4.1 Experimental Setup

We implemented our algorithm in Matlab using tensor toolbox library [2] and we evaluate our algorithm on both synthetic and real data. We use [12] method available in literature to find rank of incoming batch.

In order to have full control of the drift phenomena, we generate synthetic tensors with different ranks for every tensor batch, we control the batch rank of the tensor with factor matrix $\mathbf{C}$. Table 2 shows the specification of the datasets created. For instance dataset **SDS2** has an initial tensor batch whose tensor rank is 2 and last tensor batch whose tensor rank is 10 (full rank). The batches in between the initial and final tensor batch can have any rank between initial and final rank (in this case 2–10). The reason we assign the final batch rank as the full rank is to make sure the tensor created is not rank deficient. We make the synthetic tensor generator available as part of our code release.

In order for us to obtain robust estimates of performance, we require all experiments to either (1) run for 1000 iterations, or (2) the standard deviation converges to a second significant digit (whichever occurs first). For all reported results, we use the median and the standard deviation.

Table 2. Table of Datasets analyzed

DataSet	Dimension	Initial rank	Full rank	Batch size	Matching threshold
SDS1	100 × 100 × 100	2	5	10	0.6
SDS2			10		
SDS3	300 × 300 × 300	2	5	50	0.6
SDS4			10		
SDS5	500 × 500 × 500	2	5	100	0.6
SDS6			10		

4.2 Evaluation Metrics

We evaluate *SeekAndDestroy* and the baselines methods using *relative error*. Relative Error provides the measure of effectiveness of the computed tensor with respect to the original tensor and is defined as follows (lower is better):

$$RelativeError = \left(\frac{||\underline{\mathbf{X}}_{original} - \underline{\mathbf{X}}_{computed}||_F}{||\underline{\mathbf{X}}_{original}||_F}\right) \quad (8)$$

4.3 Baselines for Comparison

To evaluate our method, we compare *SeekAndDestroy* with two state-of-the-art streaming baselines: OnlineCP [16] and SamBaTen [6]. Both baselines assume that the rank remains fixed throughout the entire stream. When we evaluate the approximation accuracy of the baselines, we run two different versions of each method, with different input ranks: (1) *Initial Rank*, which is the rank of the initial batch, same as the one that *SeekAndDestroy* uses, and (2) *Full Rank*, which is the "oracle" rank of the full tensor, if we assume we could compute that in the beginning of the stream. Clearly, *Full Rank* offers a great advantage to the baselines since it provides information from the future.

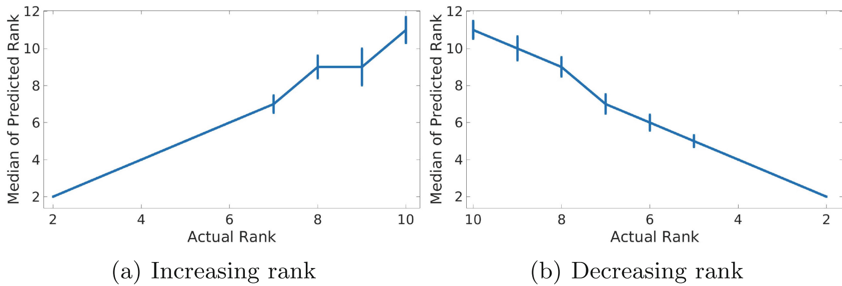
4.4 Q1: Approximation Quality

The first dimension that we evaluate is the approximation quality. More specifically, we evaluate whether *SeekAndDestroy* is able to achieve good approximation of the original tensor (in the form of low error) in case where concept drift is occurring in the stream. Table 3 contains the general results of *SeekAndDestroy*'s accuracy, as compared to the baselines. We observe that *SeekAndDestroy* outperforms the two baselines, in the pragmatic scenario where they are given the same starting rank as *SeekAndDestroy* (Initial Rank). In the non-realistic, "oracle" case, OnlineCP performs better than SamBaTen and *SeekAndDestroy*, however this case is a very advantageous lower bound on the error for OnlineCP.

Through extensive experimentation we made the following interesting observation: in the cases where most of the concepts in the stream appear in the beginning of the stream (e.g., in batches 2 and 3), *SeekAndDestroy* was able to further outperform the baselines. This is due to the fact that, if *SeekAndDestroy* has already "seen" most of the possible concepts early-on in the stream, it is more likely to correctly match concepts in later batches of the stream, since there already exists an almost-complete set of concepts to compare against. Indicatively, in this case *SeekAndDestroy* achieved 0.1176 ± 0.0305 where as OnlineCP achieved 0.1617 ± 0.0702.

4.5 Q2: Concept Drift Detection Accuracy

The second dimension along which we evaluate *SeekAndDestroy* is its ability to successfully detect concept drift. Figure 4 shows the rank discovered by *SeekAndDestroy* at every point of the stream, plotted against the actual rank. We observe that *SeekAndDestroy* is able to successfully identify changes in rank, which, as we have already argued, signify concept drift. Furthermore, Table 4(b) shows three example runs that demonstrate the concept drift detection accuracy.



(a) Increasing rank (b) Decreasing rank

Fig. 4. *SeekAndDestroy* is able to successfully detect concept drift, which is manifested as changes in the rank throughout the stream

Table 3. Approximation error for *SeekAndDestroy* and the baselines. *SeekAndDestroy* outperforms the baselines in the realistic case where all methods start with the same rank

DataSet	OnlineCP (Initial Rank)	OnlineCP (Full Rank)	SamBaTen (Initial Rank)	SamBaTen (Full Rank)	*SeekAndDestroy*
SDS1	0.2782 ± 0.0221	0.197 ± 0.086	$\mathbf{0.261 \pm 0.048}$	0.317 ± 0.058	0.283 ± 0.075
SDS2	0.2537 ± 0.0125	0.168 ± 0.507	$\mathbf{0.244 \pm 0.028}$	0.480 ± 0.051	0.253 ± 0.0412
SDS3	0.2731 ± 0.0207	0.205 ± 0.164	0.385 ± 0.021	0.445 ± 0.164	$\mathbf{0.266 \pm 0.081}$
SDS4	0.245 ± 0.013	0.171 ± 0.537	0.299 ± 0.045	0.402 ± 0.049	$\mathbf{0.221 \pm 0.0423}$
SDS5	0.2719 ± 0.0198	0.206 ± 0.022	0.559 ± 0.046	0.519 ± 0.0219	$\mathbf{0.256 \pm 0.105}$
SDS6	0.238 ± 0.013	0.171 ± 0.374	0.510 ± 0.036	0.547 ± 0.0276	$\mathbf{0.208 \pm 0.0433}$

4.6 Q3: Sensitivity Analysis

The results we have presented so far for *SeekAndDestroy* have used a matching threshold of 0.6. The threshold was chosen because it is intuitively larger than a 50% match, which is a reasonable matching threshold. In this experiment, we investigate the sensitivity of *SeekAndDestroy* to the matching threshold parameter. Table 4(a) shows exemplary approximation errors for thresholds of 0.4, 0.6, and 0.8. We observe that (1) the choice of threshold is fairly robust for values around 50%, and (2) the higher the threshold, the better the approximation, with threshold of 0.8 achieving the best performance.

Table 4. (a) Experimental results for error of approximation of incoming batch with different matching threshold values. Dataset SDS2 and SDS4 are of dimension $\mathbb{R}^{100\times100\times100}$ and $\mathbb{R}^{300\times300\times300}$, respectively. We see that the threshold is fairly robust around 0.5, and a threshold of 0.8 achieves the highest accuracy (b) Experimental results on SDS1 for error of approximation of incoming slices with known and predicted rank

Threshold	SDS2	SDS4
0.4	0.253±0.041	0.221 ± 0.042
0.6	0.253±0.041	0.221 ± 0.042
0.8	0.101 ±0.040	0.033 ± 0.011

Running Rank	Actual Rank	Predicted Rank	**Approx. Error** Actual Rank	Predicted Rank
6	[2,4,3,4,3,3,5,3,3,5]	[2,4,3,4,3,3,5,3,3,6]	0.185	0.194
6	[2,4,3,4,3,3,5,3,3,5]	[2,4,3,4,3,3,5,3,3,6]	0.185	0.197
7	[2,4,3,4,3,3,5,3,3,5]	[2,4,3,5,3,3,6,3,3,6]	0.185	0.278

Table 5. Evaluation on Real dataset

Running rank	Predicted full rank	Batch size	Approximation error		
			SeekAndDestroy	SambaTen	OnlineCP
7 ± 0.88	4 ± 0.57	22	**0.68 ± 0.002**	0.759 ± 0.059	0.941 ± 0.001

4.7 Q4: Effectiveness on Real Data

To evaluate effectiveness of our method on real data, we use the Enron time-evolving communication graph dataset [1]. Our hypothesis is that in such complex real data, there should exists concept drift in streaming tensor decomposition. In order to validate that hypothesis, we compare the approximation error incurred by *SeekAndDestroy* against the one incurred by the baselines, shown in Table 5. We observe that the approximation error of *SeekAndDestroy* is lower than the two baselines. Since the main difference between *SeekAndDestroy* and the baselines is that *SeekAndDestroy* takes concept drift into consideration, and strives to alleviate its effects, this result (1) provides further evidence that there exists concept drift in the Enron data, and (2) demonstrates *SeekAndDestroy*'s effectiveness on real data.

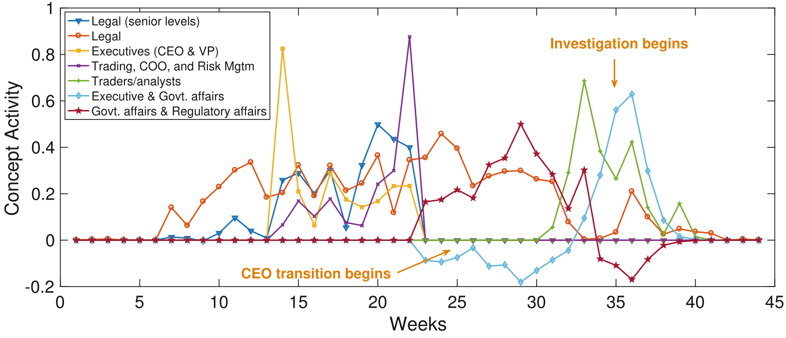


Fig. 5. Timeline of concepts discovered in Enron

The final rank for Enron as computed by *SeekAndDestroy* was 7, indicating the existence of 7 time-evolving communities in the dataset, as shown in Fig. 5. This number of communities is higher than what previous tensor-based analysis has uncovered [1,5]. However, analyzing the (static) graph using a highly-cited non-tensor based method [4], we were able to detect 7 communities, therefore *SeekAndDestroy* may be discovering subtle communities that have eluded previous tensor analysis. In order to verify that, we delved deeper into the communities and we plot their temporal evolution (taken from matrix $\mathbf{C}$) along with their annotations (when inspecting the top-5 senders and receivers within each community). Indeed, a subset of the communities discovered matches with the ones already known in the literature [1,5]. Additionally, *SeekAndDestroy* was able to discover community #3, which refers to a group of executives, including the CEO. This community appears to be active up until the point that the CEO transition begins, after which point it dies out. This behavior is indicative of concept drift, and *SeekAndDestroy* was able to successfully discover and extract it.

5 Related Work

Tensor Decomposition: Tensor decomposition techniques are widely used for static data. With the explosion of big data, data grows at a rapid speed and an extensive study required on the online tensor decomposition problem. Sidiropoulos [11] introduced two well-known PARAFAC based methods namely RLST (recursive least square) and SDT (simultaneous diagonalization tracking) to address the online 3-mode tensor decomposition. Zhou et al. [16] proposed OnlineCP for accelerating online factorization that can track the decompositions when new updates arrived for N-mode tensors. Gujral et al. [6] proposed Sampling-based Batch Incremental Tensor Decomposition algorithm which updates online computation of CP/PARAFAC and performs all computations in the reduced summary space. However, no prior work addresses concept drift.

Concept Drift: The survey paper [14] provides the qualitative definitions of characterizing the drifts on data stream models. To the best of our knowledge, however, this is the first work to discuss concept drift in tensor decomposition.

6 Conclusions

In this paper we introduce the notion of "concept drift" in streaming tensors. and provide *SeekAndDestroy*, an algorithm which detects and alleviates concept drift it without making any assumption on the rank of the tensor. *SeekAndDestroy* outperforms other state-of-the-art methods when the rank is unknown and is effective in detecting concept drift. Finally, we apply *SeekAndDestroy* on a real time-evolving dataset, discovering novel drifting concepts.

Acknowledgements. Research was supported by the Department of the Navy, Naval Engineering Education Consortium under award no. N00174-17-1-0005, the National Science Foundation EAGER Grant no. 1746031, and by an Adobe Data Science Research Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding parties.

References

1. Bader, B., Harshman, R., Kolda, T.: Analysis of latent relationships in semantic graphs using DEDICOM. In: Workshop for Algorithms on Modern Massive Data Sets (2006)
2. Bader, B.W., Kolda, T.G., et al.: MATLAB Tensor Toolbox Version 2.6, February 2015. http://www.sandia.gov/~tgkolda/TensorToolbox/
3. Bifet, A., Gama, J., Pechenizkiy, M., Zliobaite, I.: Handling concept drift: importance, challenges and solutions. PAKDD-2011 Tutorial, Shenzhen, China (2011)
4. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. J. Stat. Mech. Theory Exp. **2008**(10), P10008 (2008)
5. Papalexakis, E.E., Faloutsos, C., Sidiropoulos, N.D.: ParCube: sparse parallelizable tensor decompositions. In: Flach, P.A., De Bie, T., Cristianini, N. (eds.) ECML PKDD 2012. LNCS, vol. 7523, pp. 521–536. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33460-3_39
6. Gujral, E., Pasricha, R., Papalexakis, E.E.: SamBaTen: sampling-based batch incremental tensor decomposition. arXiv preprint arXiv:1709.00668 (2017)
7. Harshman, R.: Foundations of the PARAFAC procedure: models and conditions for an "explanatory" multimodal factor analysis (1970)
8. Håstad, J.: Tensor rank is NP-complete. J. Algorithms **11**(4), 644–654 (1990)
9. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM Rev. **51**(3), 455–500 (2009)
10. Mørup, M., Hansen, L.K.: Automatic relevance determination for multi-way models. J. Chemom. **23**(7–8), 352–363 (2009)
11. Nion, D., Sidiropoulos, N.: Adaptive algorithms to track the PARAFAC decomposition of a third-order tensor. Signal Process. **57**(6), 2299–2310 (2009)

12. Papalexakis, E.E.: Automatic unsupervised tensor mining with quality assessment. In: Proceedings of the 2016 SIAM International Conference on Data Mining, pp. 711–719. SIAM (2016)
13. Papalexakis, E.E., Faloutsos, C., Sidiropoulos, N.D.: Tensors for data mining and data fusion: models, applications, and scalable algorithms. ACM Trans. Intell. Syst. Technol. (TIST) **8**(2), 16 (2017)
14. Webb, G.I., Hyde, R., Cao, H., Nguyen, H.L., Petitjean, F.: Characterizing concept drift. Data Min. Knowl. Disc. **30**(4), 964–994 (2016)
15. Webb, G.I., Lee, L.K., Petitjean, F., Goethals, B.: Understanding concept drift. CoRR abs/1704.00362 (2017). http://arxiv.org/abs/1704.00362
16. Zhou, S., Vinh, N.X., Bailey, J., Jia, Y., Davidson, I.: Accelerating online CP decompositions for higher order tensors. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1375–1384. ACM (2016)