Incremental Pattern Discovery on Streams, Graphs and Tensors

Jimeng Sun

CMU-CS-07-149

December 10, 2007

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Christos Faloutsos, Chair
Tom Mitchell
Hui Zhang
David Steier, PWC
Philip Yu, IBM

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This material is based upon work supported by the National Science Foundation under Grants No. IIS-0326322 IIS-0534205 and under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work is also partially supported by the Pennsylvania Infrastructure Technology Alliance (PITA), an IBM Faculty Award, a Yahoo Research Alliance Gift, with additional funding from Intel, NTT and Hewlett-Packard. 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 National Science Foundation, or other funding parties.

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 National Science Foundation, DARPA, or other funding parties.

Keywords: Data mining, Stream mining, Incremental learning, Clustering, Tensor

To my parents, Lanru Zhang and Qingguo Sun, who brought me to life.
To my wife, Huiming Qu who brings me to joy.

Abstract

Incremental pattern discovery targets streaming applications where the data continuously arrive incrementally. The questions are how to find patterns (main trends) incrementally; or how to efficiently update the old patterns when new data arrive; or how to utilize the patterns to solve other problems such as anomaly detection?

As examples, 1) a sensor network monitors a large number of distributed streams (such as temperature and humidity); 2) network forensics monitor the Internet communication patterns to identify attacks; 3) cluster monitoring examines the system behaviors of a number of machines for potential failures; 4) social network analysis monitors a dynamic graph for communities and abnormal individuals; 5) financial fraud detection tries to find fraudulent activities from a large number of transactions.

We first investigate a powerful data model, *tensor stream* (TS), where there is one tensor per timestamp. To capture diverse data formats, we have a zero-order TS for a single time-series (e.g., the stock price for Google over time), a first-order TS for multiple time-series (sensor measurement streams), a second-order TS for a matrix (graphs), and a high-order TS for a multi-array (Internet communication network, source-destination-port). Second, we develop different online algorithms on TS: 1) the centralized and distributed SPIRIT for mining a *1st-order* TS, as well as its extensions for local correlation function and privacy preservation; 2) the compact matrix decomposition (CMD) and GraphScope for a *2nd-order* TS; 3) the dynamic tensor analysis (DTA), streaming tensor analysis (STA) and window-based tensor analysis (WTA) for a *high-order* TS. All the techniques are extensively evaluated for real applications such as network forensics, cluster monitoring.

In particular, this CMD achieves orders of magnitude improvements in space and time over the previous state of the art, and identifies interesting anomalies. GraphScope detects interesting communities and change-points on several time-evolving graphs such as Enron email graph and another network traffic flow graph. DTA, STA and WTA are all online methods for higher-order data that scale well with time, provide fundamental tradeoffs with each other, which have also been applied to a number of applications, such as social network community tracking, anomaly detection in data centers and network traffic monitoring.

Acknowledgments

I am extremely grateful to my adviser, Christos Faloutsos. Throughout my entire PhD journey, Christos has been a great role model as a researcher, a mentor, a friend and a human being. As a researcher, his enthusiasm and dedication has been of great inspiration to me. As a mentor, he was always there to listen and to share his genuine and thoughtful advice on all things that matter. For all of us who had the privilege to work with Christos, his charisma and warmth always gave us the great strength to move forward.

I would also like to thank all my outstanding committee members, Tom Mitchell, David Steier, Philip Yu and Hui Zhang: It was Tom who taught me about machine learning. David and Philip gave me extremely valuable industrial perspective on real-world data mining problems. Several inspiring conversations with Hui also helped me tremendously towards the end of my PhD studies.

I thank all my collaborators, Spiros Papadimitriou, Dacheng Tao, Feifei Li, Yinglian Xie, Deepay Chakrabarti, Tamara Kolda, Michael Mahoney, Petros Drineas, Evan Hoke, John Strunk, Greg Ganger. Without their invaluable discussion and feedback, my journey would not have been completed.

Thanks to all the people who shared my life in Pittsburgh: Jiaxin Fu, Juchang Hua, Yanhua Hu, Zhenzhen Kou, Yan Li, Min Luo, Minglong Shao, Sichen Sun, Chenyu Wu, Hong Yan, Hua Zhong. I remember all the joy and pain in every Steelers' game we watched together. I also want to thank my office mates Yinglian Xie and Joseph Gonzalez (Wean 5117 is the best!). Finally, my special gratitude to the School of Computer Science at Carnegie Mellon, I feel extremely fortunate to have studied at such a distinguished place.

I would also like to thank Dimitris Papadias at the Hong Kong University of Science and Technology. It was Dimitris who brought me into the research world and beyond. His unique character has had a significant impact on my life.

Most of all, I am grateful to my parents, Lanru Zhang and Qingguo Sun, and my wife,

Huiming Qu, without whose love and support none of this would be possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

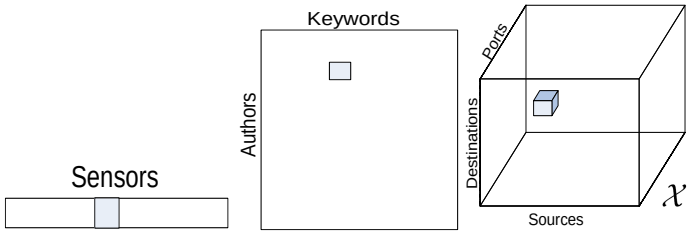
Incremental pattern discovery targets at streaming applications where data continuously arrive incrementally. In this thesis, we want to answer the following questions: How to find patterns (main trends) incrementally? How to efficiently update the old patterns when new data arrive? How to utilize the patterns to solve other problems such as anomaly detection and clustering?

Some examples include:

- *Sensor Networks* [115] monitor different measurements (such as temperature and humidity) from a large number of distributed sensors. The task is to monitor correlations among different sensors over time and identify anomalies.
- *Cluster Management* [73] monitors the many metrics (such as CPU and memory utilization, disk space, number of processes, etc) of a group of machines. The task is to find main trends, to identify anomalies or potential failures as well as to compress the signals for storage.
- *Social Network Analysis* [119] observes an evolving network of social activities (such as citation, telecommunication and corporate email networks). The task is to find communities and anomalies, and to monitor them over time.
- *Network Forensics* [119] monitors Internet communication in the form of source, destination, port, time, number of packets, etc. Again, the task is to summarize the communication patterns and to identify the potential attacks and anomalies.
- *Financial Fraud Detection* [18] examines transactional activities of a company over time and tries to identify the abnormal/fraudulent behaviors.

1.1 Data Model

To deal with the diversity of data, we introduce an expressive data model *tensor* from multi-linear analysis [43]. For the Sensor Networks example, we have one measurement (e.g., temperature) from each sensor every timestamp, which forms a high dimensional vector (first order tensor) as shown in Figure 1.1(a). For the Social Network Analysis, we have authors publishing papers, which forms graphs represented by matrices (second order tensors). For the network forensics example, the 3rd order tensor for a given time period has three modes: source, destination and port, which can be viewed as a 3D data cube (see Figure 1.1(c)). An entry (i, j, k) in that tensor (like the small blue cube in Figure 1.1(c)) has the number of packets from the corresponding source i to the destination j through port k, during the given time period. Figure 1.1 illustrates three tensor examples where the blue region indicates a single element in the tensor such as a measurement from a single sensor in (a), the number of papers that an author wrote on a given keyword in (b), the number of packets sent from a source IP to a destination IP through a certain port in (c).



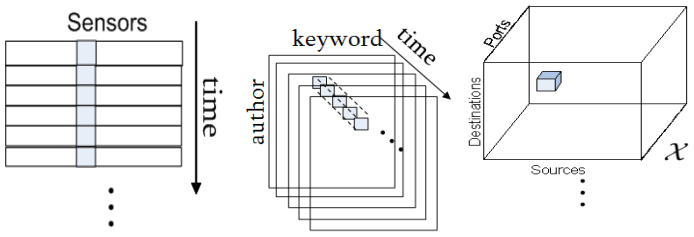
(a) 1st-order (b) 2nd-order (c) 3rd-order

Figure 1.1: Tensor examples

Focusing on incremental applications, we propose the *tensor stream* (TS) which is an unbounded sequence of tensors. The streaming aspect comes from the fact that new tensors are arriving continuously. Figure 1.2 illustrates the corresponding examples of tensor streams.

1.2 Incremental Pattern Discovery

Incremental Pattern discovery is an online summarization process. In this thesis, we focus on incrementally identifying low-rank structures of the data as the *patterns* and monitor them over time. In another words, we consider the incremental pattern discovery as an



(a) 1st-order (b) 2nd-order (c) 3rd-order

Figure 1.2: Tensor Stream examples

incremental dimensionality reduction process. In this sense, the following terms are interchangeable: summaries, patterns, hidden variables, low-dimensional representations, core tensors.

Let us illustrate the main idea through the network forensics application. In this example, the hourly communication data are represented by high dimensional (3rd order) tensors, which are summarized as low dimensional (3rd order) core tensors in a different space specified by the projection matrices (see Figure 1.3).

Moreover, the projection matrices capture the overall correlations or hidden variables along three aspects (modes): *source, destination* and *port*. For example, the *Source projection* characterizes the client correlations; the *Destination projection* summarizes the server correlations; the *Port projection* monitors the port traffic correlations. The projection matrices are dynamically monitored over time.

Furthermore, the core tensor indicates the association across different aspects. More specifically, if there are 3 source hidden variables, 5 destination hidden variables and 6 port hidden variables, the core tensor is a 3-by-5-by-6 3D array, in which the values correspond to the level of association across three different aspects active at different time. More details are covered in Section 4.2.4

The notation of projection matrices can be very general. In particular, we explore three different subspace construction strategies of the projection matrices:

1. *orthonormal projection*, which forms orthogonal matrices based on the data such as SPIRIT described in Section 2.2 [108, 116] and Dynamic and Streaming Tensor Analysis (DTA/STA) in Section 4 [119];

2. *example-based projection*, which judiciously select examples from data to form the subspace in Section 3.2 [120];

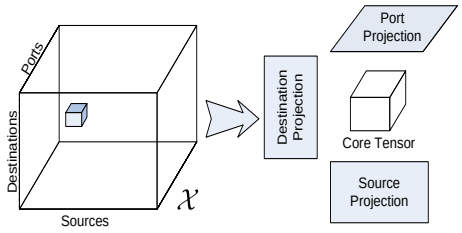


Figure 1.3: Pattern discovery on a 3rd order tensor

3. *clustering based projection*, which are indicator variable vectors based on cluster assignments in Section 3.3 [118].

The incremental aspect of the algorithms arrives from the fact that model needs to be constantly updated. More specifically, the problems we study are as the follows: Given a stream of tensors $\mathcal{X}_1 \dots \mathcal{X}_n$, how to compress them incrementally and efficiently? How to find patterns and anomalies? We plan to address two aspects of incremental pattern discovery:

- *Incremental update*: We want to update the old model efficiently, when a new tensor arrives. The key is to avoid redundant computation and storage.
- *Model efficiency*: We want an efficient method in terms of computational cost and storage consumption. The goal is to achieve linear computation and storage requirement to the update size.

Why is it useful?:

The results from incremental pattern discovery can be used for many important tasks:

- *Compression*: The core tensor captures most of the information of the original data but in a much lower dimension.
- *Anomaly detection*: From the core tensor, we can approximate the original tensor and compute the reconstruction error. A large reconstruction error often indicates an anomaly.
- *Clustering*: We can often cluster the original data based the projection (see Section 3.3 and Section 4.2.4 for details). The idea is closely related to co-clustering [47] and latent semantic indexing (LSI) [44].

More importantly, all these tasks can be done incrementally which is essential to many monitoring applications as listed in the beginning of this section.

1.3 Contributions and Outline

Thesis statement *Incremental and efficient summarization of streaming data through a general and concise presentation enables many real applications in diverse domains.*

Following this philosophy, we present a suite of tools for summarizing complex streaming data incrementally.

In Chapter 2, we present SPIRIT, a stream mining algorithm for first order tensor streams. The main contribution is that SPIRIT provides an efficient and fully automatic approach to summarize high dimensional streams (first order tensor streams) using a variant of incremental PCA. Several applications immediately follow such as the InteMon system [73](see Section 2.2), which does compression and anomaly detection on performance sensors of a data center. Using SPIRIT as a building block, several other techniques are introduced such as the distributed stream monitoring in Section 2.3 [116], local correlation tracking in Section 2.4 [109] and privacy preservation on streams in Section 2.5 [92].

Chapter 3 presents two different techniques for analyzing large graphs (i.e., second-order tensors) and time-evolving graphs (second-order tensor streams). Section 3.2 presents Compact Matrix Decomposition (CMD) [120] and its application on network forensics. The key contribution of CMD is to efficiently summarize a matrix (i.e., a graph or a second-order tensor) as a linear combination of judiciously sampled columns and rows, i.e., the projection matrix is the actual "example" columns and rows of the original data. Unlike SVD/PCA, CMD preserves the data sparsity (therefore, fast and space efficient) and provides easy interpretation based on the original data. Finally, we illustrate how to apply CMD on time-evolving graphs. Section 3.3 presents Graphscope, a parameter-free approach on summarizing and clustering time-evolving graphs (i.e., multiple graphs indexed by time, a second-order tensor stream). Graphscope frames the graph mining problem as a compression problem, and therefore, utilizes the Minimal Description Length (MDL) principle to automatically determine all the parameters. More specifically, Graphscope involves two online procedures to summarize all graphs. 1) It automatically groups the consecutive graphs/matrices into time segments which minimizes the compression objective (code length). 2) Within a time segment, it partitions the matrices into homogeneous sub-matrices and determines the correct number of clusters which minimizes the code length. Graphscope successfully identifies interesting groups and time segments on several large social networks such as Enron email dataset.

Chapter 4 focuses on analyzing general tensor streams, under which three techniques are presented, namely, Dynamic Tensor Analysis (DTA), Streaming Tensor Analysis (STA) [119] and Window-based Tensor Analysis (WTA) [117]. The main contribution is the design and applications of several high-order streaming algorithms. In particular, DTA and WTA utilizes the incremental construction of covariance matrices, while STA generalizes the SPIRIT algorithm to high-order. The applications include Multi-way latent semantic indexing (high-order clustering) and anomaly detection. We also present several case studies such as environmental sensor monitoring, data center monitoring and network forensics.

In summary, this thesis concerns two aspects of the incremental pattern discovery: First, *tensor order* varies from one (Section 2 [108, 116]), two (Section 3 [120, 118]) to higher order (Section 4 [119, 117]). The complexity increases as we are going to higher order tensors. We studied the fundamental trade-off in designing mining algorithms on tensors through both theoretical analysis and empirical evaluation. Second, we exploit different *subspace formation* strategies under the tensor stream framework. In particular, we studied orthogonal projection, example-based projection and clustering based projection.

Chapter 2

Stream mining

"How to summarize a set of numeric streams efficiently? How does the summarization enable other applications such as anomaly detection, similarity search and privacy preservation?"

Data streams or first-order tensor stream have received tremendous attention in various communities such as theory, databases, data mining, network systems, due to many important applications:

- **Network forensics**: As the Internet becomes most prevalent media affecting people's daily life, network security has also been a key problem for both the research community and the IT industry. Especially, as the network traffic grows with new technology such as P2P file sharing, Voice over IP, and IPTV, it becomes a grand challenge to monitor network traffic flows for intrusion and other abnormal behaviors in a streaming fashion.
- **Environmental sensor monitoring**: In the water distribution system, chemical sensors can be deployed to monitor various substances in the water such as Chlorine concentration level. All the sensor measurements need to be analyzed in real time in order to identify possible pollution, attacks and pipe leaks in the system.
- **Data center monitoring**: Modern data centers are awash in monitoring data. Nearly every application, host, and network device exports statistics that could (and should) be monitored. Additionally, many of the infrastructure components such as UPSes, power distribution units, and computer room air conditioners (CRACs) provide data about the status of the computing environment. Being able to monitor and respond to abnormal conditions is critical for maintaining a high availability installation.

- **Financial applications**: Various financial systems process billions of transactions every day including stock trades, PoS transactions, online shopping and banking information. Huge flows of financial data are streamed into the systems and require real-time validation and monitoring.

Most of the streams are numeric in nature. In particular, a single stream can be viewed as a sequence of numbers (scalars) with an increasing size, i.e., a zero-order tensor stream. Similarly, multiple streams are represented as a sequence of vectors with an increasing size, i.e., a first-order tensor stream.

How to efficiently summarize data streams incrementally?(Section 2.2 and [108]) This is a fundamental problem for stream mining. Note that due to the streaming nature, the two important requirements are posed as the follows: 1) *Efficient:* the process should incur small (bounded) computation and storage requirement over time; 2) *Incremental:* the process should reflect the time order in the data, i.e., time-dependency on the models.

We address this fundamental problem with our SPIRIT algorithm in Section 2.2. SPIRIT can incrementally find correlations and hidden variables, which summarize the key trends in multiple streams. It can do this quickly, with no buffering of stream values and without comparing pairs of streams. The discovered trends can also be used to immediately spot potential anomalies and to do efficient forecasting. Our experimental evaluation and case studies show that SPIRIT can incrementally capture correlations and discover trends, efficiently and effectively. We also developed InteMon system [73] which utilizes the SPIRIT algorithm to monitor multiple performance sensor measurements (e.g., CPU and memory utilization) and look for anomalies in a data center.

Following SPIRIT, several important problems can be answered:

How to perform stream mining in a distributed environment?(Section 2.3 and [116]) This is a difficult but very practical problem that needs to be solved in order to deploy SPIRIT into large scale systems.

We developed the distributed SPIRIT algorithm [116] for doing distributed system monitoring. More specifically, given several distributed groups of streams, we want to: (1) incrementally find local patterns within a single group, (2) efficiently obtain global patterns across groups, and more importantly, (3) efficiently do that in real time while limiting shared information across groups. Distributed SPIRIT adopts a hierarchical method addressing these problems. It first monitors local patterns within each group using SPIRIT and further summarizes all local patterns from different groups into global patterns. The global patterns are leveraged to improve and refine the local patterns, in a simple and elegant way. Moreover, it requires only a single pass over the data, without any buffering, and limits information sharing and communication across groups. The experimental case

studies and evaluation confirm that distributed SPIRIT can perform hierarchical correlation detection efficiently and effectively.

How to define and compute correlations for streams? (Section 2.4 and [109]) The notion of correlation (or, similarity) is important, since it allows us to discover groups of objects with similar behavior and, consequently, discover potential anomalies which may be revealed by a change in correlation. Unlike static time series, finding correlation between a pair of streams is a non-trivial problem for two reasons: First, data characteristics may change over time. In this case, a single, static correlation score for the entire streams is less useful. Instead, it is desirable to have a notion of correlation that also evolves with time and tracks the changing relationships. Second, the correlation of data streams often exhibit strong but fairly complex, non-linear correlations. Traditional measures, such as the widely used cross-correlation coefficient (or, Pearson coefficient), are less effective in capturing these complex relationships.

To approach this problem, we compare the local auto-covariance of each stream which generalizes the notion of linear cross-correlation. In this way, it is possible to concisely capture a wide variety of local patterns or trends. Our method produces a general similarity score, which evolves over time, and accurately reflects the changing relationships. Finally, it can also be estimated incrementally using the SPIRIT algorithm. We demonstrate its usefulness, robustness and efficiency on a wide range of real datasets.

How to preserve privacy in data streams?(Section 2.5 and [92]) There has been an increasing concern regarding privacy breaches, especially those involving sensitive personal data of individuals. Meanwhile, unprecedented massive data from various sources are providing us with a great opportunity for data mining. Unfortunately, the privacy requirement and data mining pose conflicting expectations from data publishing. How to balance data utility and privacy guarantee becomes a key problem. Prior works on privacy preservation are discussed in subsection 2.1.5. Guaranteeing data privacy is especially challenging in the case of data streams for two reasons: 1) Performance requirement: The continuous arrival of new data prohibits storage of the entire stream for analysis, rendering the current offline algorithms inapplicable. 2) Time evolution: Data streams are usually evolving, and correlations and autocorrelations can change over time. These characteristics make most offline algorithms for static data inappropriate,

In Section 2.5 , we show that it is possible to leverage the SPIRIT algorithm to track the correlation and autocorrelation and add noise which maximally preserves privacy, in the sense that it is very hard to remove. Our techniques achieve much better results than previous static, global approaches, while requiring limited processing time and memory. We provide both a mathematical analysis and experimental evaluation on real data to validate the correctness, efficiency, and effectiveness of our algorithms.

We first present some background and related work in section 2.1. We then present *SPIRIT*, an online algorithm for summarizing multiple numeric streams in section 2.2. After that, we show how to extend the centralized SPIRIT algorithm to work in the distributed environment in section 2.3. Finally, we illustrate two applications of SPIRIT on local correlation computation and privacy preservation in section 2.4 and 2.5.

2.1 Stream related work

In the following, we use lowercase bold letters for column vectors ($\mathbf{u}, \mathbf{v}, \ldots$) and uppercase bold for matrices ($\mathbf{U}, \mathbf{V}, \ldots$). For a general matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, m and n are the number of rows and columns, respectively. The inner product of two vectors is denoted by $\mathbf{x}^\mathsf{T}\mathbf{y}$ and the outer product by $\mathbf{x} \circ \mathbf{y} \equiv \mathbf{x}\mathbf{y}^\mathsf{T}$. The Euclidean norm of $\mathbf{x}$ is $\|\mathbf{x}\|$.

2.1.1 Singular value decomposition

Theorem 2.1 (Singular Value Decomposition (SVD))**.** *If $\mathbf{A}$ is a m-by-n matrix with rank r, then there exist orthogonal matrices $\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_r] \in \mathbb{R}^{m \times r}$ and $\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_r] \in \mathbb{R}^{n \times r}$ such that $\mathbf{U}^\mathsf{T}\mathbf{A}\mathbf{V} = diag(\sigma_1, \ldots, \sigma_r) \in \mathbb{R}^{r \times r}$ where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r \geq 0$.*

Proof. See [64] for the proof. □

The columns of $\mathbf{U}$ and $\mathbf{V}$ are called left and right singular vectors, respectively; $\sigma_1, \ldots, \sigma_r$ are singular values as shown in Figure 2.1. In addition to represent SVD as a matrix decomposition as $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathsf{T}$, it can also be viewed as the sum of a number of rank-one matrices. This is called "Spectral decomposition", i.e., $\mathbf{A} = \sum_i \sigma_i \mathbf{u}_i {\mathbf{v}_i}^\mathsf{T}$ where $\mathbf{u}_i {\mathbf{v}_i}^\mathsf{T}$ is a rank-one matrix.

SVD reveals latent connections in the data through singular vectors. Example applications of SVD include latent semantic indexing (LSI) [44] in information retrieval and HITS algorithm for web ranking [85].

2.1.2 Principal component analysis

PCA, as shown in Figure 2.2, finds the best linear projections of a set of high dimensional points to minimize least-squares cost. More formally, given n points represented as vectors $\mathbf{x}_i|_{i=1}^n \in \mathbb{R}^N$ in an N dimensional space, PCA computes n points $\mathbf{y}_i|_{i=1}^n \in \mathbb{R}^R$ ($R \ll N$) in

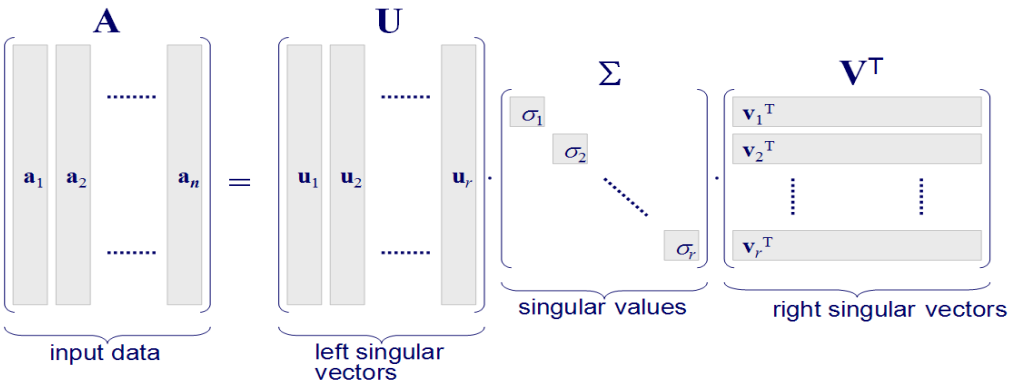


Figure 2.1: Singular value decomposition (SVD)

a lower dimensional space and the projection matrix $\mathbf{U} \in \mathbb{R}^{N \times R}$ such that the least-squares cost $e = \sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{U}\mathbf{y}_i\|_2^2$ is minimized.

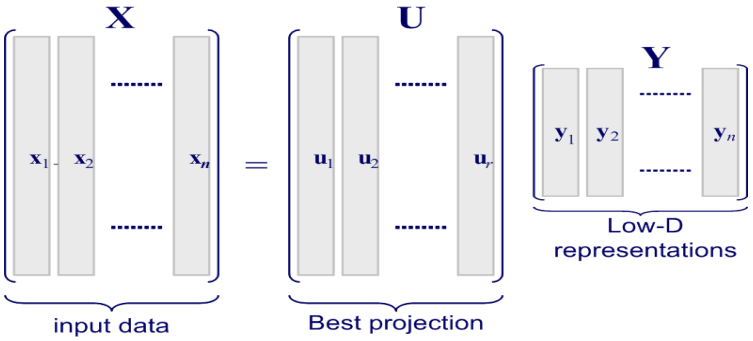


Figure 2.2: PCA projects the N-D vectors $\mathbf{x}_i|_{i=1}^{n}$ into R-D vectors $\mathbf{y}_i|_{i=1}^{n}$ and $\mathbf{U}$ is the projection matrix.

The solution of PCA can be computed efficiently by diagonalizing the covariance matrix of $\mathbf{x}_i|_{i=1}^{n}$. Alternatively, if the rows are zero mean, then PCA is computed by the Singular Value Decomposition (SVD): if the SVD of $\mathbf{X}$ is $\mathbf{X} = \mathbf{U}_{svd}\mathbf{\Sigma}_{svd}{\mathbf{V}_{svd}}^{T}$, then our $\mathbf{U} = \mathbf{U}_{svd}$ and $\mathbf{Y} = \mathbf{\Sigma}_{svd}{\mathbf{V}_{svd}}^{\mathsf{T}}$. See [80] for more details on PCA.

2.1.3 Covariance and auto-covariance

The covariance of two random variables X, Y is defined as $\mathrm{Cov}[X,Y] = \mathrm{E}[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])]$. If $X_1, X_2, \ldots, X_m$ is a group of m random variables, their covariance matrix $\mathbf{C} \in \mathbb{R}^{m \times m}$ is the symmetric matrix defined by $c_{ij} := \mathrm{Cov}[X_i, X_j]$, for $1 \leq i, j \leq m$. If $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ is a collection of n observations $\mathbf{x}_i \equiv [x_{i,1}, x_{i,2}, \ldots, x_{i,m}]^\mathsf{T}$ of all m variables, the sample covariance estimate[1] is defined as

$$\hat{\mathbf{C}} := \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i \circ \mathbf{x}_i.$$

where $\mathbf{x}_i \circ \mathbf{x}_i$ is the outer product of the i-th observation.

In the context of a time series process $\{X_t\}_{t \in \mathbb{N}}$, we are interested in the relationship between values at different times. To that end, the auto-covariance is defined as $\gamma_{t,t'} := \mathrm{Cov}[X_t, X_{t'}] = \mathrm{E}[X_t X_{t'}]$, where the last equality follows from the zero-mean assumption. By definition, $\gamma_{t,t'} = \gamma_{t',t}$.

Spectral decomposition Any real symmetric matrix is always equivalent to a diagonal matrix, in the following sense.

Theorem 2.2 (Diagonalization)**.** *If $\mathbf{A} \in \mathbb{R}^{n \times n}$ is a symmetric, real matrix, then it is always possible to find a column-orthonormal matrix $\mathbf{U} \in \mathbb{R}^{n \times n}$ and a diagonal matrix $\mathbf{\Lambda} \in \mathbb{R}^{n \times n}$, such that $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\mathsf{T}$.*

Proof. See [64]. □

Thus, given any vector $\mathbf{x}$, we can write $\mathbf{U}^\mathsf{T}(\mathbf{A}\mathbf{x}) = \mathbf{\Lambda}(\mathbf{U}^\mathsf{T}\mathbf{x})$, where pre-multiplication by $\mathbf{U}^\mathsf{T}$ amounts to a change of coordinates. Intuitively, if we use the coordinate system defined by $\mathbf{U}$, then $\mathbf{A}\mathbf{x}$ can be calculated by simply scaling each coordinate independently of all the rest (i.e., multiplying by the diagonal matrix $\mathbf{\Lambda}$).

Given any symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, we will denote its eigenvectors by $\mathbf{u}_i(\mathbf{A})$ and the corresponding eigenvalues by $\lambda_i(\mathbf{A})$, in order of decreasing magnitude, where $1 \leq i \leq n$. The matrix $\mathbf{U}_k(\mathbf{A})$ has the first k eigenvectors as its columns, where $1 \leq k \leq n$.

[1] The unbiased estimator uses $n - 1$ instead of n, but this constant factor does not affect the eigendecomposition.

2.1.4 Stream mining

There is a large body of work on streams, which we loosely classify in two groups.

Data stream management systems (DSMS). We include this very broad category for completeness. DSMS include Aurora [4], Stream [101], Telegraph [30] and Gigascope [40]. The common hypothesis is that (i) massive data streams come into the system at a very fast rate, and (ii) near real-time monitoring and analysis of incoming data streams is required. The new challenges have made researchers re-think many parts of traditional DBMS design in the streaming context, especially on query processing using correlated attributes [46], scheduling [16, 23], load shedding [124, 41] and memory requirements [14].

In addition to system-building efforts, a number of approximation techniques have been studied in the context of streams, such as sampling [21], sketches [50, 38, 61], exponential histograms [42], and wavelets [68]. The main goal of these methods is to estimate a global aggregate (e.g. sum, count, average) over a window of size w on the recent data. The methods usually have resource requirements that are sublinear with respect to w. Most focus on a single stream.

The emphasis in this line of work is to support traditional SQL queries on streams. None of them tries to find patterns, nor to do forecasting.

Data mining on streams. Researchers have started to redesign traditional data mining algorithms for data streams. Much of the work has focused on finding interesting patterns in a single stream, but multiple streams have also attracted significant interest. Ganti et al. [62] propose a generic framework for stream mining. Guha et al. [69] propose a one-pass k-median clustering algorithm. Domingos and Hulten [51] construct a decision tree online, by passing over the data only once. Recently, [75, 129] address the problem of finding patterns over concept drifting streams. Papadimitriou et al. [106] proposed a method to find patterns in a single stream, using wavelets. More recently, Palpanas et al. [104] consider approximation of time-series with *amnesic* functions. They propose novel techniques suitable for streaming, and applicable to a wide range of user-specified approximating functions.

Keogh et al. [83] propose parameter-free methods for classic data mining tasks (i.e., clustering, anomaly detection, classification), based on compression. Lin et al. [93] perform clustering on different levels of wavelet coefficients of multiple time series. Both approaches require having all the data in advance.

CluStream [7] is a flexible clustering framework with online and offline components. The online component extends micro-cluster information [137] by incorporating exponentially-

sized sliding windows while coalescing micro-cluster summaries. Actual clusters are found by the offline component. StatStream [140] uses the DFT to summarize streams within a finite window and then compute the highest pairwise correlations among all pairs of streams, at each timestamp. BRAID [112] addresses the problem of discovering lag correlations among multiple streams. The focus is on time and space efficient methods for finding the earliest and highest peak in the cross-correlation functions between all pairs of streams. Neither CluStream, StatStream nor BRAID explicitly focus on discovering hidden variables.

Guha et al. [67] improve on discovering correlations, by first doing dimensionality reduction with random projections, and then periodically computing the SVD. However, the method incurs high overhead because of the SVD re-computation and it can not easily handle missing values. MUSCLES [136] is exactly designed to do forecasting (thus it could handle missing values). However, it can not find hidden variables and it scales poorly for a large number of streams n, since it requires at least quadratic space and time, or expensive reorganizations (*selective MUSCLES*).

Finally, a number of the above methods usually require choosing a sliding window size, which typically translates to buffer space requirements. Our approach does not require any sliding windows and does not need to buffer *any* of the stream data.

In conclusion, none of the above methods simultaneously satisfy the requirements in the introduction: "any-time" streaming operation, scalability on the number of streams, adaptivity, and full automation.

Distributed data mining. Most works on distributed data mining focus on extending classic (centralized) data mining algorithms into distributed environment, such as association rules mining [34], frequent item sets [98]. Web is a popular distributed environment. Several techniques are proposed specifically for that, for example, distributed top-k query [17] and Bayes-net mining on web [32]. But our focus is on finding numeric patterns, which is different.

2.1.5 Privacy preservation

Privacy preserving data mining was first proposed in [10] and [9]. This work paved the road for an expanding field, and various privacy preservation techniques have been proposed since. These methods apply to the traditional relational data model, and can be classified as data perturbation [10, 9, 96, 31, 59, 11], k-anonymity [84, 121, 8, 99, 131] and secure multi-party computation [94, 126]. Our work focuses on privacy preservation in the context of the randomized data perturbation approach and we will focus on discussing

related work in this area.

Data perturbation can be further classified in two groups: retention replacement perturbation [11, 59] and data value perturbation [10, 96, 31]. For each element in a column j, the retention replacement perturbation retains this element with probability p_j and with probability $1-p_j$ replaces it with an item generated from the replacing pdf on this column. This approach works for categorical data as well, and it has been applied to privacy preserving association mining [59]. Our work focuses on numerical data value perturbation. Initial solutions in this category, [10, 9], proposed adding random i.i.d. noise to the original data and showed that, with knowledge of the noise distribution, the distribution of the original data can be estimated from the perturbed data, and aggregate values are preserved. In [81, 74] the authors pointed out that adding random i.i.d. noise is not optimal for privacy preservation. They showed how to reconstruct the original data (individual data values) using Spectral Filtering (SF) or the equivalent PCA method. The main conclusion is that random noise should be distributed along the principal components of the original data, so that linear reconstruction methods cannot separate the noise from the original data. Motivated by this observation and in similar spirit, [31] proposed the random rotation technique for privacy preserving classification and [96] proposed data perturbation based on random projection. The work of [58] discussed a method to quantify the privacy breach for privacy preserving algorithms, namely $\alpha - \beta$ analysis or γ-amplification. The basic idea is that, on the perturbed data, the adversaries' knowledge measured by their confidence about a given property of the original data should not be increased by more than a certain amount. The work in [13] considered the problem of setting the perturbation parameters while maintaining γ-amplification.

All these techniques have been developed for the traditional relational data model. There is no prior work on privacy preservation on data streams, except the work on private search over data streams [103, 20]. However, the goal there is to protect the privacy of the query over the data stream, not of the data stream itself. Finally, our data perturbation techniques rely on PCA for data streams with respect to both correlations and autocorrelations. Streaming PCA and eigen-space tracking of correlations (but not autocorrelation) among multiple data streams has been studied in [108, 67].

2.2 SPIRIT: Multiple Stream Mining

How to efficiently summarize data streams incrementally? To address this question, we introduce *SPIRIT - Streaming Pattern dIscoveRy in multIple sTreams* and its applications. In short, SPIRIT satisfies the following requirements:

- It is *streaming*, i.e., it is incremental, scalable. It requires very little memory and processing time per time tick. In fact, both are independent of the stream length t.
- It scales *linearly* with the number of streams n, not quadratically. This may seem counter-intuitive, because the naïve method to spot correlations across n streams examines all $O(n^2)$ pairs.
- It is *adaptive*, and fully *automatic*. It dynamically detects changes (both gradual, as well as sudden) in the input streams, and automatically determines the number k of hidden variables.

The correlations and hidden variables we discover have multiple uses. They provide a succinct summary to the user, they can help to do fast forecasting and detect outliers, and they facilitate interpolations and handling of missing values, as we discuss later.

2.2.1 Tracking Correlations

In summary, SPIRIT does the following:

- Given n streams, it produces a value $x_{t,j}$, for every stream $1 \leq j \leq n$ and for every time-tick $t = 1, 2, \ldots$.
- It adapts the number k of *hidden variables* necessary to explain/summarize the main trends in the collection.
- It adapts the *participation weights* $w_{i,j}$ of the j-th stream on the i-th hidden variable ($1 \leq j \leq n$ and $1 \leq i \leq k$), so as to produce an accurate summary of the stream collection.
- It monitors the hidden variables $y_{t,i}$, for $1 \leq i \leq k$.
- It keeps updating all the above efficiently.

More precisely, SPIRIT operates on the column-vectors of observed stream values $\mathbf{x}_t \equiv [x_{t,1}, \ldots, x_{t,n}]^T \in \mathbb{R}^n$ and continually updates the participation weights $w_{i,j}$. The *participation weight vector* $\mathbf{w}_i$ for the i-th stream is $\mathbf{w}_i := [w_{i,1} \cdots w_{i,n}]^T \in \mathbb{R}^n$. The hidden variables $\mathbf{y}_t \equiv [y_{t,1}, \ldots, y_{t,k}]^T \in \mathbb{R}^k$ are the projections of $\mathbf{x}_t$ onto each $\mathbf{w}_i$, over time, i.e.,

$$y_{t,i} := w_{i,1}x_{t,1} + w_{i,2}x_{t,2} + \cdots + w_{i,n}x_{t,n},$$

Or in matrix form,

$$\mathbf{y}_t := \mathbf{W}^\mathsf{T}\mathbf{x}_t$$

SPIRIT also adapts the number k of hidden variables necessary to capture most of the information. The adaptation is performed so that the approximation achieves a desired mean-square error. In particular, let $\tilde{\mathbf{x}}_t = [\tilde{x}_{t,1} \cdots \tilde{x}_{t,n}]^T \in \mathbb{R}^n$ be the *reconstruction* of $\mathbf{x}_t$, based on the weights and hidden variables, defined by

$$\tilde{x}_{t,j} := w_{1,j}y_{t,1} + w_{2,j}y_{t,2} + \cdots + w_{k,j}y_{t,k},$$

or in matrix form,

$$\tilde{\mathbf{x}}_t = \mathbf{W}\mathbf{y}_t$$

Definition 2.1 (SPIRIT tracking)**.** *SPIRIT updates the participation weights $w_{i,j}$ so as to guarantee that the reconstruction error $\|\tilde{\mathbf{x}}_t - \mathbf{x}_t\|^2$* over time *is predictably small.*

If we assume that the $\mathbf{x}_t$ are drawn according to some distribution that does not change over time (i.e., under *stationarity* assumptions), then the weight vectors $\mathbf{w}_i$ converge to the principal directions. However, even if there are non-stationarities in the data (i.e., gradual drift), in practice we can deal with these very effectively, as we explain later.

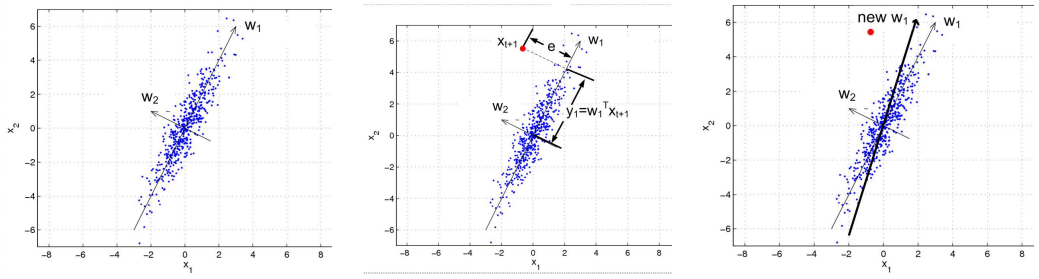
An additional complication is that we often have missing values, for several reasons: either failure of the system, or delayed arrival of some measurements. For example, the sensor network may get overloaded and fail to report some of the chlorine measurements in time or some sensor may temporarily black-out. At the very least, we want to continue processing the rest of the measurements.

Tracking the hidden variables

The first step is, for a given k, to incrementally update the k participation weight vectors $\mathbf{w}_i$, $1 \leq i \leq k$, so as to summarize the original streams with only a few numbers (the hidden variables). In Section 2.2.1, we describe the complete method, which also adapts k.

For the moment, assume that the number of hidden variables k is given. Furthermore, our goal is to minimize the average reconstruction error $\sum_t \|\tilde{\mathbf{x}}_t - \mathbf{x}_t\|^2$. In this case, the desired weight vectors $\mathbf{w}_i, 1 \leq i \leq k$ are the principal directions and it turns out that we can estimate them incrementally.

We use an algorithm based on adaptive filtering techniques [133, 71], which have been tried and tested in practice, performing well in a variety of settings and applications (e.g., image compression and signal tracking for antenna arrays). We experimented with



(a) Original $\mathbf{w}_1$ (b) Update process (c) Resulting $\mathbf{w}_1$

Figure 2.3: Illustration of updating $\mathbf{w}_1$ when a new point $\mathbf{x}_{t+1}$ arrives.

several alternatives [102, 48] and found this particular method to have the best properties for our setting: it is very efficient in terms of computational and memory requirements, while converging quickly, with no special parameters to tune. The main idea behind the algorithm is to read in the new values $\mathbf{x}_{t+1} \equiv [x_{(t+1),1}, \ldots, x_{(t+1),n}]^T$ from the n streams at time $t+1$, and perform three steps:

1. Compute the hidden variables $y'_{t+1,i}, 1 \leq i \leq k$, based on the *current* weights $\mathbf{w}_i, 1 \leq i \leq k$, by projecting $\mathbf{x}_{t+1}$ onto these.
2. Estimate the reconstruction error ($\mathbf{e}_i$ below) and the energy, based on the $y'_{t+1,i}$ values.
3. Update the estimates of $\mathbf{w}_i, 1 \leq i \leq k$ and output the *actual* hidden variables $y_{t+1,i}$ for time $t+1$.

To illustrate this, Figure 2.3(b) shows the $\mathbf{e}_1$ and y_1 when the new data $\mathbf{x}_{t+1}$ enter the system. Intuitively, the goal is to adaptively update $\mathbf{w}_i$ so that it quickly converges to the "truth." In particular, we want to update $\mathbf{w}_i$ more when $\mathbf{e}_i$ is large. However, the magnitude of the update should also take into account the past data currently "captured" by $\mathbf{w}_i$. For this reason, the update is inversely proportional to the current *energy* $E_{t,i}$ of the i-th hidden variable, which is $E_{t,i} := \frac{1}{t}\sum_{\tau=1}^{t} y_{\tau,i}^2$. Figure 2.3(c) shows $\mathbf{w}_1$ after the update for $\mathbf{x}_{t+1}$.

The *forgetting factor* λ will be discussed in Section 2.2.1 (for now, assume $\lambda = 1$). For each i, $d_i = tE_{t,i}$ and $\acute{\mathbf{x}}_i$ is the component of $\mathbf{x}_{t+1}$ in the orthogonal complement of the space spanned by the updated estimates $\mathbf{w}_{i'}, 1 \leq i' < i$ of the participation weights. The vectors $\mathbf{w}_i, 1 \leq i \leq k$ are in order of importance (more precisely, in order of decreasing eigenvalue or energy). It can be shown that, under stationarity assumptions, these $\mathbf{w}_i$ in these equations converge to the true principal directions[133].

Algorithm 2.1: TRACKW

1 Initialize the k participation weight vector $\mathbf{w}_i$ to unit vectors $\mathbf{w}_1 = [10\cdots0]^T$, $\mathbf{w}_2 = [010\cdots0]^T$, etc.
2 Initialize d_i ($i = 1, ...k$) to a small positive value.
3 **foreach** *new vector* $\mathbf{x}_{t+1}$ **do**
4 Initialize $\acute{\mathbf{x}}_1 := \mathbf{x}_{t+1}$
5 **for** $1 \leq i \leq k$ **do**
6 $y_i := \mathbf{w}_i^T \acute{\mathbf{x}}_i$ /* $y_{t+1,i}$ = projection onto $\mathbf{w}_i$ */
7 $d_i \leftarrow \lambda d_i + y_i^2$ /* energy $\propto i$-th eigenval. of $\mathbf{X}_t^T \mathbf{X}_t$ */
8 $\mathbf{e}_i := \acute{\mathbf{x}}_i - y_i \mathbf{w}_i$ /* error, $\mathbf{e}_i \perp \mathbf{w}_i$ */
9 $\mathbf{w}_i \leftarrow \mathbf{w}_i + \frac{1}{d_i} y_i \mathbf{e}_i$ /* update PC estimate */
10 $\acute{\mathbf{x}}_{i+1} := \acute{\mathbf{x}}_i - y_i \mathbf{w}_i$ /* repeat with remainder of $\mathbf{x}_t$ */
11 orthogonalize $\mathbf{w}_i, 1 \leq i \leq k$ by fixing $\mathbf{w}_1$

Line 11 of the algorithm TrackW is to maintain the orthogonality of all the participation weight vectors $\mathbf{w}_i, 1 \leq i \leq k$.

Complexity. We only need to keep the k weight vectors $\mathbf{w}_i$ ($1 \leq i \leq k$), each n-dimensional. Thus the total cost is $O(nk)$, both in terms of time and of space. The update cost does not depend on t. This is a tremendous gain, compared to the usual PCA computation cost of $O(tn^2)$.

Detecting the number of hidden variables

In practice, we do not know the number k of hidden variables. We propose to estimate k on the fly, so that we maintain a high percentage f_E of the *energy* E_t. Energy thresholding is a common method to determine how many principal components are needed [80]. Formally, the energy E_t (at time t) of the sequence of $\mathbf{x}_t$ is defined as

$$E_t := \frac{1}{t} \sum_{\tau=1}^{t} \|\mathbf{x}_\tau\|^2 = \frac{1}{t} \sum_{\tau=1}^{t} \sum_{i=1}^{n} x_{\tau,i}^2.$$

Similarly, the energy $\tilde{E}_t$ of the reconstruction $\tilde{\mathbf{x}}$ is defined as

$$\tilde{E}_t := \frac{1}{t} \sum_{\tau=1}^{t} \|\tilde{\mathbf{x}}_\tau\|^2.$$

Lemma 2.1. *Assuming the* $\mathbf{w}_i, 1 \leq i \leq k$ *are orthonormal, we have*

$$\tilde{E}_t = \frac{1}{t} \sum_{\tau=1}^{t} \|\mathbf{y}_\tau\|^2 = \frac{t-1}{t} \tilde{E}_{t-1} + \frac{1}{t} \|\mathbf{y}_t\|^2.$$

Proof. If the $\mathbf{w}_i, 1 \leq i \leq k$ are orthonormal, then it follows easily that $\|\tilde{\mathbf{x}}_\tau\|^2 = \|y_{\tau,1}\mathbf{w}_1 + \cdots + y_{\tau,k}\mathbf{w}_k\|^2 = y_{\tau,1}^2\|\mathbf{w}_1\|^2 + \cdots + y_{\tau,k}^2\|\mathbf{w}_k\|^2 = y_{\tau,1}^2 + \cdots + y_{\tau,k}^2 = \|\mathbf{y}_\tau\|^2$ (Pythagorean theorem and normality). The result follows by summing over τ. □

.

From the user's perspective, we have a low-energy and a high-energy threshold, f_E and F_E, respectively. We keep enough hidden variables k, so the retained energy is within the range $[f_E \cdot E_t, F_E \cdot E_t]$. Whenever we get outside these bounds, we increase or decrease k. In more detail, the steps are:

1. Estimate the full energy E_{t+1}, incrementally, from the sum of squares of $x_{\tau,i}$.

2. Estimate the energy $\tilde{E}_{(k)}$ of the k hidden variables.

3. Possibly, adjust k. We introduce a new hidden variable (update $k \leftarrow k + 1$) if the current hidden variables maintain too little energy, i.e., $\tilde{E}_{(k)} < f_E E$. We drop a hidden variable (update $k \leftarrow k - 1$), if the maintained energy is too high, i.e., $\tilde{E}_{(k)} > F_E E$.

The energy thresholds f_E and F_E are chosen according to recommendations in the literature [80]. We use a lower energy threshold $f_E = 0.95$ and an upper energy threshold $F_E = 0.98$. Thus, the reconstruction $\tilde{\mathbf{x}}_t$ retains between 95% and 98% of the energy of $\mathbf{x}_t$.

The following lemma proves that the above algorithm guarantees the relative reconstruction error is within the specified interval $[f_E, F_E]$.

Lemma 2.2. *The relative squared error of the reconstruction satisfies*

$$1 - F_E \leq \frac{\sum_{\tau=1}^{t} \|\tilde{\mathbf{x}}_\tau - \mathbf{x}_\tau\|^2}{\sum_t \|\mathbf{x}_\tau\|^2} \leq 1 - f_E.$$

Proof. From the orthogonality of $\mathbf{x}_\tau$ and the complement $\tilde{\mathbf{x}}_\tau - \mathbf{x}_\tau$ we have $\|\tilde{\mathbf{x}}_\tau - \mathbf{x}_\tau\|^2 = \|\mathbf{x}_\tau\|^2 - \|\tilde{\mathbf{x}}_\tau\|^2 = \|\mathbf{x}_\tau\|^2 - \|\mathbf{y}_\tau\|^2$ (by Lemma 2.1). The result follows by summing over τ and from the definitions of E and $\tilde{E}$. □

Finally, in Section 2.2.4 we demonstrate that the incremental weight estimates are extremely close to the principal directions computed with offline PCA.

Algorithm 2.2: SPIRIT

Initialize $k \leftarrow 1$ and the total energy estimates of $\mathbf{x}_t$ and $\tilde{\mathbf{x}}_t$ per time tick to $E \leftarrow 0$ and $\tilde{E}_1 \leftarrow 0$.

foreach *new vector* $\mathbf{x}_{t+1}$ **do**

  Update $\mathbf{w}_i$, for $1 \leq i \leq k$ (step 1, TRACKW)

  Update the estimates (for $1 \leq i \leq k$)

$$E \leftarrow \frac{(t-1)E + \|\mathbf{x}_t\|^2}{t} \quad \text{and} \quad \tilde{E}_i \leftarrow \frac{(t-1)\tilde{E}_i + y_{t,i}^2}{t}.$$

  Let the estimate of retained energy be

$$\tilde{E}_{(k)} := \sum_{i=1}^{k} \tilde{E}_i.$$

  if $\tilde{E}_{(k)} < f_E E$ **then** Start estimating $\mathbf{w}_{k+1}$ (initializing as in TRACKW)

  if $\tilde{E}_{(k)} > F_E E$ **then** Discard $\mathbf{w}_k$ and $\tilde{E}_k$ and decrease $k \leftarrow k-1$

Exponential forgetting

We can adapt to more recent behavior by using an exponential forgetting factor $0 < \lambda < 1$. This allows us to follow trend drifts over time. We use the same λ for the estimation of both $\mathbf{w}_i$ as well as the AR models (see Section 2.2.2). However, we also have to properly keep track of the energy, discounting it with the same rate, i.e., the update at each step is:

$$E \leftarrow \frac{\lambda(t-1)E + \|\mathbf{x}_t\|^2}{t} \quad \text{and} \quad \tilde{E}_i \leftarrow \frac{\lambda(t-1)\tilde{E}_i + y_{t,i}^2}{t}.$$

Typical choices are $0.96 \leq \lambda \leq 0.98$ [71]. As long as the values of $\mathbf{x}_t$ do not vary wildly, the exact value of λ is not crucial. We use $\lambda = 0.96$ throughout. A value of $\lambda = 1$ makes sense when we know that the sequence is stationary (rarely true in practice, as most sequences gradually drift). Note that the value of λ does not affect the computation cost of our method. In this sense, an exponential forgetting factor is more appealing than a sliding window, as the latter has explicit buffering requirements.

2.2.2 Applications

We show how we can exploit the correlations and hidden variables discovered by SPIRIT to do (a) forecasting, (b) missing value estimation, (c) summarization of the large number of streams into a small, manageable number of hidden variables, and (d) outlier detection.

Forecasting and missing values

The hidden variables $\mathbf{y}_t$ give us a much more compact representation of the "raw" variables $\mathbf{x}_t$, with guarantees of high reconstruction accuracy (in terms of relative squared error, which is less than $1 - f_E$). When our streams exhibit correlations, as we often expect to be the case, the number k of the hidden variables is much smaller than the number n of streams. Therefore, we can apply *any* forecasting algorithm to the vector of hidden variables $\mathbf{y}_t$, instead of the raw data vector $\mathbf{x}_t$. This reduces the time and space complexity by orders of magnitude, because typical forecasting methods are quadratic or worse on the number of variables.

In particular, we fit the forecasting model on the $\mathbf{y}_t$ instead of $\mathbf{x}_t$. The model provides an estimate $\hat{\mathbf{y}}_{t+1} = f(\mathbf{y}_t)$ and we can use this to get an estimate for

$$\hat{\mathbf{x}}_{t+1} := \hat{y}_{t+1,1}\mathbf{w}_1[t] + \cdots + \hat{y}_{t+1,1}\mathbf{w}_k[t],$$

using the weight estimates $\mathbf{w}_i[t]$ from the previous time tick t. We chose auto-regression for its intuitiveness and simplicity, but any online method can be used.

Correlations. Since the principal directions are orthogonal ($\mathbf{w}_i \perp \mathbf{w}_j, i \neq j$), the components of $\mathbf{y}_t$ are *by construction uncorrelated*—the correlations have already been captured by the $\mathbf{w}_i, 1 \leq i \leq k$. We can take advantage of this de-correlation to reduce forecasting complexity. In particular for auto-regression, we found that one AR model per hidden variable provides results comparable to multivariate AR but much better in speed.

Auto-regression. Space complexity for multivariate AR (e.g., MUSCLES [136]) is $O(n^3\ell^2)$, where ℓ is the auto-regression window length. For AR per stream (ignoring correlations), it is $O(n\ell^2)$. However, for SPIRIT, we need $O(kn)$ space for the $\mathbf{w}_i$ and, with one AR model per y_i, the total space complexity is $O(kn+k\ell^2)$. As published, MUSCLES requires space that grows cubically with respect to the number of streams n. We believe it can be made to work with quadratic space, but this is still prohibitive. Both AR per stream and SPIRIT require space that grows linearly with respect to n, but in SPIRIT k is typically very small ($k \ll n$) and, in practice, SPIRIT requires less memory and time per update than AR per stream. More importantly, a single, independent AR model per stream cannot capture *any* correlations, whereas SPIRIT indirectly exploits the correlations present *within* a time tick.

Missing values. When we have a forecasting model, we can use the forecast based on $\mathbf{x}_{t-1}$ to estimate missing values in $\mathbf{x}_t$. We then use these estimated missing values to update the weight estimates, as well as the forecasting models. Forecast-based estimation of missing values is the most time-efficient choice and gives very good results.

Dataset	n	k	Description
Chlorine	166	2	Chlorine concentrations from EPANET.
Critter	8	1–2	Temperature sensor measurements.
Motes	54	2–4	Light sensor measurements.

Table 2.1: Description of datasets

2.2.3 Experimental case-study

In this section we present case studies on real and realistic datasets to demonstrate the effectiveness of our approach in discovering the underlying correlations among streams. In particular, we show that:

- We capture the appropriate number of hidden variables. As the streams evolve, we capture these changes in real-time [115] and adapt the number of hidden variables k and the weights $\mathbf{w}_i$.
- We capture the essential behavior with very few hidden variables and small reconstruction error.
- We successfully deal with missing values.
- We can use the discovered correlations to perform good forecasting, with *much* fewer resources.
- We can easily spot outliers.
- Processing time per stream is constant.

Section 2.2.4 elaborates on performance and accuracy. Next we describe each of the datasets and the mining results.

Chlorine concentrations

The Chlorine dataset was generated by EPANET 2.0[2] that accurately simulates the hydraulic and chemical phenomena within drinking water distribution systems. Given a network as the input, EPANET tracks the flow of water in each pipe, the pressure at each node, the height of water in each tank, and the concentration of a chemical species throughout the network, during a simulation period comprised of multiple timestamps. We

[2] http://www.epa.gov/ORD/NRMRL/wswrd/epanet.html

monitor the chlorine concentration level at all the 166 junctions in the network shown in Figure 2.4(a), for 4310 timestamps during 15 days (one time tick every five minutes). The data was generated by using the input network with the demand patterns, pressures, flows specified at each node.

Data characteristics. The two key features are:

- A clear global periodic pattern (daily cycle, dominating residential demand pattern). Chlorine concentrations reflect this, with few exceptions.
- A slight time shift across different junctions, which is due to the time it takes for fresh water to flow down the pipes from the reservoirs.

Thus, most streams exhibit the same sinusoidal-like pattern, except with gradual phase shifts as we go further away from the reservoir.

Results of SPIRIT. SPIRIT can successfully summarize the data using just two numbers (hidden variables) per time tick, as opposed to the original 166 numbers. Figure 2.4(a) shows the reconstruction for four of the sensors (out of 166). Only two hidden variables give very good reconstruction.

Interpretation. The two hidden variables (Figure 2.4(b)) reflect the two key dataset characteristics:

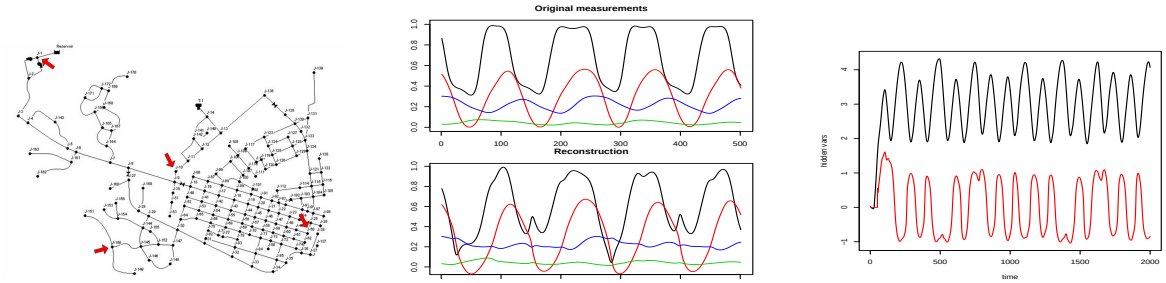
- The first hidden variable captures the global, periodic pattern.
- The second one also follows a very similar periodic pattern, but with a slight "phase shift." It turns out that the two hidden variables together are sufficient to express (via a linear combination) any other time series with an arbitrary "phase shift."

Light measurements

Description. The `Motes` dataset consists of light intensity measurements collected using Berkeley Mote sensors, at several different locations in a lab (see Figure 2.5), over a period of a month.

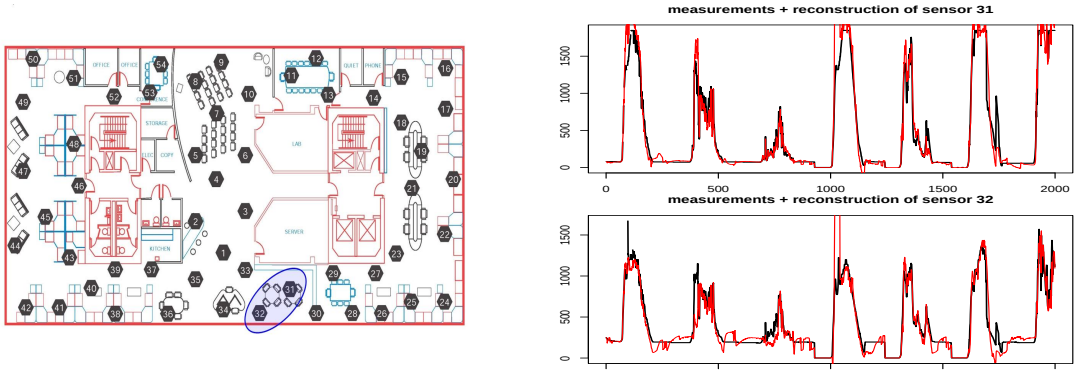
Data characteristics. The main characteristics are:

- A clear global periodic pattern (daily cycle).



(a) Network map (b) Measurements and reconstruction (c) Hidden variables

Figure 2.4: `Chlorine` dataset: (a) the network layout. (b) actual measurements and reconstruction at four junctions (highlighted in (a)). We plot only 500 consecutive timestamps (the patterns repeat after that). (c) shows SPIRIT's hidden variables.



(a) Lab map (b) Original measurements vs. reconstruction

Figure 2.5: `Mote` reconstruction on two highlighted sensors

- Occasional big spikes from some sensors (outliers).

Results of SPIRIT. SPIRIT detects four hidden variables (see Figure 2.6). Two of these are intermittent and correspond to outliers, or changes in the correlated trends. We show the reconstructions for some of the observed variables in Figure 2.5(b).

Interpretation. In summary, the first two hidden variables (see Figure 2.6) correspond to the global trend and the last two, which are intermittently present, correspond to outliers. In particular:

- The first hidden variable captures the global periodic pattern.
- The interpretation of the second one is again similar to the `Chlorine` dataset. The

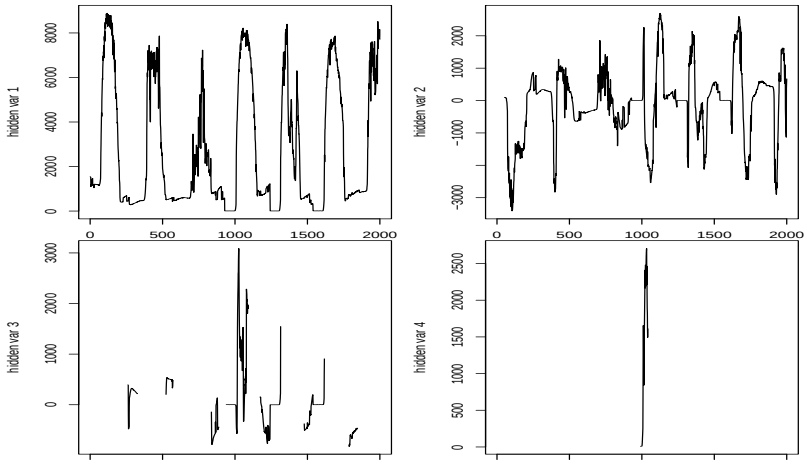


Figure 2.6: `Mote` dataset, hidden variables: The third and fourth hidden variables are intermittent and indicate "anomalous behavior." Note that the axes limits are different in each plot.

first two hidden variables together are sufficient to express arbitrary phase shifts.

- The third and fourth hidden variables indicate some of the potential outliers in the data. For example, there is a big spike in the 4th hidden variable at time $t = 1033$, as shown in Figure 2.6. Examining the participation weights $\mathbf{w}_4$ at that timestamp, we can find the corresponding sensors "responsible" for this anomaly, i.e., those sensors whose participation weights have very high magnitude. Among these, the most prominent are sensors 31 and 32. Looking at the actual measurements from these sensors, we see that before time $t = 1033$ they are almost 0. Then, very large increases occur around $t = 1033$, which bring an additional hidden variable into the system.

Room temperatures

Description. The `Critter` dataset consists of 8 streams (see Figure 2.8). Each stream comes from a small sensor[3] (aka. Critter) that connects to the joystick port and measures temperature. The sensors were placed in 5 neighboring rooms. Each time tick represents the average temperature during one minute.

[3] `http://www.ices.cmu.edu/sensornets/`

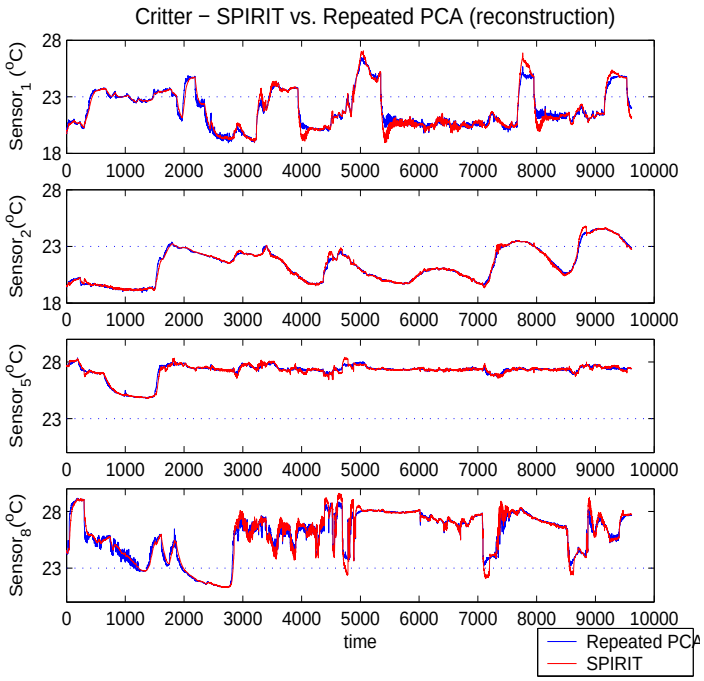


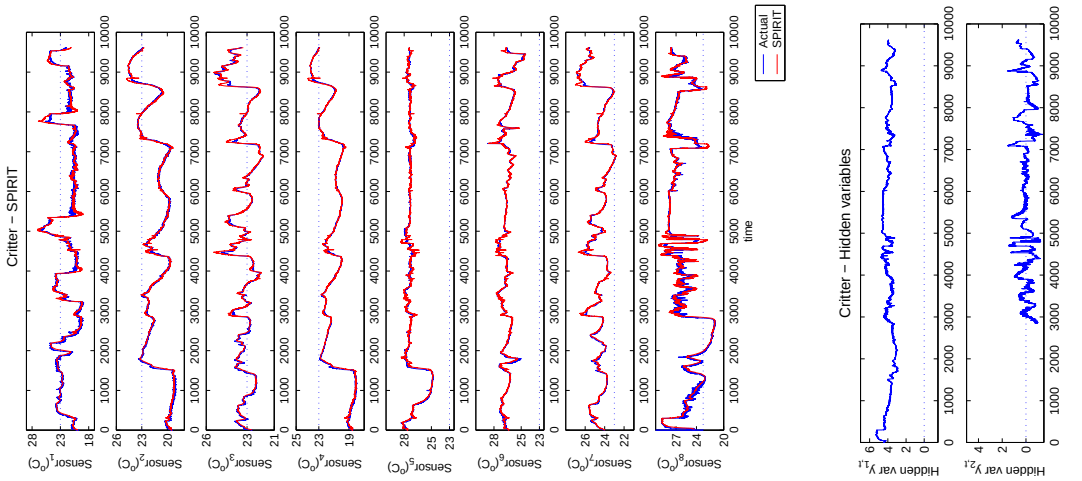
Figure 2.7: Reconstructions $\tilde{\mathbf{x}}_t$ for `Critter`.

Furthermore, to demonstrate how the correlations capture information about missing values, we repeated the experiment after blanking 1.5% of the values (five blocks of *consecutive* timestamps; see Figure 2.9).

Data characteristics. Overall, the dataset does not seem to exhibit a clear trend. Upon closer examination, all sensors fluctuate slightly around a constant temperature (which ranges from 22–27^{o}C, or 72–81^{o}F, depending on the sensor). Approximately half of the sensors exhibit a more similar "fluctuation pattern."

Results of SPIRIT. SPIRIT discovers one hidden variable, which is sufficient to capture the general behavior. However, if we utilize prior knowledge (such as, e.g., that the pre-set temperature was 23^{o}C), we can ask SPIRIT to detect trends with respect to that. In that case, SPIRIT comes up with two hidden variables, which we explain later.

SPIRIT is also able to deal successfully with missing values in the streams. Figure 2.9 shows the results on the blanked version (1.5% of the total values in five blocks of *consecutive* timestamps, starting at a different position for each stream) of `Critter`. The correlations captured by SPIRIT's hidden variable often provide useful information about



(a) SPIRIT reconstruction (b) Hidden variables

Figure 2.8: (a) Actual `Critter` data and SPIRIT output, (b) hidden variables.

the missing values. In particular, on sensor 8 (second row, Figure 2.9), the correlations picked by the *single* hidden variable successfully capture the missing values in that region (consisting of 270 ticks). On sensor 7, (first row, Figure 2.9; 300 blanked values), the upward trend in the blanked region is also picked up by the correlations. Even though the trend is slightly mis-estimated, as soon as the values are observed again, SPIRIT very quickly gets back to near-perfect tracking.

Interpretation. If we examine the participation vector $\mathbf{w}_1$, the largest entries in $\mathbf{w}_1$ correspond primarily to streams 5 and 6, and then to stream 8. If we examine the data, sensors 5 and 6 consistently have the highest temperatures, while sensor 8 also has a similar temperature most of the time.

However, if the sensors are calibrated based on the fact that these are building temperature measurements, where we have set the thermostat to 23^oC (73^oF), then SPIRIT discovers two hidden variables (see Figure 2.8). More specifically, if we reasonably assume that we have the prior knowledge of what the temperature *should be* (note that this has nothing to do with the average temperature in the observed data) and want to discover what happens around that temperature, we can subtract it from each observation and SPIRIT will discover patterns and anomalies based on this information. Actually, this is what a human operator would be interested in discovering: "Does the system work as I expect it to?" (based on my knowledge of how it should behave) and "If not, what is wrong?" So, in this case, we indeed discover this information.

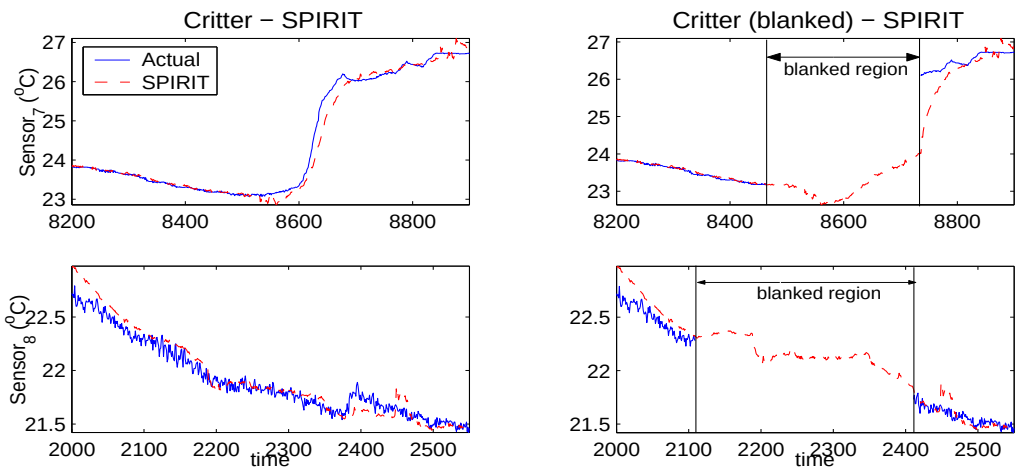
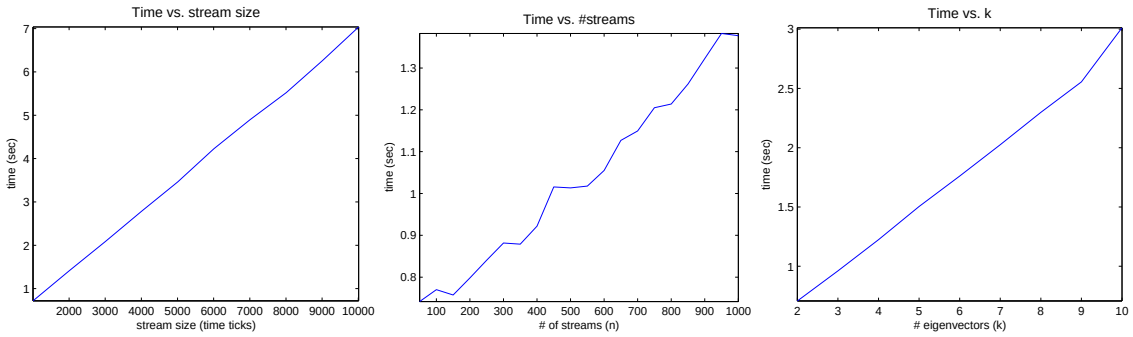


Figure 2.9: Missing value imputation: Detail of the forecasts on `Critter` with blanked values. The second row shows that the correlations picked by the *single* hidden variable successfully capture the missing values in that region (consisting of 270 *consecutive* ticks). In the first row (300 consecutive blanked values), the upward trend in the blanked region is also picked up by the correlations to other streams. Even though the trend is slightly mis-estimated, as soon as the values are observed again SPIRIT quickly gets back to near-perfect tracking.

- The interpretation of the first hidden variable is similar to that of the original signal: sensors 5 and 6 (and, to a lesser extent, 8) deviate from that temperature the most, for most of the time. Maybe the thermostats are broken or set wrong?
- For $\mathbf{w}_2$, the largest weights correspond to sensors 1 and 3, then to 2 and 4. If we examine the data, we notice that these streams follow a similar, fluctuating trend (close to the pre-set temperature), the first two varying more violently. The second hidden variable is added at time $t = 2016$. If we examine the plots, we see that, at the beginning, most streams exhibit a slow dip and then ascent (e.g., see 2, 4 and 5 and, to a lesser extent, 3, 7 and 8). However, a number of them start fluctuating more quickly and violently when the second hidden variable is added.

2.2.4 Performance and accuracy

In this section we discuss performance issues. First, we show that SPIRIT requires very limited space and time. Next, we elaborate on the accuracy of SPIRIT's incremental esti-



(a) stream size t (b) streams n (c) hidden variables k

Figure 2.10: Wall-clock times (including time to update forecasting models). The starting values are: (a) 1000 time ticks, (b) 50 streams, and (c) 2 hidden variables (the other two held constant for each graph). It is clear that SPIRIT scales linearly.

mates.

Time and space requirements

Figure 2.10 shows that SPIRIT scales linearly with respect to number of streams n and number of hidden variables k. AR per stream and MUSCLES are essentially off the charts from the very beginning. Furthermore, SPIRIT scales linearly with stream size (i.e., requires constant processing time per tuple).

The plots were generated using a synthetic dataset that allows us to precisely control each variable. The datasets were generated as follows:

- Pick the number k of trends and generate sine waves with different frequencies, say $y_{t,i} = \sin(2\pi i/kt), 1 \leq i \leq k$. Thus, all trends are pairwise linearly independent.
- Generate each of the n streams as random linear combinations of these k trend signals.

This allows us to vary k, n and the length of the streams at will. For each experiment shown, one of these parameters is varied and the other two are held fixed. The numbers in Figure 2.10 are wall-clock times of our Matlab implementation. Both AR-per-stream as well as MUSCLES (also in Matlab) are several orders of magnitude slower and thus omitted from the charts.

It is worth mentioning that we have also implemented the SPIRIT algorithms in real

Dataset	Chlorine	Critter	Motes
MSE rate - SPIRIT	0.0359	0.0827	0.0669
MSE rate - repeated PCA	0.0401	0.0822	0.0448

Table 2.2: Reconstruction accuracy (mean squared error rate).

systems [115, 73], which can obtain measurements from sensor devices and display hidden variables and trends in real-time.

Accuracy

In terms of accuracy, everything boils down to the quality of the summary provided by the hidden variables. To this end, we show the reconstruction $\tilde{\mathbf{x}}_t$ of $\mathbf{x}_t$, from the hidden variables $\mathbf{y}_t$ in Figure 2.7. One line uses the true principal directions, the other the SPIRIT estimates (i.e., weight vectors). SPIRIT comes very close to repeated PCA.

We should note that this is an unfair comparison for SPIRIT, since repeated PCA requires (i) storing *all* stream values, and (ii) performing a very expensive SVD computation for *each* time tick. However, the tracking is still very good. This is always the case, provided the corresponding eigenvalue is large enough and fairly well-separated from the others. If the eigenvalue is small, then the corresponding hidden variable is of no importance and we do not track it anyway.

Reconstruction error. Table 2.2 shows the reconstruction error (mean square error - MSE), $\sum \|\tilde{\mathbf{x}}_t - \mathbf{x}_t\|^2 / \sum \|\mathbf{x}_t\|^2$, achieved by SPIRIT. In every experiment, we set the energy thresholds to $[f_E, F_E] = [0.95, 0.98]$. Also, as pointed out before, we set $\lambda = 0.96$ as a reasonable default value to deal with non-stationarities that may be present in the data, according to recommendations in the literature [71]. Since we want a metric of overall quality, the MSE rate weighs each observation equally and does not take into account the forgetting factor λ.

Still, the MSE rate is very close to the bounds we set. In Table 2.2 we also show the MSE rate achieved by repeated PCA. As pointed out before, this is already an unfair comparison. In this case, we set the number of principal components k to the maximum that SPIRIT uses at any point in time. This choice favors repeated PCA even further. Despite this, the reconstruction errors of SPIRIT are close to the ideal, while using orders of magnitude less time and space.

2.2.5 Summary

We focus on finding patterns, correlations and hidden variables, in multiple streams. SPIRIT has the following desirable characteristics:

- It discovers underlying correlations among multiple streams, incrementally and in real-time [115, 73] and provides a very compact representation of the stream collection, via a few *hidden variables*.
- It automatically estimates the number k of hidden variables to track, and it can automatically adapt, if k changes (e.g., an air-conditioner switching on, in a temperature sensor scenario).
- It scales up extremely well, both on database size (i.e., number of time ticks t), and on the number n of streams. Therefore it is suitable for a large number of sensors / data sources.
- Its computation demands are low: it only needs $O(nk)$ floating point operations—no matrix inversions nor SVD (both infeasible in streaming settings). Its space demands are similarly small.
- It can naturally hook up with any forecasting method, and thus easily do prediction, as well as handle missing values.

We showed that the output of SPIRIT has a natural interpretation. We evaluated our method on several datasets, where indeed it discovered the hidden variables. Moreover, SPIRIT-based forecasting was several times faster than other methods.

2.3 Distributed Stream Mining

How to perform stream mining in a distributed environment? After describing SPIRIT in a centralized environment, we now extend it to a distributed environment. Multiple co-evolving streams often arise in a large distributed system, such as computer networks and sensor networks. Centralized approaches usually will not work in this setting. The reasons are:

1. **Communication constraint**; it is too expensive to transfer all data to a central node for processing and mining.

2. **Power consumption**; in a wireless sensor network, minimizing information exchange is crucial because many sensors have very limited power. Moreover, wireless power consumption between two nodes usually increases quadratically with the distance, which implies that transmitting all messages to single node is prohibitively expensive.

3. **Robustness concerns**; centralized approaches always suffer from single point of failure.

4. **Privacy concerns**[4]; in any network connecting multiple autonomous systems (e.g., multiple companies forming a collaborative network), no system is willing to share all the information, while they all want to know the global patterns.

To sum up, a **distributed online algorithm** is highly needed to address all the above concerns.

To address this problem, we propose a hierarchical framework that intuitively works as follows:1) Each autonomous system first finds its local patterns and shares them with other groups (details in subsection 2.3.3). 2) Global patterns are discovered based on the shared local patterns (details in subsection 2.3.4). 3) From the global patterns, each autonomous system further refines/verifies their local patterns.

Contributions: The problem of pattern discovery in a large number of co-evolving groups of streams has important applications in many different domains. We introduce a hierarchical framework to discover local and global patterns effectively and efficiently across multiple groups of streams. The proposed method satisfies the following requirements:

- It is *streaming*, i.e., it is incremental without any buffering of historical data.
- It scales *linearly* with the number of streams.
- It runs in a distributed fashion requiring small communication cost.
- It avoids a single point of failure, which all centralized approaches have.
- It utilizes the global patterns to improve and refine the local patterns, in a simple and elegant way.
- It reduces the information that has to be shared across different groups, thereby protecting privacy.

[4]Section 2.5 provides more discussion on this topic

The local and global patterns we discover have multiple uses. They provide a succinct summary to the user. Potentially, users can compare their own local patterns with the global ones to detect outliers. Finally, they facilitate interpolations and handling of missing values as shown in Section 2.2.

2.3.1 Problem Formulation

Given m groups of streams which consist of $\{n_1, \ldots, n_m\}$ co-evolving numeric streams, respectively, we want to solve the following two problems: (i) incrementally find patterns within a single group (*local pattern monitoring*), and (ii) efficiently obtain global patterns from all the local patterns (*global pattern detection*).

More specifically, we view original streams as points in a high-dimensional space, with one point per time tick. Local patterns are then extracted as low-dimensional projections of the original points. Furthermore, we continuously track the basis of the low-dimensional spaces for each group in a way that global patterns can be easily constructed. More formally, the i-th group $\mathbf{S}^{(i)}$ consists of a (unbounded) sequence of n_i-dimensional vectors(points) where n_i is the number of streams in $\mathbf{S}^{(i)}$, $1 \leq i \leq m$. $\mathbf{S}^{(i)}$ can also be viewed as a matrix with n_i rows and an unbounded number of columns. The intersection $\mathbf{s}_t^{(i)}(l)$ at the l-th row and t-th column of $\mathbf{S}^{(i)}$, represents the value of the l-th node/stream recorded at time t in the i-th group. The t-th column of $\mathbf{S}^{(i)}$, denoted as $\mathbf{s}_t^{(i)}$, is the column vector of all the values recorded at time t in i-th group. Note that we assume measurements from different nodes within a group are synchronized along the time dimension. We believe this constraint can be relaxed, but it would probably lead to a more complicated solution. *Local pattern monitoring* can be modeled as a function,

$$F_L : (\mathbf{s}_{t+1}^{(i)}, \mathbf{g}_t) \rightarrow \mathbf{l}_{t+1}^{(i)}, \tag{2.1}$$

where the inputs are 1) the new input vector $\mathbf{s}_{t+1}^{(i)}$ at time $t+1$ and the current global pattern $\mathbf{g}_t$ and the output is the local pattern $\mathbf{l}_{t+1}^{(i)}$ at time $t+1$. Details on constructing such a function will be explained in subsection 2.3.3. Likewise, *global pattern detection* is modeled as another function,

$$F_G : (\mathbf{l}_{t+1}^{(1)}, \ldots, \mathbf{l}_{t+1}^{(m)}) \rightarrow \mathbf{g}_{t+1}, \tag{2.2}$$

where the inputs are local patterns $\mathbf{l}_{t+1}^{(i)}$ from all groups at time $t+1$ and the output is the new global pattern $\mathbf{G}_{t+1}$.

Algorithm 2.3: DISTRIBUTEDMINING

(*Initialization*) At $t = 0$, set $\mathbf{g}_t \leftarrow$ null
forall $t > 1$ **do**
    `// Update local patterns`
    for $i \leftarrow 1$ *to* m **do**
        set $\mathbf{l}_t^{(i)} := F_L(\mathbf{s}_t^{(i)}, \mathbf{g}_{t-1})$
    `// Update global patterns`
    Set $\mathbf{g}_t := F_G(\mathbf{l}_t^{(1)}, \ldots, \mathbf{l}_t^{(m)})$

2.3.2 Distributed mining framework

In this section, we introduce the general framework for distributed mining. More specifically, we present the meta-algorithm to show the overall flow, using F_L (*local patterns monitoring*) and F_G (*global patterns detection*) as black boxes.

Intuitively, it is natural that global patterns are computed based on all local patterns from m groups. On the other hand, it might be a surprise that the local patterns of group i take as input both the stream measurements of group i and the global patterns. Stream measurements are a natural set of inputs, since local patterns are their summary. However, we also need global patterns as another input so that local patterns can be represented consistently across all groups. This is important at the next stage, when constructing global patterns out of the local patterns; we elaborate on this later. The meta-algorithm is the following:

2.3.3 Local pattern monitoring

In this section we present the method for discovering patterns within a stream group. More specifically, we explain the details of function F_L (Equation 2.1). We first describe the intuition behind the algorithm and then present the algorithm formally. Finally we discuss how to determine the number of local patterns k_i.

The goal of F_L is to find the low dimensional projection $\mathbf{l}_t^{(i)}$ and the participation weight matrix $\mathbf{W}^{(i,t)} \in \mathbb{R}^{n_i \times k_i}$ so as to guarantee that the reconstruction error $\|\mathbf{s}_t^{(i)} - \hat{\mathbf{S}}_t^{(i)}\|^2$ over time is predictably small. Note that the reconstruction of $\mathbf{s}_t^{(i)}$ is defined as $\hat{\mathbf{S}}_t^{(i)} := \mathbf{W}^{(i,t)\mathsf{T}}\mathbf{l}_t^{(i)}$. Weight matrix $\mathbf{W}^{(i,t)}$ provides the transformation between original space and projected low-dimensional space for the i-th stream group at time t.

Algorithm 2.4: F_L(new vector $\mathbf{s}_{t+1}^{(i)} \in \mathbb{R}^{n_i}$, old global patterns $\mathbf{g}_t$)

Output: local patterns (k_i-dimensional projection) $\mathbf{l}_{t+1}^{(i)}$

Initialize $\mathbf{x} := \mathbf{s}_{t+1}^{(i)}$

for $1 \leq j \leq k$ **do**

  $y_j := \mathbf{w}_j^{(i,t)\mathsf{T}}\mathbf{x}$ /* y_j = projection onto $\mathbf{w}_j^{(i,t)}$ */

  $d_j \leftarrow \lambda d_j + y_j^2$ /* local energy, determining update magnitude */

  $\mathbf{e} := \mathbf{x}_j - \mathbf{g}_t(j)\mathbf{w}_j^{(i,t)}$ /* error, $\mathbf{e} \perp \mathbf{w}_j^{(i,t)}$ */

  $\mathbf{w}_j^{(i,t+1)} \leftarrow \mathbf{w}_j^{(i,t)} + \frac{1}{d_j}\mathbf{g}_t(j)\mathbf{e}$ /* update participation weight */

  $\mathbf{x} := \mathbf{x} - \mathbf{g}_t(j)\mathbf{w}_j^{(i,t+1)}$ /* repeat with remainder of $\mathbf{x}$ */

Compute the new projection $\mathbf{l}_{t+1}^{(i)} := \mathbf{W}^{(i,t+1)\mathsf{T}}\mathbf{s}_{t+1}^{(i)}$

Tracking local patterns: The first step is, for a given k_i, to incrementally update the $n_i \times k_i$ participation weight matrix $\mathbf{W}^{(i,t)}$, which serves as a basis of the low-dimensional projection for $\mathbf{s}_t^{(i)}$. In particular, the j-th column of $\mathbf{W}^{(i,t)}$ is denoted as $\mathbf{w}_j^{(i,t)}$. $\mathbf{W}^{(i,t)}$ is orthonormal, i.e., $\mathbf{w}_j^{(i,t)} \perp \mathbf{w}_k^{(i,t)}, j \neq k$ and $\|\mathbf{w}_j^{(i,t)}\| = 1$. Later in this section, we describe the method for choosing k_i. For the moment, assume that the number of patterns k_i is given.

The main idea behind the algorithm is to read the new values $\mathbf{s}_{t+1}^{(i)}$ from the n_i streams of group i at time $t+1$, and perform three steps. These are very similar to SPIRIT in Section 2.2, except that the energy calculation is coordinated across all groups for the ease of global pattern computation.

1. **Compute** the low dimensional projection $y_j, 1 \leq j \leq k_i$, based on the *current* weights $\mathbf{W}^{(i,t)}$, by projecting $\mathbf{s}_{t+1}^{(i)}$ onto these.
2. **Estimate** the reconstruction error ($\mathbf{e}_j$ below) and the energy.
3. **Compute** $\mathbf{W}^{(i,t+1)}$ and output the *actual* local pattern $\mathbf{l}_{t+1}^{(i)}$.

Similar to SPIRIT in Section 2.2, we use λ to capture the concept drifting.

Detecting the number of local patterns: For each group, we have a low-energy and a high-energy threshold, $f_{i,E}$ and $F_{i,E}$, respectively. We keep enough local patterns k_i, so

Algorithm 2.5: F_G(all local patterns $\mathbf{l}_t^{(1)}, \ldots, \mathbf{l}_t^{(m)}$)

Output: global patterns $\mathbf{g}_t$

1 Set $k := max(k_i)$ for $1 \leq i \leq m$

2 Zero-padding all $\mathbf{l}_t^{(i)}$ to be of length k

3 Set $\mathbf{g}_t := \sum_{i=1}^{m} \mathbf{l}_t^{(i)}$

the retained energy is within the range $[f_{i,E} \cdot E_{i,t}, F_{i,E} \cdot E_{i,t}]$. This is the same criteria as in SPIRIT (Section 2.2) but applied to each stream group.

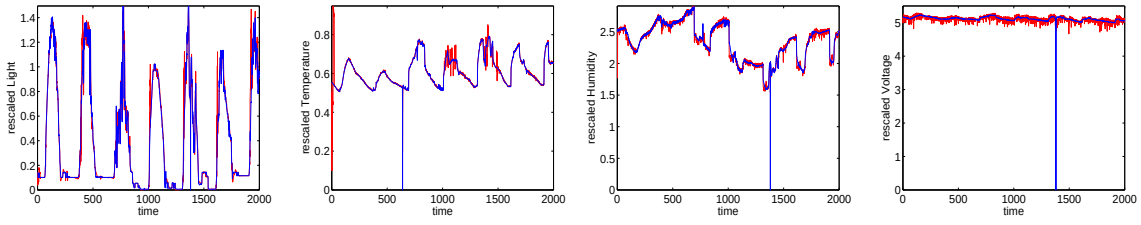
2.3.4 Global pattern detection

In this section we present the method for obtaining global patterns over all groups. More specifically, we explain the details of function F_G (Equation 2.2).

First of all, what is a global pattern? Similar to local pattern, global pattern is low dimensional projections of the streams from all groups. Loosely speaking, assume only one global group exists which consists of all streams, the global patterns are the local patterns obtained by applying F_L on the global group—this is essentially the centralized approach. In other words, we want to obtain the result of the centralized approach without centralized computation.

The algorithm exactly follows the lemma above. The j-th global pattern is the sum of all the j-th local patterns from m groups.

2.3.5 Experiments and case studies



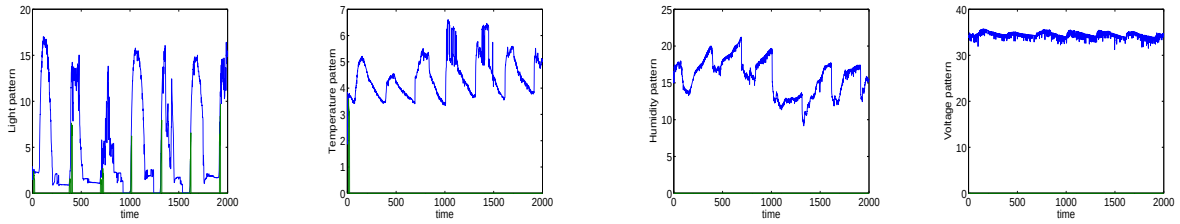
(a) Light measurements (b) Temperature (c) Humidity (c) Voltage

Figure 2.11: original measurements (blue) and reconstruction (red) are very close.

The `Motes` dataset consists of 4 groups of sensor measurements (i.e., light intensity, humidity, temperature, battery voltages) collected using 48 Berkeley Mote sensors at different locations in a lab, over a period of a month. Note that Section 2.2.3 uses the light measurements in part of the experiment, and here is a more complete data all four types of sensor measurements. This is an example of heterogeneous streams. All the streams are scaled to have unit variance, to be comparable across different measures. In particular, streams from different groups behave very differently. This can be considered as a bad scenario for our method. The goal is to show that the method can still work well even when the groups are not related. If we do know the groups are unrelated up front, we can treat them separately without bothering to find global patterns. However, in practice, such prior knowledge is not available. Our method is still a sound approach in this case.

The main characteristics (see the blue curves in Figure 2.11) are: (1) Light measurements exhibit a clear global periodic pattern (daily cycle) with occasional big spikes from some sensors (outliers), (2) Temperature shows a weak daily cycle and a lot of bursts. (3) Humidity does not have any regular pattern. (4) Voltage is almost flat with a small downward trend.

The reconstruction is very good (see the red curves in Figure 2.11(a)), with relative error below 6%. Furthermore, the local patterns from different groups are correlated well with the original measurements (see Figure 2.12). The global patterns (in Figure 2.13) are combinations of different patterns from all groups and reveal the overall behavior of all the groups.



(a) Light patterns (b) Temperature patterns (c) Humidity patterns (d) Voltage patterns

Figure 2.12: Local patterns

In terms of accuracy, everything boils down to the quality of the summary provided by the local/global patterns. To this end, we use the relative reconstruction error ($\|\mathbf{s}–\hat{\mathbf{s}}\|^2/\|\mathbf{s}\|^2$ where $\mathbf{s}$ are the original streams and $\hat{\mathbf{s}}$ are the reconstructions) as the evaluation metric. The best performance is obtained when accurate global patterns are known to all groups. But this requires exchanging up-to-date local/global patterns at every timestamp among all groups, which is prohibitively expensive. One efficient way to deal with this problem

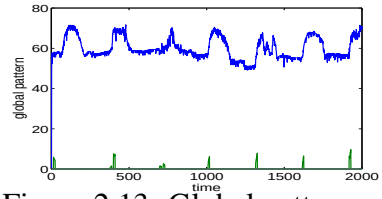


Figure 2.13: Global patterns

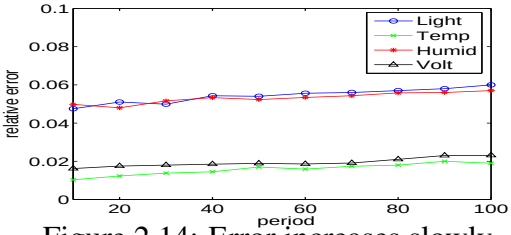


Figure 2.14: Error increases slowly

is to increase the communication period, which is the number of timestamps between successive local/global pattern transmissions. For example, we can achieve a 10-fold communication reduction by changing the period from 10 to 100 timestamps. Figure 2.14 shows reconstruction error vs. communication period. Overall, the relative error rate increases very slowly as the communication period increases. This implies that we can dramatically reduce communication with minimal sacrifice of accuracy.

2.3.6 Summary

We focus on finding patterns in a large number of distributed streams. More specifically, we first find local patterns within each group, where the number of local patterns is automatically determined based on reconstruction error. Next, global patterns are identified, based on the local patterns from all groups. Distributed SPIRIT has the following desirable characteristics:

- It discovers underlying correlations among multiple stream groups incrementally, via a few patterns.
- It automatically estimates the number k_i of local patterns to track, and it can automatically adapt, if k_i changes.
- It is distributed, avoiding a single point of failure and reducing communication cost and power consumption.
- It utilizes the global patterns to improve and refine the local patterns, in a simple and elegant way.
- It requires limited shared information from different groups, while being able to successfully monitor global patterns.
- It scales up extremely well, due to its incremental and hierarchical nature.

- Its computation demands are low. Its space demands are also limited: no buffering of any historical data.

We evaluated our method on several datasets, where it indeed discovered the patterns. We gain significant communication savings, with small accuracy loss.

2.4 Local Correlation Tracking of a pair of streams

How to define and compute correlations for streams The notion of correlation (or, similarity) is important, since it allows us to discover groups of objects with similar behavior and, consequently, discover potential anomalies which may be revealed by a change in correlation. In this section we consider correlation among time series which often exhibit two important properties.

First, their characteristics may change over time. In this case, a single, static correlation score for the entire time series is less useful. Instead, it is desirable to have a notion of correlation that also evolves with time and tracks the changing relationships. On the other hand, a time-evolving correlation score should not be overly sensitive to transients; if the score changes wildly, then its usefulness is limited.

The second property is that many time series exhibit strong but fairly complex, non-linear correlations. Traditional measures, such as the widely used *cross-correlation coefficient* (or, *Pearson coefficient*), are less effective in capturing these complex relationships. Consequently, we seek a concise but powerful model that can capture various trend or pattern types.

Data with such features arise in several application domains, such as:

- Monitoring of network traffic flows or of system performance metrics (e.g., CPU and memory utilization, I/O throughput, etc), where changing workload characteristics may introduce non-stationarity [73].
- Financial applications, where prices may exhibit linear or seasonal trends, as well as time-varying volatility [18].
- Medical applications, such as EEGs (electroencephalograms) [25].

In this section, we introduce LoCo (LOcal COrrelation), a time-evolving, local similarity score for time series, by generalizing the notion of cross-correlation coefficient. The model upon which our score is based can capture fairly complex relationships and

track their evolution. The linear cross-correlation coefficient is included as a special case. Our approach is also amenable to robust streaming estimation. We illustrate our proposed method or real data, discussing its qualitative interpretation, comparing it against natural alternatives and demonstrating its robustness and efficiency.

2.4.1 Localizing correlation estimates

Our goal is to derive a time-evolving correlation scores that tracks the similarity of time-evolving time series. Thus, our method should have the following properties:

(P1) Adapt to the time-varying nature of the data,

(P2) Employ a simple, yet powerful and expressive joint model to capture correlations,

(P3) The derived score should be robust, reflecting the evolving correlations accurately, and

(P4) It should be possible to estimate it efficiently.

We will address most of these issues in Section 2.4.2, which describes our proposed method. In this section, we introduce some basic definitions to facilitate our discussion. We also introduce localized versions of popular similarity measures for time series.

As a first step to deal with (P1), any correlation score at time instant $t \in \mathbb{N}$ should be based on observations in the "neighborhood" of that instant. Therefore, we introduce the notation $\mathbf{x}_{t,w} \in \mathbb{R}^w$ for the subsequence of the series, starting at t and having length w,

$$\mathbf{x}_{t,w} := [x_t, x_{t+1}, \ldots, x_{t+w-1}]^\mathsf{T}.$$

Furthermore, any correlation score should satisfy two elementary and intuitive properties.

Definition 2.2 (Local correlation score)**.** *Given a pair of time series X and Y, a* local correlation score *is a sequence $c_t(X, Y)$ of real numbers that satisfy the following properties, for all $t \in \mathbb{N}$:*

$$0 \leq c_t(X, Y) \leq 1 \quad \textit{and} \quad c_t(X, Y) = c_t(Y, X).$$

We denote a time series process as an indexed collection X of random variables $X_t, t \in \mathbb{N}$, i.e., $X = \{X_1, X_2, \ldots, X_t, \ldots\} \equiv \{X_t\}_{t \in \mathbb{N}}$. Without loss of generality, we will assume zero-mean time series, i.e., $\mathrm{E}[X_t] = 0$ for all $t \in \mathbb{N}$. The values of a particular realization of X are denoted by lower-case letters, $x_t \in \mathbb{R}$, at time $t \in \mathbb{N}$.

Symbol	Description
$\mathbf{U}, \mathbf{V}$	Matrix (uppercase bold).
$\mathbf{u}, \mathbf{v}$	Column vector (lowercase bold).
x_t	Time series, $t \in \mathbb{N}$.
w	Window size.
$\mathbf{x}_{t,w}$	Window starting at t, $\mathbf{x}_{t,w} \in \mathbb{R}^w$.
m	Number of windows (typically, $m = w$).
β	Exponential decay weight, $0 \leq \beta \leq 1$.
$\hat{\Gamma}_t$	Local autocorrelation matrix estimate (w-by-w).
$\mathbf{u}_i(\mathbf{A})$, $\lambda_i(\mathbf{A})$	Eigenvectors and corresponding eigenvalues of $\mathbf{A}$.
$\mathbf{U}_k(\mathbf{A})$	Matrix of k largest eigenvectors of $\mathbf{A}$.
ℓ_t	LoCo score.
ϕ_t	Fourier local correlation score.
ρ_t	Pearson local correlation score.

Table 2.3: Main symbols used in Section 2.4

Local Pearson

Before proceeding to describe our approach, we formally define natural extensions of two methods that have been widely used for global correlation or similarity among "static" time series.

Pearson coefficient A natural local adaptation of cross-correlation is the following:

Definition 2.3 (Local Pearson correlation)**.** *The local Pearson correlation is the linear cross-correlation coefficient*

$$\rho_t(X, Y) := \frac{\left|\mathrm{Cov}[\mathbf{x}_{t,w}, \mathbf{y}_{t,w}]\right|}{\mathrm{Var}[\mathbf{x}_{t,w}]\,\mathrm{Var}[\mathbf{y}_{t,w}]} = \frac{|\mathbf{x}_{t,w}^{\mathsf{T}}\mathbf{y}_{t,w}|}{\|\mathbf{x}_{t,w}\| \cdot \|\mathbf{y}_{t,w}\|},$$

where the last equality follows from $\mathrm{E}[X_t] = \mathrm{E}[Y_t] = 0$.

It follows directly from the definition that ρ_t satisfies the two requirements, $0 \leq \rho_t(X, Y) \leq 1$ and $\rho_t(X, Y) = \rho_t(Y, X)$.

2.4.2 Correlation tracking through local auto-covariance

In this section we develop our proposed approach, the *Local Correlation (LoCo)* score. Returning to properties (P1)–(P4) listed in the beginning of Section 2.4.1, the next section addresses primarily (P1) and Section 2.4.2 continues to address (P2) and (P3). Next, Section 2.4.2 shows how (P4) can also be satisfied and, finally, Section 2.4.3 discusses the time and space complexity of the various alternatives.

Local auto-covariance

The first step towards tracking local correlations at time $t \in \mathbb{N}$ is restricting, in some way, the comparison to the "neighborhood" of t, which is the reason for introducing the notion of a window $\mathbf{x}_{t,w}$.

If we stop there, we can compare the two windows $\mathbf{x}_{t,w}$ and $\mathbf{y}_{t,w}$ *directly*. If, in addition, the comparison involves capturing any *linear* relationships between localized values of X and Y, this leads to the local Pearson correlation score ρ_t. However, this joint model of the series it is too simple, leading to two problems: (i) it cannot capture more complex relationships, and (ii) it is too sensitive to transient changes, often leading to widely fluctuating scores.

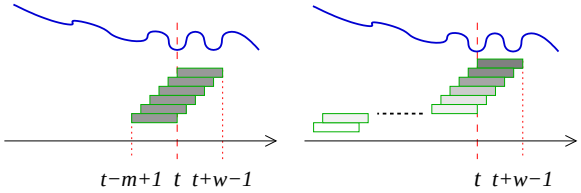
Intuitively, we will estimate the *full* auto-covariance matrix of values "near" t, and avoid making *any* assumptions about stationarity[5]. Any estimate of the local auto-covariance at time t needs to be based on a "localized" sample set of windows with length w. We will consider two possibilities:

- Sliding (a.k.a. boxcar) window (see Figure 2.15[a]): We use exactly m windows around t, specifically $\mathbf{x}_{\tau,w}$ for $t - m + 1 \leq \tau \leq t$, and we weigh them equally. This takes into account $w + m - 1$ values in total, around time t.

- Exponential window (see Figure 2.15[b]): We use all windows $\mathbf{x}_{\tau,w}$ for $1 \leq \tau \leq t$, but we weigh those close to t more, by multiplying each window by a factor of $\beta^{t-\tau}$.

These two alternatives are illustrated in Figure 2.15, where the shading corresponds to the weight. We will explain how to "compare" the local auto-covariance matrices of two series in Section 2.4.2. Next, we formally define these estimators.

[5]If $\gamma_{t,'t} \equiv \gamma_{|t-t'|}$, then the auto-covariance matrix is *circulant*, i.e., it has constant diagonals and (up to) w distinct values.



(a) Sliding window (b) Exponential window

Figure 2.15: Local auto-covariance; shading corresponds to weight.

Definition 2.4 (Local autocovariance, sliding window)**.** *Given a time series X, the local auto-covariance matrix estimator $\hat{\Gamma}_t$ using a sliding window is defined at time $t \in \mathbb{N}$ as*

$$\hat{\Gamma}_t(X, w, m) := \sum_{\tau=t-m+1}^{t} \mathbf{x}_{\tau,w} \circ \mathbf{x}_{\tau,w}.$$

The sample set of m windows is "centered" around time t. We typically fix the number of windows to $m = w$, so that $\hat{\Gamma}_t(X, w, m) = \sum_{\tau=t-w+1}^{t} \mathbf{x}_{\tau,w} \circ \mathbf{x}_{\tau,w}$. A normalization factor of $1/m$ is ignored, since it is irrelevant for the eigenvectors of $\hat{\Gamma}_t$.

Definition 2.5 (Local autocovariance, exponential window)**.** *Given a time series X, the local auto-covariance matrix estimator $\hat{\Gamma}_t$ at time $t \in \mathbb{N}$ using an exponential window is*

$$\hat{\Gamma}_t(X, w, \beta) := \sum_{\tau=1}^{t} \beta^{t-\tau} \mathbf{x}_{\tau,w} \circ \mathbf{x}_{\tau,w}.$$

Similar to the previous definition, we ignore the normalization factor $(1-\beta)/(1-\beta^{t+1})$. In both cases, we may omit some or all of the arguments X, w, m, β, when they are clear from the context.

Under certain assumptions, the *equivalent window* corresponding to an exponential decay factor β is given by $m = (1-\beta)^{-1}$ [133]. However, one of the main benefits of the exponential window is based on the following simple observation.

Property 2.1. *The sliding window local auto-covariance follows the equation*

$$\hat{\Gamma}_t = \hat{\Gamma}_{t-1} - \mathbf{x}_{t-w,w} \circ \mathbf{x}_{t-w,w} + \mathbf{x}_{t,w} \circ \mathbf{x}_{t,w},$$

whereas for the exponential window it follows the equation

$$\hat{\Gamma}_t = \beta \hat{\Gamma}_{t-1} + \mathbf{x}_{t,w} \circ \mathbf{x}_{t,w}.$$

Updating the sliding window requires buffering multiple windows, while updating the exponential window only needs the current window, therefore, more desirable.

The next simple lemma will be useful later, to show that ρ_t is included as a special case of the LoCo score. Intuitively, if we use an *instantaneous* estimate of the local auto-covariance $\hat{\Gamma}_t$, which is based on just the latest sample window $\mathbf{x}_{t,w}$, its eigenvector is the window itself.

Lemma 2.3. *If $m = 1$ or, equivalently, $\beta = 0$, then*

$$\mathbf{u}_1(\hat{\Gamma}_t) = \frac{\mathbf{x}_{t,w}}{\|\mathbf{x}_{t,w}\|} \quad \textit{and} \quad \lambda_1(\hat{\Gamma}_t) = \|\mathbf{x}_{t,w}\|^2.$$

Proof. In this case, $\hat{\Gamma}_t = \mathbf{x}_{t,w} \circ \mathbf{x}_{t,w}$ with rank 1. Its row and column space are $\operatorname{span} \mathbf{x}_{t,w}$, whose orthonormal basis is, trivially, $\mathbf{x}_{t,w}/\|\mathbf{x}_{t,w}\| \equiv \mathbf{u}_1(\hat{\Gamma}_t)$. The fact that $\lambda_1(\hat{\Gamma}_t) = \|\mathbf{x}_{t,w}\|^2$ then follows by straightforward computation, since $\mathbf{u}_1 \circ \mathbf{u}_1 = \mathbf{x}_{t,w} \circ \mathbf{x}_{t,w}/\|\mathbf{x}_{t,w}\|^2$, thus $(\mathbf{x}_{t,w} \circ \mathbf{x}_{t,w})\mathbf{u}_1 = \|\mathbf{x}_{t,w}\|^2 \mathbf{u}_1$. □

Pattern similarity

Given the estimates $\hat{\Gamma}_t(X)$ and $\hat{\Gamma}_t(Y)$ for the two series, the next step is how to "compare" them and extract a correlation score. Intuitively, we want to extract the "key information" contained in the auto-covariance matrices and measure how close they are. This is precisely where the spectral decomposition helps. The eigenvectors capture the key aperiodic and oscillatory trends, even in short, non-stationary series [63, 65]. These trends explain the largest fraction of the variance. Thus, we will use the subspaces spanned by the first few (k) eigenvectors of each local auto-covariance matrix to locally characterize the behavior of each series. The following definition formalizes this notion.

Definition 2.6 (LoCo score)**.** *Given two series X and Y, their LoCo score is defined by*

$$\ell_t(X, Y) := \tfrac{1}{2}\big(\|\mathbf{U}_X^\mathsf{T}\mathbf{u}_Y\| + \|\mathbf{U}_Y^\mathsf{T}\mathbf{u}_X\|\big),$$

where $\mathbf{U}_X \equiv \mathbf{U}_k(\hat{\Gamma}_t(X))$ and $\mathbf{U}_Y \equiv \mathbf{U}_k(\hat{\Gamma}_t(Y))$ are the eigenvector matrices of the local auto-covariance matrices of X and Y, respectively, and $\mathbf{u}_X \equiv \mathbf{u}_1(\hat{\Gamma}_t(X))$ and $\mathbf{u}_Y \equiv \mathbf{u}_1(\hat{\Gamma}_t(Y))$ are the corresponding eigenvectors with the largest eigenvalue.

In the above equation, $\mathbf{U}_X^\mathsf{T}\mathbf{u}_Y$ is the projection of $\mathbf{u}_Y$ onto the subspace spanned by the columns of the orthonormal matrix $\mathbf{U}_X$. The absolute cosine of the angle $\theta \equiv$

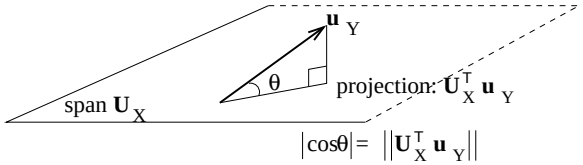


Figure 2.16: Illustration of LoCo definition.

$\angle(\mathbf{u}_Y, \operatorname{span} \mathbf{U}_X) = \angle(\mathbf{u}_Y, \mathbf{U}_X^\mathsf{T}\mathbf{u}_Y)$ is $|\cos\theta| = \|\mathbf{U}_X^\mathsf{T}\mathbf{u}_Y\|/\|\mathbf{u}_Y\| = \|\mathbf{U}_X^\mathsf{T}\mathbf{u}_Y\|$, since $\|\mathbf{u}_Y\| = 1$ (see Figure 2.16). Thus, ℓ_t is the average of the cosines $|\cos\angle(\mathbf{u}_Y, \operatorname{span} \mathbf{U}_X)|$ and $|\cos\angle(\mathbf{u}_X, \operatorname{span} \mathbf{U}_Y)|$. From this definition, it follows that $0 \leq \ell_t(X, Y) \leq 1$ and $\ell_t(X, Y) = \ell_t(Y, X)$. Furthermore, $\ell_t(X, Y) = \ell_t(-X, Y) = \ell_t(Y, -X) = \ell_t(-X, -Y)$—as is also the case with the other two scores, $\rho_t(X, Y)$ and $\phi_t(X, Y)$.

Intuitively, if the two series X, Y are locally similar, then the principal eigenvector of each series should lie within the subspace spanned by the principal eigenvectors of the other series. Hence, the angles will be close to zero and the cosines will be close to one.

The next simple lemma reveals the relationship between ρ_t and ℓ_t.

Lemma 2.4. *If window size $m = 1$ (whence, $k = 1$ necessarily), then LoCo score ℓ_t equals Pearson local correlation score ρ_t.*

Proof. From Lemma 2.3 it follows that $\mathbf{U}_X = \mathbf{u}_X = \mathbf{x}_{t,w}/\|\mathbf{x}_{t,w}\|$ and $\mathbf{U}_Y = \mathbf{u}_Y = \mathbf{y}_{t,w}/\|\mathbf{y}_{t,w}\|$. From the definitions of ℓ_t and ρ_t, we have $\ell_t = \frac{1}{2}\left(\frac{|\mathbf{x}_{t,w}^\mathsf{T}\mathbf{y}_{t,w}|}{\|\mathbf{x}_{t,w}\|\cdot\|\mathbf{y}_{t,w}\|} + \frac{|\mathbf{y}_{t,w}^\mathsf{T}\mathbf{x}_{t,w}|}{\|\mathbf{y}_{t,w}\|\cdot\|\mathbf{x}_{t,w}\|}\right) = \frac{|\mathbf{x}_{t,w}^\mathsf{T}\mathbf{y}_{t,w}|}{\|\mathbf{x}_{t,w}\|\cdot\|\mathbf{y}_{t,w}\|} = \rho_t$. □

Choosing k As we shall also see in Section 2.4.4, the directions of $\mathbf{x}_{t,w}$ and $\mathbf{y}_{t,w}$ may vary significantly, even at neighboring time instants. As a consequence, the Pearson score ρ_t (which is essentially based on the instantaneous estimate of the local auto-covariance) is overly sensitive. However, if we consider the low-dimensional subspace which is (mostly) occupied by the windows during a short period of time (as LoCo does), this is much more stable and less susceptible to transients, while still able to track changes in local correlation.

One approach is to set k based on the fraction of variance to retain (similar to criteria used in PCA [80], as well as in spectral estimation [113]). A simpler practical choice is to fix k to a small value; we use $k = 4$ throughout all experiments.

Choosing w Windows are commonly used in stream and signal processing applications. The size w of each window $\mathbf{x}_{t,w}$ (and, consequently, the size $w \times w$ of the auto-covariance matrix $\hat{\Gamma}_t$) essentially corresponds to the time scale we are interested in.

As we shall also see in Section 2.4.4, the LoCo score ℓ_t derived from the local auto-covariances changes gradually and smoothly with respect to w. Thus, if we set the window size to any of, say, 55, 60 or 65 seconds, we will qualitatively get the same results, corresponding approximately to patterns in the minute scale. Of course, at widely different time scales, the correlation scores will be different. If desirable, it is possible to track the correlation score at multiple scales, e.g., hour, day, month and year. If buffer space and processing time are a concern, either a simple decimated moving average filtering scheme or a more elaborate hierarchical SVD scheme (such as in [110]) can be employed.

Online estimation

In this section we show how ϕ_t can be incrementally updated in a streaming setting. We also briefly discuss how to update ρ_t and ϕ_t.

LoCo score The eigenvector estimates of the exponential window local auto-covariance matrix can be updated incrementally using SPIRIT algorithm described in Section 2.2. Recall SPIRIT incrementally monitors the top left singular vectors which are the eigenvectors for symmetric positive definite matrices such as auto-covariance matrix.

Local Pearson score Updating the Pearson score ρ_t requires an update of the inner product and norms. For the former, this can be done using the simple relationship $\mathbf{x}_{t,w}^{\mathsf{T}}\mathbf{y}_{t,w} = \mathbf{x}_{t-1,w}^{\mathsf{T}}\mathbf{y}_{t-1,w} - x_{t-1}y_{t-1} + x_{t+w-1}y_{t+w-1}$. Similar simple relationships hold for $\|\mathbf{x}_{t,w}\|$ and $\|\mathbf{y}_{t,w}\|$.

2.4.3 Complexity

The time and space complexity of each method is summarized in Table 2.4. Updating ρ_t which requires $O(1)$ time (adding $x_{t+w-1}y_{t+w-1}$ and subtracting $x_{t-1}y_{t-1}$) and also buffering w values.

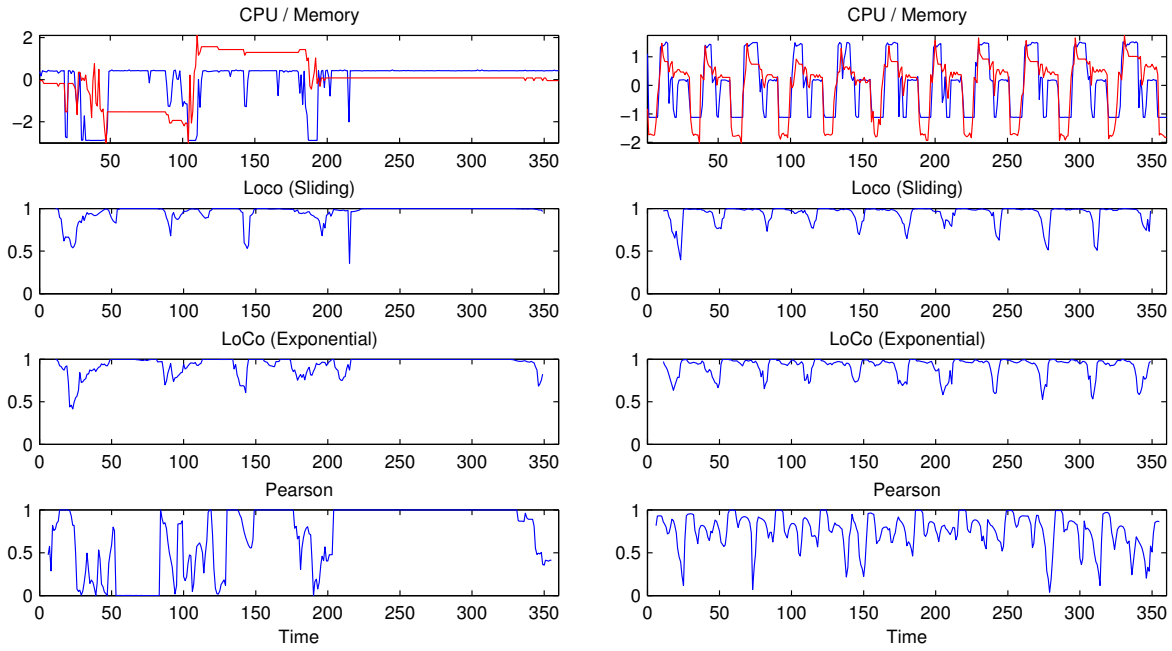
Estimating the LoCo score ℓ_t using a sliding window requires $O(wmk) = O(w^2k)$ time (since we set $m = w$) to compute the largest k eigenvectors of the covariance matrix for m windows of size w. We also need $O(wk)$ space for these k eigenvectors and $O(w +$

m) space for the series values, for a total of $O(wk + m) = O(wk)$. Using an exponential window still requires storing the $w \times k$ matrix $\mathbf{V}$, so the space is again $O(wk)$. However, the eigenspace estimate $\mathbf{V}$ can be updated in $O(wk)$ time, instead of $O(w^2k)$ for sliding window.

Method	Time (per point)	Space (total)
Pearson	$O(1)$	$O(w)$
LoCo sliding	$O(wmk)$	$O(wk + m)$
LoCo exponential	$O(wk)$	$O(wk)$

Table 2.4: Time and space complexity.

2.4.4 Experiments



(a) `MemCPU1` (b) `MemCPU2`

Figure 2.17: Local correlation scores, machine cluster.

This section presents our experimental evaluation, with the following main goals:

1. Illustration of LoCo on real time series.

2. Comparison to alternatives.

3. Demonstration of LoCo's robustness.

4. Comparison of exponential and sliding windows for LoCo score estimation.

5. Evaluation of LoCo's efficiency in a streaming setting.

Datasets The first two datasets, `MemCPU1` and `MemCPU2` were collected from a set of Linux machines. They measure total free memory and idle CPU percentages, at 16 second intervals. Each pair comes from different machines, running different applications, but the series within each pair are from the same machine. The last dataset, `ExRates`, was obtained from the UCR TSDMA [82]. and consists of daily foreign currency exchange rates, measured on working days (5 measurements per week) for a total period of about 10 years. Although the order is irrelevant for the scores since they are symmetric, the first series is always in blue and the second in red. For LoCo with sliding window we use exact, batch SVD on the sample set of windows—we do not explicitly construct $\hat{\Gamma}_t$. For exponential window LoCo, we use the incremental eigenspace tracking procedure. The raw scores are shown, without any smoothing, scaling or post-processing of any kind.

1. Qualitative interpretation We should first point out that, although each score has one value per time instant $t \in \mathbb{N}$, these values should be interpreted as the similarity of a "neighborhood" or window around t (Figure 2.17 and 2.18). All scores are plotted so that each neighborhood is centered around t. The window size for `MemCPU1` and `MemCPU2` is $w = 11$ (about 3 minutes) and for `ExRates` it is $w = 20$ (4 weeks). Next, we discuss the LoCo scores for each dataset.

Machine data Figure 2.17[a] shows the first set of machine measurements, `MemCPU1`. At time $t \approx 20$–50 one series fluctuates (oscillatory patterns for CPU), while the other remains constant after a sharp linear drop (aperiodic patterns for memory). This discrepancy is captured by ℓ_t, which gradually returns to one as both series approach constant-valued intervals. The situation at $t \approx 185$–195 is similar. At $t \approx 100$–110, both resources exhibit large changes (aperiodic trends) that are not perfectly synchronized. This is reflected by ℓ_t, which exhibits three dips, corresponding to the first drop in CPU, followed by a jump in memory and then a jump in CPU. Toward the end of the series, both resources are fairly

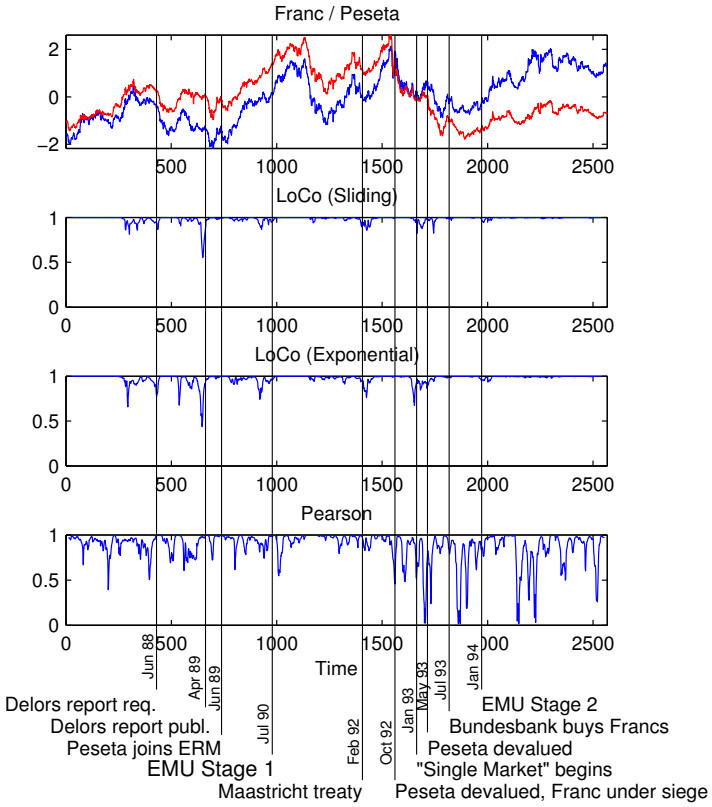


Figure 2.18: Local correlation scores, `ExRates`.

constant (but, at times, CPU utilization fluctuates slightly, which affects ρ_t and ϕ_t). In summary, ℓ_t behaves well across a wide range of joint patterns.

The second set of machine measurements, `MemCPU2`, is shown in Figure 2.17[b]. Unlike `MemCPU1`, memory and CPU utilization follow each other, exhibiting a very similar periodic pattern, with a period of about 30 values or 8 minutes. This is reflected by the LoCo score, which is mostly one. However, about in the middle of each period, CPU utilization drops for about 45 seconds, without a corresponding change in memory. At precisely those instants, the LoCo score also drops (in proportion to the discrepancy), clearly indicating the break of the otherwise strong correlation.

Exchange rate data Figure 2.18 shows the exchange rate (`ExRates`) data. The blue line is the French Franc and the red line is the Spanish Peseta. The plot is annotated with

an approximate timeline of major events in the European Monetary Union (EMU). Even though one should always be very careful in suggesting any causality, it is still remarkable that most major EMU events are closely accompanied by a break in the correlation as measured by LoCo, and vice versa. Even in the cases when an accompanying break is absent, it often turns out that at those events both currencies received similar pressures (thus leading to similar trends, such as, e.g., in the October 1992 events). It is also interesting to point out that events related to anticipated regulatory changes are typically *preceded* by correlation breaks. After regulations are in effect, ℓ_t returns to one. Furthermore, after the second stage of the EMU, both currencies proceed in lockstep, with negligible discrepancies.

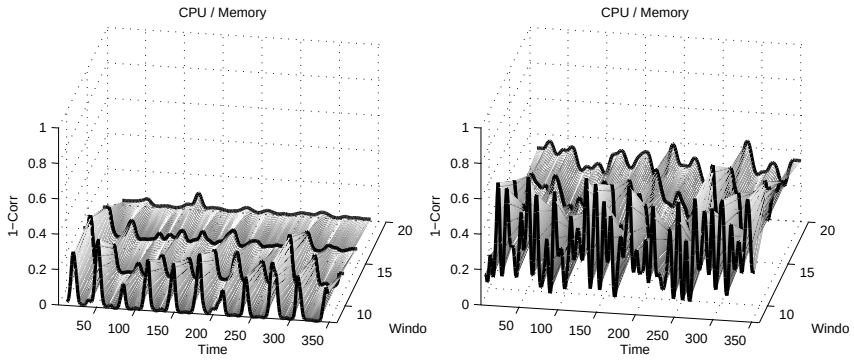
In summary, the LoCo score successfully and accurately tracks evolving local correlations, even when the series are widely different in nature.

2. LoCo versus Pearson Figure 2.17 and 2.18 also show the local Pearson score (fourth row), along with the LoCo score. It is clear that it either fails to capture changes in the joint patterns among the two series, or exhibit high sensitivity to small transients. We also tried using a window size of $2w - 1$ instead of w for ρ_t (so as to include the same number of points as ℓ_t in the "comparison" of the two series). The results thus obtained where slightly different but similar, especially in terms of sensitivity and lack of accurate tracking of the evolving relationships among the series.

3. Robustness This brings us to the next point in our discussion, the robustness of LoCo. We measure the "stability" of any score $c_t, t \in \mathbb{N}$ by its smoothness. We employ a common measure of smoothness, the (discrete) total variation V of c_t, defined as $V(c_t) := \sum_\tau |c_{\tau+1} - c_\tau|$, which is the total "vertical length" of the score curve. Table 2.5 (top) shows the total variation. If we scale the total variations with respect to the range (i.e., use $V(c_t)/R(c_t)$ instead of just $V(c_t)$—which reflects how many times the vertical length "wraps around" its full vertical range), then Pearson's variation is consistently about 3 times larger, over all data sets.

	Dataset		
Method	`MemCPU1`	`MemCPU2`	`ExRates`
Pearson	23.75	35.38	39.56
LoCo	5.71	10.53	6.37

Table 2.5: Relative stability (total variation)



(a) LoCo (b) Pearson

Figure 2.19: Score vs. window size; LoCo is robust with respect to both time and scale, accurately tracking correlations at any scale, while Pearson performs poorly at all scales.

Window size Figure 2.19(a) shows the LoCo scores of `MemCPU2` (see Figure 2.17(b)) for various windows w, in the range of 8–20 values (2–5 minutes). We chose the dataset with the highest total score variation and, for visual clarity, Figure 2.19 shows $1 - \ell_t$ instead of ℓ_t. As expected, ℓ_t varies smoothly with respect to w. Furthermore, it is worth pointing out that at about a 35-value (10-minute) resolution (or coarser), both series the exhibit clearly the same behavior (a periodic increase then decrease, with a period of about 10 minutes—see Figure 2.17(b)), hence they are perfectly correlated and their LoCo score is almost constantly one (but not their Pearson score, which gets closer to one while still fluctuating noticeably). Only at much coarser resolutions (e.g., an hour or more) do both scores become one. This convergence to one is not generally the case and some time series may exhibit interesting relationships at all time scales. However, the LoCo score is robust and changes gracefully also with respect to resolution/scale, while accurately capturing any interesting relationship changes that may be present at any scale.

Dataset	`MemCPU1`	`MemCPU2`	`ExRates`
Avg. var.	0.051	0.071	0.013
Rel. var.	5.6%	7.8%	1.6%

Table 2.6: Sliding vs. exponential score.

4. Exponential vs. sliding window Figure 2.17 and 2.18 show the LoCo scores based upon both sliding (second row) and exponential (third row) windows, computed using

appropriately chosen equivalent window sizes. Upon inspection, it is clear that both LoCo score estimates are remarkably close. In order to further quantify this similarity, we show the average variation $\hat{V}$ of the two scores, which is defined as $\hat{V}(\ell_t, \ell'_t) := \frac{1}{t}\sum_{\tau=1}^{t} |\ell_\tau - \ell'_\tau|$, where ℓ_t uses exact, batch SVD on sliding windows and ℓ'_t uses eigenspace tracking on exponential windows. Table 2.6 shows the average score variations for each dataset, which are remarkably small, even when compared to the mean score $\hat{\ell} := \frac{1}{t}\sum_{\tau=1}^{t} \ell_\tau$ (the bottom line in the table is $\hat{V}/\hat{\ell}$).

2.4.5 Summary

Time series correlation or similarity scores are useful in several applications. Beyond global scores, in the context of data streams it is desirable to track a time-evolving correlation score that captures their changing similarity. We propose such a measure, the Local Correlation (LoCo) score. It is based on a joint model of the series which, naturally, does not make any assumptions about stationarity. The model may be viewed as a generalization of simple linear cross-correlation (which it includes as a special case). The score is robust to transients, while accurately tracking the time-varying relationships among the series. Furthermore, it lends itself to efficient estimation in a streaming setting. We demonstrate its qualitative interpretation on real datasets, as well as its robustness and efficiency.

2.5 Privacy Preservation on streams

How to preserve privacy in data streams? There has been an increasing concern regarding privacy breaches, especially those involving sensitive personal data of individuals [58]. Meanwhile, unprecedented massive data from various sources are providing us with great opportunity for data mining and information integration. Unfortunately, the privacy requirement and data mining applications pose exactly opposite expectations from data publishing [58]. The utility of the published data w.r.t the mining application decreases with increasing levels of privacy guarantees [84].

Guaranteeing data privacy is especially challenging in the case of stream data, mainly for two reasons:

1. **Performance requirement:** The continuous arrival of new tuples prohibits storage of the entire stream for analysis, rendering the current offline algorithms inapplicable.

2. **Time evolution:** Data streams are usually evolving, and correlations and autocorrelations can change over time. These characteristics make most offline algorithms for static data inappropriate, as we show later.

Our goal is to insert random perturbation that "mirrors" the original streams' statistical properties, in an online fashion. Thus, a number of important mining operations can still be performed, by controlling perturbation magnitude. However, the original data streams cannot be reconstructed with high confidence.

Our contributions are: 1) we define the notion of utility and privacy for perturbed data streams, 2) we explore the effect of evolving correlations and autocorrelation in data streams, and their implications in designing additive random perturbation techniques, 3) we design efficient online algorithms under the additive random perturbation framework, which maximally preserve the privacy of data streams given a fixed utility while, additionally, better preserving the statistical properties of the data, and 4) we provide both theoretical arguments and experimental evaluation to validate our ideas.

2.5.1 Problem Formulation

To ensure privacy of streaming data, the values of incoming tuples are modified by adding noise. We denote the original data as $\mathbf{A} \in \mathbb{R}^{N \times T}$, the random noise as $\mathbf{E} \in \mathbb{R}^{N \times T}$ where each entry e_t^i is the noise added to the i-th stream at time t. Therefore, the perturbed streams are $\mathbf{A}^* = \mathbf{A} + \mathbf{E}$. Without loss of generality, we assume the noise has zero mean.

Discrepancy:. To facilitate the discussion on utility and privacy, we define the concept of *discrepancy* $\mathcal{D}$ between two versions of the data, $\mathbf{A}$ and $\mathbf{B}$, as the normalized squared Frobenius norm[6],

$$\mathcal{D}(\mathbf{A}, \mathbf{B}) := \frac{1}{T}\|\mathbf{A} - \mathbf{B}\|_F^2, \quad \text{where } \mathbf{A}, \mathbf{B} \in \mathbb{R}^{N \times T}.$$

Utility:. Considering the perturbed versus the original data, the larger the amplitude of the perturbation (i.e., the variance of the added noise), the less useful the data become. Therefore, we define the utility to be the inverse of this discrepancy. However, throughout the section, we typically use discrepancy, since the two concepts are essentially interchangeable.

[6]The squared Frobenius norm is defined as $\|\mathbf{A}\|_F^2 := \sum_{i,j}(a_i^j)^2$

Privacy:. Distorting the original values is only one of the challenges. We also have to make sure that this distortion cannot be filtered out. Thus, to measure the privacy, we have to consider the power of an adversary in reconstructing the original data. Specifically, suppose that $\tilde{\mathbf{A}}$ are the reconstructed data streams obtained by the adversary, in a way that will be formalized shortly. Then the privacy is the discrepancy between the original and the reconstructed streams, i.e., $\mathcal{D}(\mathbf{A}, \tilde{\mathbf{A}})$.

We formulate two problems: data reconstruction and data perturbation. The adversary wants to recover the original streams from the perturbed data. Conversely, data owners want to prevent the reconstruction from happening.

Problem 2.1 (Reconstruction)**.** *Given the perturbed streams $\mathbf{A}^*$, how to compute the reconstruction streams $\tilde{\mathbf{A}}$ such that $\mathcal{D}(\mathbf{A}, \tilde{\mathbf{A}})$ is minimized?*

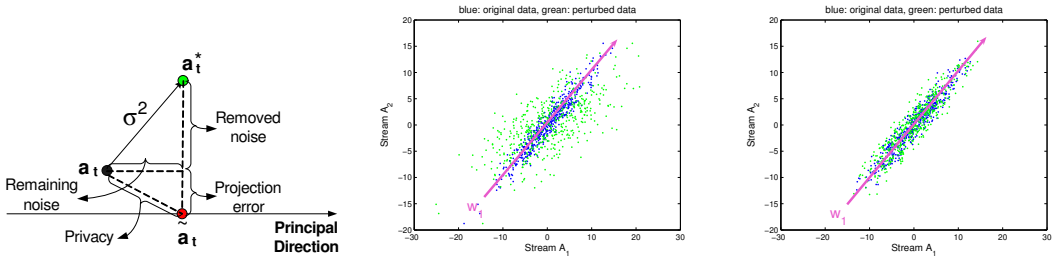
We focus on linear reconstruction methods which have been used by many existing works [81, 74, 96, 31]. Intuitively, the adversary can only use linear transformations on the perturbed data, such as projections and rotations, in the reconstruction step.

Definition 2.7 (Linear reconstruction)**.** *Given the perturbed streams $\mathbf{A}^*$, the linear reconstruction is $\tilde{\mathbf{A}} = \mathbf{R}\mathbf{A}^*$, such that $\mathcal{D}(\mathbf{A}, \tilde{\mathbf{A}})$ is minimized.*

If both the perturbed streams $\mathbf{A}^*$ and the original streams $\mathbf{A}$ are available, the solution $\tilde{\mathbf{A}}$ can be easily identified using linear regression. However, $\mathbf{A}$ is not available. Therefore, in order to estimate $\tilde{\mathbf{A}}$, some additional constraints or assumptions must be imposed to make the problem solvable. A widely adopted assumption [80] is that the data lie in a static low dimensional subspace (i.e, global correlation exists). This is reasonable, since if no correlations are present, then i.i.d. perturbations are already sufficient to effectively hide the data. However, real data typically exhibit such correlations. As we will formally show later, we rely on the dynamic (rather than static) correlations among streams, as well as on dynamic autocorrelations.

Problem 2.2 (Perturbation)**.** *Given the original streams $\mathbf{A}$ and the desirable discrepancy threshold σ^2, how to obtain the perturbed streams $\mathbf{A}^*$ such that 1) $\mathcal{D}(\mathbf{A}, \mathbf{A}^*) = \sigma^2$ and 2) for any linear reconstruction $\tilde{\mathbf{A}}$, $\mathcal{D}(\mathbf{A}, \tilde{\mathbf{A}}) \geq \sigma^2$.*

Perturbation has exactly the opposite goal from the *reconstruction*. However, the correlation and autocorrelation properties of the streams are still the keys to the solution of both problems.



(a) noise decomposition (b) i.i.d random noise (c) correlated noise

Figure 2.20: Impact of Correlation on Perturbing the Data

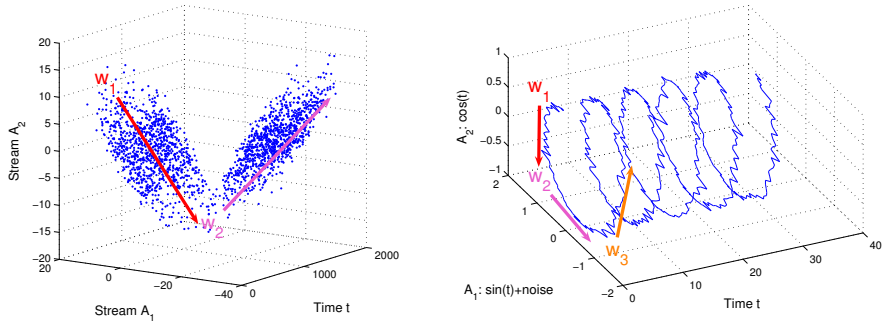
2.5.2 Privacy with Dynamic Correlations

Let us first illustrate how the perturbation and reconstruction work in detail (see Figure 2.20(a) for the visualization). For the perturbation process, the stream measurements $\mathbf{a}_t$ at time t, represented as an N-dimensional vector, are mapped to the perturbed vector $\mathbf{a}_t^*$ with additive Gaussian noise with zero-mean and variance σ^2. For any reconstruction effort, the goal is to transform the perturbed measurements, $\mathbf{a}_t^*$ onto $\tilde{\mathbf{a}}_t$ so that $\mathcal{D}(\mathbf{a}_t, \tilde{\mathbf{a}}_t)$ is small.

A principled way of reconstruction is to project the data onto the principal component subspace [74] such that most noise is removed, while the original data are maximally preserved, i.e, not much additional error is included. The idea is illustrated in Figure 2.20(a). When $\mathbf{a}_t^*$ is projected onto the principal direction, the projection is exactly the reconstruction $\tilde{\mathbf{a}}_t$. Note that the distance between $\mathbf{a}_t^*$ and $\tilde{\mathbf{a}}_t$ consists of two parts: 1) *removed noise*, i.e., the perturbation that is removed by the reconstruction and 2) *projection error*, i.e, the new error introduced by the reconstruction. Finally, the distance between $\mathbf{a}_t$ and $\tilde{\mathbf{a}}_t$, i.e., the privacy, comes from two sides: 1) *remaining noise*, i.e, the perturbation noise that has not been removed, and 2) *projection error*.

When the noise is added exactly along the principal direction, the *removed noise* becomes zero. However, additional *projection error* is included. In this case, the perturbation is robust towards this reconstruction attempt, in the sense that $\mathcal{D}(\mathbf{A}, \tilde{\mathbf{A}}) \geq \mathcal{D}(\mathbf{A}, \mathbf{A}^*)$. In general, a good practice is to add correlated noise following the trends present in the streams. Consider the example shown in Figure 2.20 where blue points represent the original data and green points represent the perturbed data with same amount of noise. Figure 2.20(b) and 2.20(c) show the i.i.d. noise and the correlated noise on the same data, respectively. Clearly, correlated noise has been successfully "hidden" in the original data, and therefore, is hard to remove.

Data streams often present strong correlations and these correlations change dynami-



(a) Example one. (b) Example two.

Figure 2.21: Dynamic correlations in Data Streams

cally. Consider the examples in Figure 2.21, where the principal components can change over time. In such cases, online PCA is better for characterizing the evolving, local trends. Global, offline PCA will fail to identify these important properties. Next, we will show how to dynamically insert noise using SPIRIT.

Algorithm 2.6: STREAMING CORRELATION ADDITIVE NOISE (SCAN)

Input : Original tuple $\mathbf{a}_t$, utility threshold σ^2
Old subspace $\mathbf{U} \in \mathbb{R}^{N \times k}$, $\mathbf{\Lambda} \in \mathbb{R}^{k \times k}$
Output: Perturbed tuple $\mathbf{a}^*_t$, new subspace $\mathbf{U}$, $\mathbf{\Lambda}$

Update left-singular vectors $\mathbf{U}$, singular values Λ based on $\mathbf{a}_t$ using SPIRIT
Initialize δ, η to $\mathbf{0}_k$
//add noise in top-k principal component subspace
for $1 \leq i \leq k$ **do**
    δ(i)=$\sigma^2 \times \frac{\Lambda(i)^2}{\|(\Lambda)\|_F^2}$ where $\Lambda(i)$ is the i-th singular value
    $\eta(i) =$ Gaussian noise with mean zero and variance δ(i)
// rotation back to the original space
$\mathbf{E}_t$=$\mathbf{U}\eta$ and $\mathbf{A}^*_t$=$\mathbf{A}_t$+$\mathbf{E}_t$

Streaming Correlated Additive Noise (SCAN). SCAN does two things whenever new tuples arrive from the N input streams: 1) it updates the estimation of local principal components; and 2) it distributes the noise along the principal components in proportional to their singular values dynamically. In short, SCAN tracks the covariance matrix and adds noise with essentially the same covariance as the data streams.

Theorem 2.3. *At any time instant T, the perturbed data streams $\mathbf{A}^*$ from SCAN satisfy $\mathcal{D}(\mathbf{A}, \mathbf{A}^*) = \sigma^2$.*

Proof.

$$\begin{aligned}\mathcal{D}(\mathbf{A}, \mathbf{A}^*) &= \frac{1}{T}\sum_{i=1}^{T} \|\mathbf{a}_i - \tilde{\mathbf{a}}_i\|^2 \\ &= \frac{1}{T}\sum_{i=1}^{T} \sigma^2 \\ &= \sigma^2\end{aligned}$$

where $\mathbf{a}_i$ ($\tilde{\mathbf{a}}_i$) is the i-th row in $\mathbf{A}$ ($\mathbf{A}^*$). □

Streaming correlation online reconstruction (SCOR):. The privacy achieved by SCAN is determined by the best linear reconstruction an adversary could perform on $\mathbf{A}^*$. For evolving data streams as illustrated in Figure 2.21, the best choice for the adversary is to utilize online estimation of local principal components for reconstruction.

Intuitively, SCOR reconstruction removes all the noise orthogonal to the local principal components and inserts little additional projection error, since local PCA can usually track the data accurately.

Theorem 2.4. *The reconstruction error of $SCOR$ on the perturbation from $SCAN$ is $\approx \sigma^2$.*

Proof. Formally, given a linear reconstruction $\tilde{\mathbf{A}} = \mathbf{A}^*\mathbf{R}$, the privacy can be decomposed

as

$$
\begin{aligned}
\mathcal{D}(\mathbf{A}, \tilde{\mathbf{A}}) &= \frac{1}{T}\|\mathbf{A} - \mathbf{R}\mathbf{A}^*\|_F^2 \\
&= \frac{1}{T}\|\mathbf{A} - \mathbf{R}(\mathbf{A} + \mathbf{E})\|_F^2 \\
&= \frac{1}{T}\|(\mathbf{I} - \mathbf{R})\mathbf{A} + \mathbf{R}\mathbf{E}\|_F^2. \\
&= \frac{1}{T}\| \underbrace{(\mathbf{I} - \mathbf{U}\mathbf{U}^\mathsf{T})\mathbf{A}}_{\text{projection error}} + \underbrace{\mathbf{U}\mathbf{U}^\mathsf{T}\mathbf{E}}_{\text{remaining error}} \|_F^2 \\
&\approx \frac{1}{T}\|\mathbf{U}\mathbf{U}^\mathsf{T}\mathbf{E}\|_F^2 && \text{// projection error is small} \\
&= \frac{1}{T}\|\mathbf{E}\|_F^2 && \text{// } \mathbf{E} \text{ is in the subspace spanned by } \mathbf{U} \\
&= \sigma^2
\end{aligned}
$$

where $\mathbf{R}$ is a projection matrix, i.e., $\mathbf{R} = \mathbf{U}\mathbf{U}^\mathsf{T}$ with $\mathbf{U} \in \mathbb{R}^{N \times k}$ orthonormal. Since the subspaces tracked by both SCOR and SCAN are the same, the remaining noise is σ^2, i.e., no noise is removed. Therefore, $\mathcal{D}(\mathbf{A}, \tilde{\mathbf{A}}) \approx \sigma^2$. □

Note that the projection error for SCOR is small, provided that the data are locally correlated. Therefore, the reconstruction error (i.e., privacy, as defined in Section 2.5.1) of SCOR is approximately σ^2, i.e., equal to the original discrepancy.

Algorithm 2.7: STREAMING CORRELATION ONLINE RECONSTRUCTION (SCOR)

Input : Perturbed tuple $\mathbf{a}_t^*$, utility threshold σ^2
Old subspace $\mathbf{U} \in \mathbb{R}^{N \times k}$, $\mathbf{\Lambda} \in \mathbb{R}^{k \times k}$
Output: Perturbed tuple $\tilde{\mathbf{a}}_t$, new subspace $\mathbf{U}, \mathbf{\Lambda}$

1 Update left-singular vectors $\mathbf{U}$, singular values Λ based on $\mathbf{a}_t$)
2 //project to the estimated online principal components $\tilde{\mathbf{a}}_t = \mathbf{U}\mathbf{U}^\mathsf{T}\mathbf{a}_t^*$

2.5.3 Privacy with Dynamic Autocorrelations

The noise added should mirror the dominant trends in the series. Consider the following simple examples: If the stream always has a constant value, the right way to hide this

value is to add the same noise throughout time. Any other noise can be easily filtered out by simple averaging. The situation is similar for a linear trend.If the stream is a sine wave, the right way to hide it is by adding noise with the same frequency (but potentially a different phase); anything else can be filtered out. Our algorithm is the generalization, in a principled manner, of these notions.

For example, the green and blue curves in Figure 2.22(b) are the autocorrelated noise and the original stream, respectively, where the noise follows the same trends as the streams, over time. In comparison, Figure 2.22(a) shows i.i.d. noise, which can be easily filtered out. The goal is to find a principled way to automatically determine what is the "right" noise, which is "most similar" to the stream.

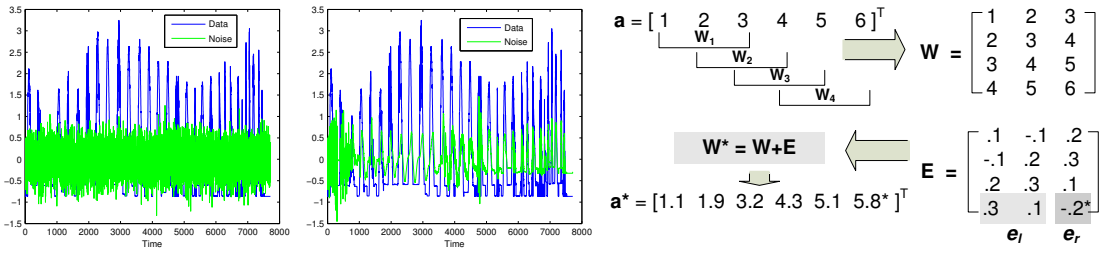
Connection to correlation:. In the previous section, we showed how to track the local statistical properties of the N-dimensional sequence of the vectors $\mathbf{a}_t$, indexed over time t. More specifically, we track the principal subspace of this matrix, thereby focusing on the most dominant (in a least-squares sense) of these relationships. We subsequently add noise that "mirrors" those relationships, making it indistinguishable from the original data.

Next, we will show that the same principles used to capture relationships across many attributes can be used to capture relationships of one attribute across time. In fact, there is a natural way to move between the original time domain and a high-dimensional sequence space, which is formalized next. The t-th *window* of the time series stream $\mathbf{a}(t)$ is an h-dimensional point,

$$\mathbf{W}_t := [\mathbf{a}(t), \mathbf{a}(t+1), \ldots, \mathbf{a}(t+h-1)]^\mathsf{T} \in \mathbb{R}^h.$$

The *window matrix* $\mathbf{W}$ has the windows $\mathbf{W}_t$ as rows. Thus, $\mathbf{W}_i^j = \mathbf{a}((i-1)h + j)$ by construction. The space spanned by the sequence of windows $\mathbf{W}_t$ is known as the h-th order *phase space* of the series $\mathbf{a}(t)$ [63]. Subsequently, we can essentially apply the same technique as before, using $\mathbf{W}$ in place of $\mathbf{A}$. All of the previous discussion and properties of our algorithm can be directly transferred to the autocorrelation case. An example is shown in the top of Figure 2.22(c). However, there are some additional properties and issues that need to be resolved.

Hankel Constraint:. Notice that the window matrix $\mathbf{W}$ is a *Hankel* matrix, i.e., the *anti*-diagonals are constants: $\mathbf{W}_i^j = \mathbf{W}_{i-1}^{j-1}$. Under the assumption that the series is stationary, the auto-covariance matrix $\mathbf{W}^T\mathbf{W}$ is, in expectation is *circulant*, i.e., it is symmetric with constant diagonals. Additionally, if we perform a batch eigen-analysis on the global window matrix of a static series, the sample auto-covariance matrix computed from the actual data (i.e., $\mathbf{W}^\mathsf{T}\mathbf{W}$ above) is also circulant. In this case, the eigenvectors of $\mathbf{W}^\mathsf{T}\mathbf{W}$ essentially provide the same information as the Fourier coefficients of the series $\mathbf{a}$. In that sense,



(a) iid noise (b) Autocorrelated noise (c) Streaming autocorrelation additive noise

Figure 2.22: Dynamic Autocorrelation

our approach includes traditional Fourier analysis. If these assumptions do not hold, the technique we employ is more robust and effective.

Constraint on autocorrelated noise:. Next, we address the issues that arise from the fact that $\mathbf{W}$ is a Hankel matrix. Similarly, the noise matrix $\mathbf{E}$ has to be a Hankel matrix (see Figure 2.22(c) for an example). Similar to the correspondence between $\mathbf{a}$ and $\mathbf{W}$, the noise matrix $\mathbf{E}$ has a corresponding noise sequence $\mathbf{e}$, such that

$$\mathbf{E}_t \equiv [\mathbf{e}(t), \mathbf{e}(t+1), \ldots, \mathbf{e}(t+h-1)]^\mathsf{T} \in \mathbb{R}^h.$$

We will essentially use the same insight, that $\mathbf{E}_t$ has to lie in the subspace of $\mathbf{U}$, but in a different way. Formally stated, the residual $\mathbf{E}_t - \mathbf{U}\mathbf{U}^\mathsf{T}\mathbf{E}_t$ must be zero, or

$$(\mathbf{I} - \mathbf{U}\mathbf{U}^\mathsf{T})\mathbf{E}_t \equiv \mathbf{Q}\mathbf{E}_t = \mathbf{0}, \tag{2.3}$$

where $\mathbf{P} = \mathbf{U}\mathbf{U}^\mathsf{T}$ is the projection operator onto the subspace of $\mathbf{U}$ and $\mathbf{Q} = \mathbf{I} - \mathbf{P} = \mathbf{I} - \mathbf{U}\mathbf{U}^\mathsf{T}$ is the projector onto the orthogonal complement.

Assume that we have chosen the noise values up to time $t-k$. Based on these and on the current estimate of $\mathbf{U}$, we will determine the next k noise values (where k is the principal subspace dimension)—the reason for determining them simultaneously will become clear soon. Let

$$\begin{aligned}\mathbf{E}_{t-h+1} &\equiv [\underbrace{\mathbf{e}(t-h+1), \ldots, \mathbf{e}(t-k)}_{\text{known values}} \mid \underbrace{\mathbf{e}(t-k+1), \ldots, \mathbf{e}(t)}_{\text{unknown values}}]^\mathsf{T} \\ &\equiv [\mathbf{e}_l^\mathsf{T} \mid \mathbf{e}_r^\mathsf{T}]^\mathsf{T},\end{aligned}$$

where $\mid$ denotes element-wise concatenation (for example, $[1, 2 \mid 3, 4]$ results into a vector $[1, 2, 3, 4]$. The first block $\mathbf{e}_l \in \mathbb{R}^{h-k}$ consists of $h-k$ known values, whereas the second

Algorithm 2.8: Streaming Auto-Correlation Additive Noise(SACAN)

Input : Original value $\mathbf{a}^*(t)$, utility σ^2
Old subspace $\mathbf{U} \in \mathbb{R}^{h\times k}$, $\mathbf{\Lambda} \in \mathbb{R}^{k\times k}$
Output: Perturbed value $\mathbf{a}^*(t)$, new subspace $\mathbf{U}, \mathbf{\Lambda}$

Construct window $\mathbf{W}_{t-h+1} = [\mathbf{a}(t-h+1), \ldots, \mathbf{a}(t)]^\mathsf{T}$
Update $\mathbf{U}$, $\mathbf{V}$ using $\mathbf{W}_{t-h+1}$
every k arriving values **do**
    Let $[\mathbf{w}_l^\mathsf{T} \mid \mathbf{w}_r^\mathsf{T}]^\mathsf{T} \equiv \mathbf{W}_{t+h+1}$
    Solve equation 2.4 to obtain $\mathbf{e}_r$
    Rescale $\mathbf{e}_r$ based on σ^2
    Perturbed values $\mathbf{w}_r^* = \mathbf{w}_r + \mathbf{e}_r$
    Publish values $\mathbf{a}^*(t-k+i) = \mathbf{w}_r^*(i)$, $1 \leq i \leq k$

block $\mathbf{e}_r \in \mathbb{R}^k$ consists of the k unknown noise values we wish to determine. Similarly decomposing $\mathbf{Q} \equiv [\mathbf{Q}_l \mid \mathbf{Q}_r]$ into blocks $\mathbf{Q}_l \in \mathbb{R}^{h\times(h-k)}$ and $\mathbf{Q}_r \in \mathbb{R}^{h\times k}$, we can rewrite equation 2.3 as

$$\mathbf{Q}_l\mathbf{e}_l + \mathbf{Q}_r\mathbf{e}_r = \mathbf{0} \quad \text{or} \quad \mathbf{Q}_r\mathbf{e}_r = -\mathbf{Q}_l\mathbf{e}_l. \tag{2.4}$$

This is a linear equation system with k variables and k unknowns. Since the principal subspace has dimension k by construction, the linear system is full-rank and can always be solved. The bottom right of Figure 2.22(c) highlights the known $\mathbf{e}_l$ and unknown $\mathbf{e}_r$ (with one principle component $k = 1$).

The above equation cannot be applied for initial values of the noise; we will use i.i.d. noise for those. Initially, we do not know anything about the patterns present in the signal, therefore i.i.d. noise is the best choice, since there are no correlations yet. However, the adversary has also not observed any correlations that can be leveraged to remove that noise. The important point is that, as soon as correlations become present, our method will learn them and use them to intelligently add the noise, before the adversary can exploit this information.

Figure 2.22(b) and 2.24 show that our approach accurately tracks the dominant local trends, over a wide range of stream characteristics. Algorithm 2.8 shows the pseudocode. The algorithm for reconstructing the original data is simpler; we only need to project each window $\mathbf{W}_t$ onto the current estimate of $\mathbf{U}$, exactly as we did for the correlation case. The pseudocode is shown in Algorithm 2.9.

Preserving the autocorrelation properties, in addition to the privacy, is desirable, since several fundamental mining operations, such as autoregressive modeling and forecasting

Algorithm 2.9: Streaming Auto-Correlation Online Reconstruction (SACOR)

Input : Perturbed value $\mathbf{a}^*(t)$
Old subspace $\mathbf{U} \in \mathbb{R}^{N \times k}$, $\mathbf{\Lambda} \in \mathbb{R}^{k \times k}$
Output: Reconstruction $\tilde{\mathbf{a}}(t)$, new subspace $\mathbf{U}, \mathbf{\Lambda}$

1 Construct window $\mathbf{W}_{t-h+1} = [\mathbf{a}(t-h+1), \ldots, \mathbf{a}(t)]^\mathsf{T}$
2 Update $\mathbf{U}$, $\mathbf{\Lambda}$ using SPIRIT by $\mathbf{W}_{t-h+1}$
3 Project onto est. eigenspace $\tilde{\mathbf{W}} = \mathbf{U}\mathbf{U}^\mathsf{T}\mathbf{W}_{t-h+1}$
4 Reconstruction is the last element of $\tilde{\mathbf{W}}$, $\tilde{\mathbf{a}}(t) = \tilde{\mathbf{W}}_t^h$

as well as periodicity detection [22], rely on them.

2.5.4 Experiments

We have implemented the proposed algorithms and study their performance on real data streams. Specifically, we show that: 1) in terms of preserving the input streams' privacy, SCAN and SACAN outperform both i.i.d. noise as well as noise added based on offline analysis; 2) SCOR and SACOR achieve smaller reconstruction error than static, offline algorithms; 3) all proposed algorithms have considerably small computation and memory overhead.

Data Streams	Dimension	Description
Chlorine [57]	4310×166	Environmental sensors
Lab [45]	7712×198	Room sensors
Stock [78]	8000×2	Stock price

Table 2.7: Three Real Data Sets

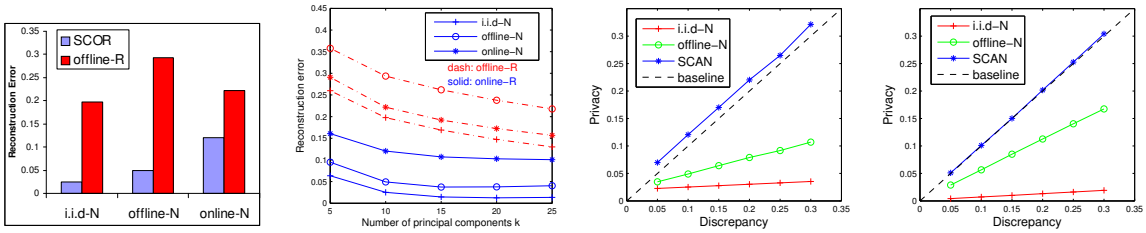
Experiment Setup

The real-world data sets we use are summarized in table 2.7. "Chlorine" measures water quality in a drinking water distribution system, and "Lab" measures light, humidity, temperature and voltage of sensors in the Intel Research Berkeley lab, which has been used in Section 2.2 and Section 2.3.

For simplicity, both discrepancy and reconstruction error are always expressed relative to the energy of the original streams, i.e., $\mathcal{D}(\mathbf{A}, \mathbf{A}^*)/\|\mathbf{A}\|_F^2$ and $\mathcal{D}(\mathbf{A}, \tilde{\mathbf{A}})/\|\mathbf{A}\|_F^2$, respec-

tively. Equivalently, the streams are normalized to zero mean and unit variance, which does not change their correlation or autocorrelation properties. The random noise distribution is zero mean Gaussian, with variance determined by the discrepancy parameter. Maximum discrepancy is 0.3, as large noise will destroy the utility of the perturbed data, making them practically useless for the mining application. Without loss of generality and to facilitate presentation, we assume that perturbation and reconstruction use the same number of principal components. Our prototype is implemented in Matlab and all experiments are performed on an Intel P4 2.0GHz CPU.

Dynamic Correlation



(a) Reconstruction Error: SCOR vs. offline-R

(b) Reconstruction Error: vary k

(c) Privacy vs. Discrepancy, online-R: Lab data set

(d) Privacy vs. Discrepancy, online-R: Chlorine data set

Figure 2.23: Privacy Preservation for Streams with Dynamic Correlations

The perturbation and reconstruction methods investigated in our experiments are summarized in Table 2.8, where "N" stands for noise and "R" for reconstruction. The offline algorithms, for both perturbation and reconstruction, are essentially the existing work on the static, relational data model, using PCA on the entire stream history to identify correlations and add or remove noise accordingly. Except otherwise specified, we set the number of principal components k to 10. Although the offline algorithms may not be applicable in a streaming setting due to large storage requirements, they are included for comparison and we show that, besides high overheads, their performance is sub-optimal due to the

Perturbation	i.i.d-N	offline-N	online-N:SCAN
Reconstruction	baseline	offline-R	online-R:SCOR

Table 2.8: Perturbation/Reconstruction Method

time-evolving nature of streams. Finally, baseline reconstruction is simply the perturbed data themselves (i.e., no attempt to recover the original data).

Reconstruction Error. Figure 2.23(a) shows the reconstruction error of online and offline reconstruction, w.r.t all types of noise. The figure presents results from Lab data with discrepancy is set to 10%. In all cases, SCOR clearly outperforms the offline method. The main reason is that offline-R has considerable projection error for streams with dynamically evolving correlation, whereas SCOR has almost negligible projection error and its reconstruction error is dominated by the remaining noise. Similar phenomena were observed for other discrepancy values. Therefore, online reconstruction should be the candidate for measuring the privacy of the perturbed data.

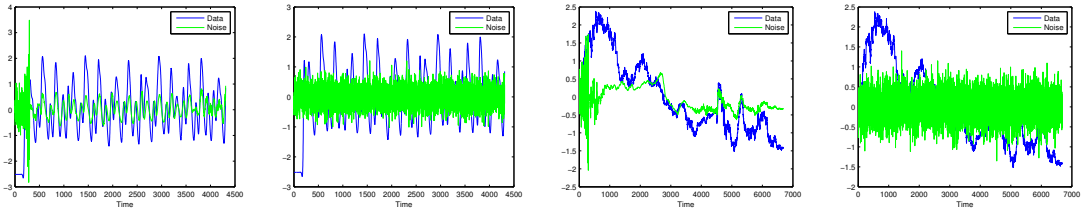
Effect of k. The number of principal components k will affect both the projection error and the remaining noise, which in turn have an impact on the overall reconstruction error. Figure 2.23(b) studies the effect of k on both offline-R and online-R on the Lab data with discrepancy fixed at 10%. For both approaches, reconstruction errors decrease as k increases. There are two interesting observations: First, online-R requires smaller k to reach a "flat," stable reconstruction error. This is beneficial since both the computation and memory cost increase in proportion to k, due to the complexity of the core algorithm (SPIRIT) described in Section 2.2. Second, online-R achieves smaller reconstruction error than offline-R, for all types of noise.

Perturbation Performance. Next, we measure the ability of different perturbation methods to preserve privacy of data streams with dynamic correlations. Results on the Lab and Chlorine data are presented in Figure 2.23(c) and 2.23(d). Clearly, for both data streams, SCAN achieves the best privacy over all discrepancy values, i.e., SCAN is the only one above baseline in the curve meaning no privacy is compromised. Therefore, SCAN effectively achieves the best privacy w.r.t. the allowed discrepancy.

Dynamic Autocorrelation

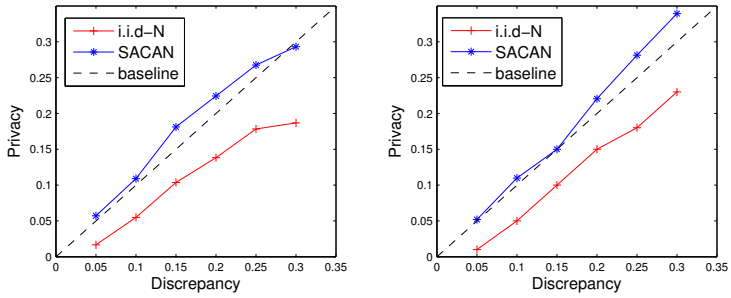
This section demonstrates the correctness and effectiveness of our algorithms for data perturbation in streams with autocorrelation. Except otherwise specified, the window size h is set to 300 and the number of principal components k is 10. We compare our method against i.i.d. noise and we use the online reconstruction algorithm SACOR, in order to measure privacy.

Effectiveness of SACAN. Figure 2.24 shows the results on the Chlorine and Stock data sets. The discrepancy is set to $.1$ for all experiments. We observe from Figure 2.24(a)



(a) Chlorine: SACAN (b) Chlorine: i.i.d (c) Stock: SACAN (d) Stock: i.i.d.

Figure 2.24: Online Random Noise for Stream with Autocorrelation



(a) Chlorine (b) Stock

Figure 2.25: Privacy vs. Discrepancy: Online Reconstruction using Autocorrelation

and 2.24(c) that SACAN initially produces i.i.d. noise but it is quickly able to estimate and track the autocorrelation of the input stream. Hence, SACAN adds random noise that closely follows the estimated autocorrelation. Intuitively, the noise generated by SACAN exhibits: 1) the same frequency as the input data stream; 2.) amplitude that is determined by the discrepancy. Thus, since the SACAN perturbation follows the same trend as the input stream, it is hardly distinguishable or separable once added. The advantage of SACAN becomes clear when comparing the results shown in Figure 2.24(b) (SACAN) and 2.24(d) (i.i.d noise).

Privacy of SACAN and Effectiveness of SACOR. Using SACOR for online reconstruction, more than $\frac{1}{3}$ of the i.i.d. noise added to the Chlorine and Stock streams can be removed. However, the noise produced by SACAN could not be removed at all. Figure 2.25 demonstrates these results, for varying discrepancy. The reconstruction error from SACOR is used to measure privacy. The baseline is the discrepancy between perturbed and input streams (i.e., no attempt at reconstruction). Since SACAN produces noise that follows the same trend as the input data stream, it is hard to remove any such noise from the perturbed data.

2.5.5 Summary

Data streams in the real world typically exhibit both significant correlations as well as autocorrelation, thereby providing ample opportunities for adversaries to breach privacy. We develop the basic building blocks for privacy preservation on numerical streams. In particular, we focus on the fundamental cases of correlation across multiple streams and of autocorrelation within one stream. We present methods to dynamically track both and subsequently add noise that "mirrors" these statistical properties, making it indistinguishable from the original data. Thus, our techniques prevent adversaries from leveraging these properties to remove the noise and thereby breach privacy. We provide both a mathematical analysis and experimental evaluation on real data to validate the correctness, efficiency, and effectiveness of our algorithms. Our techniques track the evolving nature of these relationships and achieve much better results than previous static, global approaches.

2.6 Chapter summary: stream mining

Data streams represented in first-order tensor streams have numerous applications in diverse domains, such as network forensics, sensor monitoring and financial applications. The key questions we addressed are the following: How to summarize the data streams? How to identify patterns in the data? The main challenges are the high dimensionality of the data and the streaming requirement in both speed and space.

To answer the questions, we first present a powerful technique SPIRIT to incrementally summarize the high dimensional streams. The SPIRIT algorithm has shown tremendous improvement in both speed and space to the existing method such as SVD. SPIRIT can also help find interesting patterns in the data: e.g., trends and anomalies in many different datasets. Second, we extend SPIRIT to work for distributed environments. We showed that the distributed SPIRIT performs similarly to the centralized SPIRIT but requires no centralized communication, therefore, it is more efficient and applicable to real scenarios. Third, we proposed a correlation function between streams, which can track complex correlation, is robust to transients and supports incremental computation. Finally, we developed a set of incremental algorithms to preserve the privacy in data streams. The SPIRIT algorithm is the driving force in many of the proposed algorithms, which confirms its great potential and applicability.

Next, we will see how to deal with second-order tensor stream, i.e., time-evolving graphs.

Chapter 3

Graph Mining

"How to find patterns such as anomalies, clusters in large graphs? How to summarize time-evolving graphs?"

Graphs arise naturally in a wide range of disciplines and application domains, since they capture the general notion of an association between two entities. Formally, we denote the graph as $G = (V, E)$, where V is the set of nodes and E the set of edges. In general, a graph $G = (V, E)$ can be represented as an adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$. Every row and column in $\mathbf{A}$ corresponds to one node in V. The entry-(i, j) of $\mathbf{A}$, denoted as a_{ij}, is not 0 if (i, j) is an edge in E. The value of a_{ij} indicates the weight of that edge. In general, we consider the graph G as the data model and $\mathbf{A}$ as the data representation. Time-evolving graphs are a sequence of graphs, one for each timestamp, or formally, $\{G_t | t = 1, 2, \ldots\}$. The corresponding representation is a sequence of adjacency matrices indexed by t, i.e., $\{\mathbf{A}_t | t = 1, 2, \ldots\}$, which is a second-order tensor stream.

The aspect of time has begun to receive some attention [91, 119]. Some examples of the time-evolving graphs include:

- *The Internet:* Network traffic events indicate ongoing communication between source and destination hosts;
- *World Wide Web:* The web can be viewed as a huge connected directed graph in which nodes are web-pages, and edges are hyperlinks. Both web-pages (nodes) and hyperlinks (edges) are changing over time.
- *Social Networks:* Email networks associate senders and recipients over time;

- *Communication networks:* Call detail records in telecommunications networks associate a caller with a callee. The set of all conversation pairs over each week forms a graph that evolves over time;
- *Financial data:* Transaction records in a financial institution, who accessed what account, and when;
- *Access Control:* In a database compliance setting [12], we need to record which user accessed what data item and when.
- *Computational Biology:* The relation between protein structure, dynamics and function can be naturally modeled as time-evolving graphs.

In this chapter, we study two fundamental problems on analyzing time-evolving graphs or second-order tensor streams:

How to efficiently summarize graphs in an effective and intuitive manner? Real graphs are often very large but *sparse*. A typical graph often consists of million of nodes and edges. But the ratio between the number of actual edges and that of all possible edges is still very small. More formally, given a graph $G=(V,E)$, the number of nodes $|V|$ is big, so is the number of edges $|E|$. But the number of edges is much smaller than the number of all possible edges, i.e., $|E|=O(|V|)\ll O(|V|^2)$. A good summarization scheme should also respect the sparsity in the data.

Real graphs are often associated with external domain knowledge. Typically the knowledge relates to actual nodes or edges in the graph. For example, in a protein-protein interaction graph, every node is a specific protein, which biologists know a lot about; in a social network, every node is a person who is associated with external attributes such as age, gender, location. A good summary of a graph should be presented naturally with respect to external knowledge.

Finally, the method has to be efficient in terms of computation and storage in order to process a sequence of large graphs over time.

Inspired by all three reasons, Section 3.2 introduces Compact Matrix Decomposition (CMD), to compute sparse low-rank approximations. CMD dramatically reduces both the computation cost and the space requirements over existing decomposition methods (SVD, CUR). Using CMD as the key building block, we further propose procedures to efficiently construct and analyze dynamic graphs from real-time application data. We provide theoretical guarantee for our methods, and present results on two real, large datasets, one on network flow data (100GB trace of 22K hosts over one month) and one on DBLP (200MB over 25 years).

We show that CMD is often an order of magnitude more efficient than the state of the art (SVD and CUR): it is over *10X faster*, but requires less than *1/10 of the space*, for the same reconstruction accuracy. Finally, we demonstrate how CMD can be used for detecting anomalies and monitoring time-evolving graphs, in which it successfully detects worm-like hierarchical scanning patterns in real network data.

How to monitor the changes of communities over time? Huge amounts of data such as those in the above examples are continuously collected and patterns are also changing over time. Therefore, batch methods for pattern discovery are not sufficient. We need tools that can incrementally find the communities and monitor the changes.

For example, we want to answer questions such as: How do the network hosts interact with each other? What kind of host groups are there, e.g., inactive/active hosts; servers; scanners? Who emails whom? Do the email communities in a organization such as ENRON remain stable, or do they change between workdays (e.g., business-related) and weekends (e.g., friend and relatives), or during major events (e.g.,the FBI investigation and CEO resignation)?

Section 3.3 presents *GraphScope*, that addresses both problems, using information theoretic principles. It needs *no* user-defined parameters. Moreover, it is designed to operate on large graphs, in a streaming fashion. We demonstrate the efficiency and effectiveness of *GraphScope* on real datasets from several diverse domains. In all cases it produces meaningful time-evolving patterns that agree with human intuition.

We first present some background and related work in section 3.1. We then present CMD in section 3.2 and GraphScope in section 3.3, respectively.

3.1 Graph related work

In this section, We survey a few related techniques and [27] provides a more comprehensive survey on graph mining.

3.1.1 Low rank approximation

SVD has served as a building block for many important applications, such as PCA [80] and LSI [105, 44], and has been used as a compression technique [88]. It has also been applied as a correlation detection routine for streaming settings [67, 108]. However, these approaches lead to dense matrices as the result.

For sparse matrices, the diagonalization and SVD are computed by the iterative methods such as Lanczos algorithm [64]. However, the results are still dense. Zhang et al [138] sparsifies the results by zeroing out entries in the singular vectors with small norm. Similarly, Achlioptas and McSherry [5] proposed to randomly zero out the entries in the original matrix.

Recently, Drineas et al. proposed Monte-Carlo approximation algorithms for the standard matrix operations such multiplication [52] and SVD [53], which are two building blocks in their CUR decomposition [54]. CUR has been applied in recommendation system [55], where based on small number of samples about users and products, it can reconstruct the entire user-product relationship.

The essence of CUR decomposition is to construct the subspace using actual columns and rows from the original matrix, which has been explored several times in the literature: Stewart et al. [19] proposed to use top largest columns and rows as the basis for the subspace, but it does not provide a theoretical guarantee. Goreinov et al. [66] proved that sets of columns and rows that span the maximum volume, form good subspaces with relative spectral norm guarantee. However, there are no practical algorithmic construction for that.

3.1.2 Parameter-free mining

Recently, "parameter-free" as a desirable property has received more and more attention in many places. Keogh et al. [83] developed a simple and effective scheme for mining time-series data through compression. Actually, compression or Minimum Description Language (MDL) have become the workhorse of many parameter-free algorithms: biclustering [29], time-evolving graph clustering [118], and spatial-clustering [107].

3.1.3 Biclustering

Biclustering/co-clustering [100] simultaneously clusters both rows and columns into coherent submatrices (biclusters). Cheng and Church [33] proposed a biclustering algorithm for gene expression data analysis, using a greedy algorithm that identifies one bicluster at a time by minimizing the sum squared residue. Cho et al. [37] use a similar measure of coherence but find all the biclusters simultaneously by alternating K-means. Information-theoretic Co-clustering [47] uses an alternating minimization algorithm for KL-divergence between the biclusters and the original matrix. The Cross-association method [29] formulates the biclustering problem as a binary matrix compression problem. More recently,

several streaming extensions of biclustering has been proposed in [6, 118].

There are two main distinctions between our proposed method and all existing work: 1) All the existing methods, except for cross-associations, require a number of parameters to be set, such as the number of biclusters and the minimum support. 2) All the existing methods are context-free approaches, which find biclusters global to the entire dataset, while our approach is context-specific and finds communities in multiple scales. The most related one is by Liu et al. [95]. on leveraging the existing ontology for biclustering, which assumes the hierarchy is given, while our method automatically learns hierarchy from the data.

3.1.4 Time-evolving Graph mining

Graph mining has been a very active area in data mining community. Because of its importance and expressiveness, various problems are studied under graph mining. Recently, the dynamic behavior of networks started to attract attentions due to important applications such as social networks, Web blogs, online recommendation systems.

From the modeling viewpoint, Faloutsos et al. [60] have shown the power-law distribution on the Internet graph. Kumar et al. [90] studied the model for web graphs. Leskovec et al. [91] discovered the shrinking diameter phenomena on time-evolving graphs. Wang et al. [130] proposed an epidemic threshold model for modeling virus propagation in the network.

From the algorithm viewpoint, Chakrabarti et al. [28] proposed evolutionary settings for k-means and agglomerative hierarchical clustering. Asur et al. [15] proposed an event-based characterization of clustering changes over time. Chi et al. [35, 36] proposed two graph clustering algorithms for blogospheres.

3.2 Compact Matrix Decomposition

How to efficiently summarize graphs in an effective and intuitive manner? Graphs are used in multiple important applications such as network traffic monitoring, web structure analysis, social network mining, protein interaction study, and scientific computing. Given a large graph, we want to discover patterns and anomalies in spite of the high dimensionality of data. We refer to this challenge as the *static graph mining* problem.

An even more challenging problem is finding patterns in graphs that evolve over time. In this setting, we want to find patterns, summaries, and anomalies for the given time

window, as well as across multiple time windows. Specifically for these applications that generate huge volume of data with high speed, the method has to be fast, so that it can catch anomalies early on. Closely related questions are how to summarize dynamic graphs, so that they can be efficiently stored, e.g., for historical analysis. We refer to this challenge as the *dynamic graph mining* problem.

The typical way of summarizing and approximating matrices (the representation of graphs) is through transformations, with SVD/PCA [64, 80] and random projections [77] being popular choices. Although all these methods are very successful in general, for large sparse graphs they may require huge amounts of space, exactly because their resulting matrices are not sparse any more.

Large, real graphs are often very sparse. For example, the web graph [90], Internet topology graphs [60], who-trusts-whom social networks, along with numerous other real graphs, are all sparse. Drineas et al. [54] proposed the CUR decomposition method, which partially addresses the loss-of-sparsity issue.

We propose a new method, called *Compact Matrix Decomposition (CMD)*, for generating low-rank matrix approximations. CMD provides provably equivalent decomposition as CUR, but it requires much *less* space and computation time, and hence is *more* efficient.

Moreover, we show that CMD can not only analyze static graphs, but it can also be extended to handle dynamic graphs. Another contribution of our work is exactly a detailed procedure to put CMD into practice, and especially for high-speed applications like Internet traffic monitoring, where new traffic matrices are streamed-in in real time. Overall, our method has the following desirable properties:

- **Fast:** Despite the high dimensionality of large graphs, the entire mining process is fast, which is especially important for high-volume, streaming applications.

- **Space efficient:** We preserve the sparsity of graphs so that both the intermediate results and the final results fit in memory, even for large graphs that are usually too expensive to mine today.

- **Anomaly detection:** We show how to spot anomalies, that is, rows, columns or time-ticks that suffer from high reconstruction error. A vital step here is our proposed fast method to estimate the reconstruction error of our approximations.

3.2.1 Problem definition

There are many approaches to extract patterns or structures from a graph given its adjacency matrix. In particular, we consider the patterns as a low dimensional summary of the adjacency matrix. Hence, the goal is to efficiently identify a low dimensional summary while preserving the sparsity of the graph.

More specifically, we formulate the problem as a matrix decomposition problem. The basic question is how to approximate $\mathbf{A}$ as the product of three smaller matrices $\mathbf{C} \in \mathbb{R}^{m \times c}$, $\mathbf{U} \in \mathbb{R}^{c \times r}$, and $\mathbf{R} \in \mathbb{R}^{r \times n}$, such that: (1) $|\mathbf{A} - \mathbf{CUR}|$[1] is small, and (2) $\mathbf{C}$,$\mathbf{U}$, and $\mathbf{R}$ can be computed quickly using a small space. More intuitively, we look for a low rank approximation of $\mathbf{A}$ that is both accurate and can be efficiently computed.

With matrix decomposition as our core component, we consider two general classes of graph mining problems, depending on the input data:

Static graph mining: Given a sparse matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, find patterns, outliers, and summarize it. In this case, the input data is a given static graph represented as its adjacency matrix.

Dynamic graph mining: Given timestamped pairs (e.g., source-destination pairs from network traffic, email messages, IM chats), potentially in high volume and high speed, construct graphs, find patterns, outliers, and summaries as they evolve. In other words, the input data are raw event records that need to be pre-processed.

The research questions now are how to sample data and construct matrices (graphs) efficiently? How to leverage the matrix decomposition of the static case, into the mining process? What are the underlying processing modules, and how do they interact with each other? These are all practical questions that require a systematic process. Next we first introduce the computational kernel CMD in Section 3.2.2; then we discuss the mining process based on CMD in Section 3.2.3.

3.2.2 Compact matrix Decomposition

In this section, we present the Compact Matrix Decomposition (CMD), to decompose large sparse matrices. Such method approximates the input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ as a product of three small matrices constructed from sampled columns and rows, while preserving the sparsity of the original $\mathbf{A}$ after decomposition. More formally, it approximates the matrix

[1]The particular norm does not matter. For simplicity, we use squared Frobenius norm, i.e., $|\mathbf{A}| = \sum_{i,j} \mathbf{A}(i,j)^2$.

Algorithm 3.1: INITIAL SUBSPACE CONSTRUCTION($\mathbf{A} \in \mathbb{R}^{n \times m}$, sample size c)

Output: $\mathbf{C}_d \in \mathbb{R}^{m \times c}$

for $x = 1$ *to* n **do**

  $P(x) = \sum_i \mathbf{A}(i,x)^2 / \sum_{i,j} \mathbf{A}(i,j)^2$ `// column distribution`

for $i = 1$ *to* c **do**

  Pick $j \in 1:n$ based on distribution $P(x)$ `// sample columns`

  Compute $\mathbf{C}_d(:,i) = \mathbf{A}(:,j)/\sqrt{cP(j)}$ `// rescale columns`

$\mathbf{A}$ as $\tilde{\mathbf{A}} = \mathbf{C_s U R_s}$, where $\mathbf{C_s} \in \mathbb{R}^{m \times c'}$ ($\mathbf{R_s} \in \mathbb{R}^{r' \times n}$) contains $c(r)$ scaled columns(rows) sampled from $\mathbf{A}$, and $\mathbf{U} \in \mathbb{R}^{c' \times r'}$ is a small dense matrix which can be computed from $\mathbf{C_s}$ and $\mathbf{R_s}$. CMD leverages the prior work of CUR [54], but requires significantly less memory and computation to achieve the same approximation accuracy in decomposition. It enables the analysis of large graphs that would not have been practical to analyze today due to the high memory and computation costs of existing methods.

We first describe how to construct the subspace for a given input matrix. We then discuss how to compute its low rank approximation.

Subspace Construction

Since the subspace is spanned by the columns of the matrix, we choose to use sampled columns to represent the subspace.

Biased sampling: The key idea for picking the columns is to sample columns with replacement, biased towards those ones with higher norms. In other words, the columns with higher entry values will have higher chance to be selected multiple times. Such sampling procedure, used by CUR, is proved to yield an optimal approximation [54]. Algorithm 3.1 lists the detailed steps to construct a low dimensional subspace for further approximation. Note that, the biased sampling will bring a lot of duplicated samples. Next we discuss how to remove them without affecting the accuracy.

Duplicate column removal: CMD carefully removes duplicate columns and rows after sampling, and thus it reduces both the storage space required as well as the computational effort. Intuitively, the directions of those duplicate columns are more important than the other columns. Thus a key step of subspace construction is to scale up the columns that are sampled multiple times while removing the duplicates. Pictorially, we take matrix $\mathbf{C}_d$, which is the result of Algorithm 3.1 (see Figure 3.1(a)) and turn it into the much narrower matrix $\mathbf{C}_s$ as shown in Figure 3.1(b), with proper scaling. The method for selecting $\mathbf{R}_d$

Algorithm 3.2: CMD SUBSPACE CONSTRUCTION($\mathbf{A} \in \mathbb{R}^{n \times m}$, sample size c)

Output: $\mathbf{C}_s \in \mathbb{R}^{m \times c'}$

Compute $\mathbf{C}_d$ using the initial subspace construction

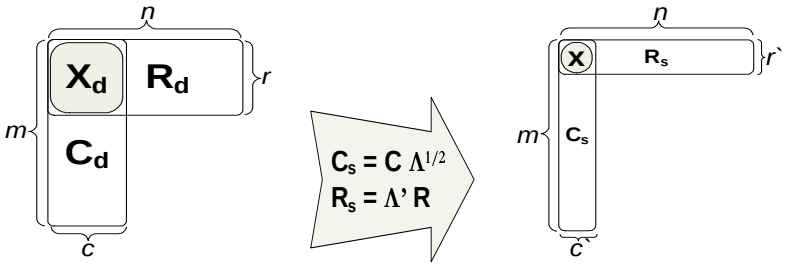
Let $\mathbf{C} \in \mathbb{R}^{m \times c'}$ be the unique columns of $\mathbf{C}_d$

for $i = 1$ *to* c' **do**

    Let u be the number of $\mathbf{C}(:, i)$ in $\mathbf{C}_d$

    Compute $\mathbf{C}_s(:, i) \leftarrow \sqrt{u} \cdot \mathbf{C}(:, i)$

and constructing $\mathbf{R}_s$ will be described shortly.



(a)with duplicates (b) without duplicates

Figure 3.1: Illustration of CUR and CMD

Algorithm 3.2 shows the algorithm to construct a low dimensional subspace represented with a set of *unique* columns. Each column is selected by sampling the input matrix $\mathbf{A}$, and then scaling it up based on square root of the number of times it being selected. The resulting subspace also emphasizes the impact of large columns to the same extent as the result in Algorithm 3.1. Using the notations in Table 3.1, we show by Theorem 3.1 that the top-k subspaces spanned by $\mathbf{C}_d$ with duplicates and $\mathbf{C}_s$ without duplicates are the same.

Theorem 3.1 (Duplicate columns)**.** *Matrices $\mathbf{C}_s$ and $\mathbf{C}_d$, defined in Table 3.1, have the same singular values and left singular vectors.*

Proof. It is easy to see $\mathbf{C}_d = \mathbf{C}\mathbf{D}^T$. Then we have

$$\mathbf{C}_d\mathbf{C}_d^T = \mathbf{C}\mathbf{D}^T(\mathbf{C}\mathbf{D}^T)^T = \mathbf{C}\mathbf{D}^T\mathbf{D}\mathbf{C}^T \tag{3.1}$$

$$= \mathbf{C}\Lambda\mathbf{C}^T = \mathbf{C}\Lambda^{1/2}\Lambda^{1/2}\mathbf{C}^T \tag{3.2}$$

$$= \mathbf{C}\Lambda^{1/2}(\mathbf{C}\Lambda^{1/2})^T = \mathbf{C}_s\mathbf{C}_s^T \tag{3.3}$$

where $D \in \mathbb{R}^{c' \times c}$ and $\Lambda \in \mathbb{R}^{k \times k}$ are defined in Table 3.1.

Definition	Size
$\mathbf{C} = [\mathbf{C_1}, \ldots, \mathbf{C_{c'}}]$	$m \times c'$
$\mathbf{C}_d = [\underbrace{\mathbf{C_1}, \ldots, \mathbf{C_1}}_{d_1}, \ldots, \underbrace{\mathbf{C_{c'}}, \ldots, \mathbf{C_{c'}}}_{d_{c'}}]$	$m \times c$, $c = \sum_i d_i$
$\mathbf{D} = [\underbrace{\mathbf{e_1}, \ldots, \mathbf{e_1}}_{d_1}, \ldots, \underbrace{\mathbf{e_{c'}}, \ldots, \mathbf{e_{c'}}}_{d_{c'}}]$	$c' \times c$, $c = \sum_i d_i$
$\mathbf{\Lambda} = \text{diag}(d_1, \ldots, d_{c'})$	$c' \times c'$
$\mathbf{C}_S = [\sqrt{d_1}\mathbf{C_1}, \ldots, \sqrt{d_{c'}}\mathbf{C_{c'}}] = \mathbf{C}\mathbf{\Lambda}^{1/2}$	$m \times c'$
$\mathbf{R} = [\mathbf{R_1}, \ldots, \mathbf{R_{r'}}]^T$	$r' \times m$
$\mathbf{R}_d = [\underbrace{\mathbf{R_1}, \ldots, \mathbf{R_1}}_{d'_1}, \ldots, \underbrace{\mathbf{R_{r'}}, \ldots, \mathbf{R_{r'}}}_{d'_{r'}}]$	$r \times n$, $r = \sum_i d'_i$
$\mathbf{D}' = [\underbrace{\mathbf{e_1}, \ldots, \mathbf{e_1}}_{d'_1}, \ldots, \underbrace{\mathbf{e_{r'}}, \ldots, \mathbf{e_{r'}}}_{d'_{r'}}]$	$r' \times r$, $r = \sum_i d'_i$
$\mathbf{\Lambda}' = \text{diag}(d'_1, \ldots, d'_{r'})$	$r' \times r'$
$\mathbf{R}_S = [d'_1\mathbf{R_1}, \ldots, d'_{r'}\mathbf{R_{r'}}] = \mathbf{\Lambda}'\mathbf{R}$	$r' \times n$

Table 3.1: Matrix Definition: $\mathbf{e}_i$ is a column vector with all zeros except a one as its i-th element

Now we can diagonalize either the product $\mathbf{C}_d\mathbf{C}_d^T$ or $\mathbf{C}_s\mathbf{C}_s^T$ to find the same singular values and left singular vectors for both $\mathbf{C}_d$ and $\mathbf{C}_s$. □

Low Rank Approximation

The goal is to form an approximation of the original matrix $\mathbf{X}$ using the sampled column $\mathbf{C}_s$. For clarity, we use $\mathbf{C}$ for $\mathbf{C}_s$. More specifically, we want to project $\mathbf{X}$ onto the space spanned by $\mathbf{C}_s$, which can be done as follows:

- project $\mathbf{X}$ onto the span of $\mathbf{C_s}$;
- reduce the cost by further duplicate row removal.

Column projection: We first construct the orthonormal basis of $\mathbf{C}$ using SVD (say $\mathbf{C} = \mathbf{U}_C\mathbf{\Sigma}_C\mathbf{V}_C^T$), and then projecting the original matrix onto this identified orthonormal basis $\mathbf{U}_C \in \mathbb{R}^{m\times c}$. Since $\mathbf{U}_C$ is usually large and dense, we do not compute the projection of matrix $\mathbf{A}$ directly as $\mathbf{U}_C\mathbf{U}_C^T\mathbf{A} \in \mathbb{R}^{m\times m}$. Instead, we compute a low rank approximation of $\mathbf{A}$ based on the observation that $\mathbf{U}_c = \mathbf{C}\mathbf{V}_C\mathbf{\Sigma}_C^{-1}$, where $\mathbf{C} \in \mathbb{R}^{m\times c}$ is large but sparse, $\mathbf{V}_C \in \mathbb{R}^{c\times k}$ is dense but small, and $\mathbf{\Sigma} \in \mathbb{R}^{k\times k}$ is a small diagonal matrix [2]. Therefore, we

[2]In our experiment, both $\mathbf{V}_C$ and $\mathbf{\Sigma}_C$ have significantly smaller number of entries than $\mathbf{A}$.

Algorithm 3.3: APPRMULTIPLICATE(matrix $\mathbf{A} \in \mathbb{R}^{c \times m}$, $\mathbf{B} \in \mathbb{R}^{m \times n}$, sample size r

Output: $\mathbf{C}_s \in \mathbb{R}^{c \times r'}$ and $\mathbf{R}_s \in \mathbb{R}^{r' \times n}$

1 **for** $x = 1$ *to* m **do**

2 $\quad Q(x) = \sum_i \mathbf{B}(x,i)^2 / \sum_{i,j} \mathbf{B}(i,j)^2$ `// row distribution of` $\mathbf{B}$

3 **for** $i = 1$ *to* r **do**

4 $\quad$ Pick $j \in 1:r$ based on distribution $Q(x)$

5 $\quad$ Set $\mathbf{R}_d(i,:) = \mathbf{B}(j,:)/\sqrt{rQ(j)}$

6 $\quad$ Set $\mathbf{C}_d(:,i) = \mathbf{A}(:,j)/\sqrt{rQ(j)}$

7 $\mathbf{R} \in \mathbb{R}^{r' \times n}$ are the unique rows of $\mathbf{R}_d$

8 $\mathbf{C} \in \mathbb{R}^{c \times r'}$ are the unique columns of $\mathbf{C}_d$

9 **for** $i = 1$ *to* r' **do**

10 $\quad u$ is the number of $\mathbf{R}(i,:)$ in $\mathbf{R}_d$

11 $\quad$ Set $\mathbf{R}_s(i,:) \leftarrow u \cdot \mathbf{R}(i,:)$

12 $\quad$ Set $\mathbf{C}_s(:,i) \leftarrow \mathbf{C}(:,i)$

have the following:

$$\begin{aligned}\tilde{\mathbf{A}} &= \mathbf{U}_c\mathbf{U}_c^T\mathbf{A} = \mathbf{C}\mathbf{V}_C\mathbf{\Sigma}_C^{-1}(\mathbf{C}\mathbf{V}_C\mathbf{\Sigma}_C^{-1})^T\mathbf{A} \\ &= \mathbf{C}(\mathbf{V}_C\mathbf{\Sigma}_C^{-2}\mathbf{V}_C^T\mathbf{C}^T)\mathbf{A} = \mathbf{C}\mathbf{T}\mathbf{A}\end{aligned}$$

where $\mathbf{T} = (\mathbf{V}_C\mathbf{\Sigma}_C^{-2}\mathbf{V}_C^T\mathbf{C}^T) \in \mathbb{R}^{c \times m}$. Although $\mathbf{C} \in \mathbb{R}^{m \times c}$ is sparse, $\mathbf{T}$ is still dense and big. we further optimize the low-rank approximation by reducing the multiplication overhead of two large matrices $\mathbf{T}$ and $\mathbf{A}$. Specifically, given two matrices $\mathbf{A}$ and $\mathbf{B}$ (assume $\mathbf{AB}$ is defined), we can sample both columns of $\mathbf{A}$ and rows of $\mathbf{B}$ using the biased sampling algorithm (i.e., biased towards the ones with bigger norms)[3]. The selected rows and columns are then scaled accordingly for multiplication. This sampling algorithm brings the same problem as column sampling, i.e., there exist duplicate rows.

Duplicate row removal: CMD removes duplicate rows in multiplication based on Theorem 3.2. In our context, CMD samples and scales r' unique rows from $\mathbf{A}$ and extracts the corresponding r' columns from $\mathbf{C}^T$ (last term of $\mathbf{T}$). Algorithm 12 shows the details. Line 1-2 computes the distribution; line 3-6 performs the biased sampling and scaling; line 7-10 removes duplicates and rescales properly.

Theorem 3.2 proves the correctness of the matrix multiplication results after removing the duplicated rows. Note it is important that we use different scaling factors for removing

[3] [52] presents the details about the Monte-Carlo matrix multiplication algorithm.

Algorithm 3.4: CMD(matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, sample size c and r)

Output: $\mathbf{C} \in \mathbb{R}^{m \times c}$, $\mathbf{U} \in \mathbb{R}^{c \times r}$ and $\mathbf{R} \in \mathbb{R}^{r \times n}$

1 find $\mathbf{C}$ from CMD subspace construction

2 diagonalize $\mathbf{C}^T\mathbf{C}$ to find $\mathbf{\Sigma}_C$ and $\mathbf{V}_C$

3 find $\mathbf{C}_s$ and $\mathbf{R}_s$ using ApprMultiplication on $\mathbf{C}^T$ and $\mathbf{A}$

4 $\mathbf{U} = \mathbf{V}_C\mathbf{\Sigma}_C^{-2}\mathbf{V}_C^T\mathbf{C}_R$

duplicate columns (square root of the number of duplicates) and rows (the exact number of duplicates). Inaccurate scaling factors will incur a huge approximation error.

Theorem 3.2 (duplicate rows)**.** *Let I, J be the set of selected rows (with and without duplicates, respectively):* $J = [\underbrace{1, \ldots, 1}_{d'_1}, \ldots, \underbrace{r', \ldots, r'}_{d'_{r'}}]$ *and* $I = [1, \ldots, r']$*. Then given* $\mathbf{A} \in \mathbb{R}^{m_a \times n_a}$, $\mathbf{B} \in \mathbb{R}^{m_b \times n_b}$ *and* $\forall i \in I, i \leq$ *min*(n_a, m_b)*, we have*

$$\mathbf{A}(:, J)\mathbf{B}(J, :) = \mathbf{A}(:, I)\mathbf{\Lambda}'\mathbf{B}(I, :)$$

where $\mathbf{\Lambda}' =$ *diag*$(d'_1, \ldots, d'_{r'})$.

Proof. Denote $\mathbf{X} = \mathbf{A}(:, J)\mathbf{B}(J, :)$ and $\mathbf{Y} = \mathbf{A}(:, I)\mathbf{\Lambda}'\mathbf{B}(I, :)$. Then, we have

$$\begin{aligned}\mathbf{X}(i, j) &= \sum_{k \in J} \mathbf{A}(i, k)\mathbf{B}(k, j) \\ &= \sum_{k \in I} d_{i_k}\mathbf{A}(i, k)\mathbf{B}(k, j) = \mathbf{Y}(i, j)\end{aligned}$$

□

To summarize, Algorithm 13 lists the steps involved in CMD to perform matrix decomposition for finding low rank approximations.

Implementation caveat: In practice, $\mathbf{C}$ and $\mathbf{R}$ might not agree on the same subspace as the top k column and row subspaces induced by SVD. As a result when the subspaces are not aligned well, it can lead to a huge numeric error even if both column and row subspaces give good approximation, respectively. More specifically, $\|\mathbf{A} - \mathbf{CUR}\|$ can be big despite the fact that both $\|\mathbf{A} - \mathbf{C}\mathbf{C}^\dagger\mathbf{A}\|$ (from Algorithm 3.1) and $\|\mathbf{C}^T\mathbf{A} - \mathbf{C}_R\mathbf{R}_s\|$(from Algorithm 12) are small. Two implementation tricks are practically very effective in handling this issue: 1) regularize the number of singular values/vectors of $\mathbf{C}$ to be small; 2) re-sampling the $\mathbf{C}$ and $\mathbf{R}$ when the final error appears to be huge.

3.2.3 CMD in practice

In this section, we present several practical techniques for mining dynamic graphs using CMD, where applications continuously generate data for graph construction and analysis.

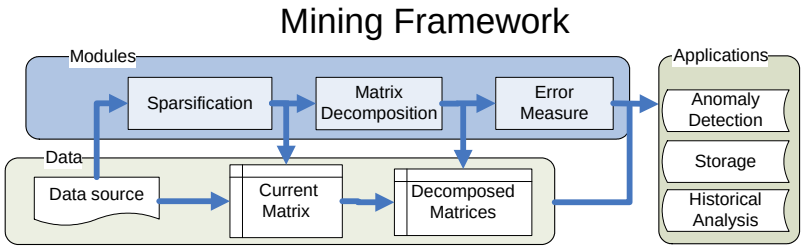


Figure 3.2: A flowchart for mining large graphs with low rank approximations

Figure 3.2 shows the flowchart of the whole mining process. The process takes as input data from application, and generates as output mining results represented as low-rank data summaries and approximation errors. The results can be fed into different mining applications such as anomaly detection and historical analysis.

The *data source* is assumed to generate a large volume of real time event records for constructing large graphs (e.g., network traffic monitoring and analysis). Because it is often hard to buffer and process all data that are streamed in, we propose one more step, namely, *sparsification*, to reduce the incoming data volume by sampling and scaling data to approximate the original full data (Section 3.2.3).

Given the input data summarized as a *current matrix* $\mathbf{A}$, the next step is *matrix decomposition* (Section 3.2.3), which is the core component of the entire flow to compute a lower-rank matrix approximation. Finally, the *error measure* quantifies the quality of the mining result as an additional output.

Sparsification

Here we present an algorithm to sparsify input data, focusing on applications that continuously generate data to construct sequences of graphs dynamically. For example, consider a network traffic monitoring system where network flow records are generated in real time. These records are of the form (source, destination, timestamp, #flows). Such traffic data can be used to construct communication graphs periodically (e.g., one graph per hour). For each time window (e.g., 1pm-2pm), we can incrementally build an adjacency matrix $\mathbf{A}$ by

Algorithm 3.5: SPARSIFICATION(update index $(s_1, d_1), \ldots, (s_n, d_n)$)

Output: Adjacency matrix $\mathbf{A}$

initialize $\mathbf{A} = 0$

for $t = 1$ *to* n **do**

    // decide whether to sample

    if *Bernoulli(p)= 1* **then**

        $\mathbf{A}(s_t, d_t) = \mathbf{A}(s_t, d_t) + \Delta v$

    $\mathbf{A} = \mathbf{A}/p$ // scale up $\mathbf{A}$ by $1/p$

updating its entries as data records are coming in. Each new record triggers an update on an entry (i, j) with a value increase of Δv, i.e., $\mathbf{A}(i, j) = \mathbf{A}(i, j) + \Delta v$.

The data usually arrive as updates to edges of the graph. More formally, an update u is a vertex pair (i, j) where i is the source, j the destination. From the matrix prospective, the update increments the (i, j)-entry of $\mathbf{A}$ by one, $\mathbf{A}(i, j) = \mathbf{A}(i, j) + 1$. The adjacency matrix (the graph) is formed based on all updates. In time evolving scenario, such matrices are built periodically based on the recent updates. For instance, in the experiment we construct a new network flow matrix every hour.

In practice, the updates are often coming into the system very fast. It is very expensive even just to keep track of all the updates. For example, the network router has thousands of packet flows per second, each packet flow can be considered as an update to a corresponding entry in the adjacency matrix $\mathbf{A}$. In general, it is very hard to monitor all the flows. The *Sparsification* process exactly aims at this problem by reducing the incoming update traffic.

The key idea to sparsify input data during the above process is to sample updates with a certain probability p, and then scale the sampled matrix by a factor $1/p$ to approximate the true matrix. Algorithm 14 lists this sparsification algorithm.

We can further simplify the above process by avoiding doing a Bernoulli draw for every update. Note that the probability of skipping k consecutive updates is $(1 - p)^k p$ (as in the reservoir sampling algorithm [128]). Thus instead of deciding whether to select the current update, we decide how many updates to skip before selecting the next update. After sampling, it is important that we scale up all the entries of $\mathbf{A}$ by $1/p$ in order to approximate the true adjacency matrix (based on all updates).

The approximation error of this sparsification process can be bounded and estimated as a function of matrix dimensions and the sampling probability p. Specifically, suppose $\mathbf{A}^*$ is the true matrix that is constructed using all updates. For a random matrix $\mathbf{A}$ that

approximates $\mathbf{A}^*$ for every of its entries, we can bound the approximation error with a high probability using the following theorem (see [5] for proof):

Theorem 3.3 (Random matrix)**.** *Given a matrix $\mathbf{A}^* \in \mathbb{R}^{m \times n}$, let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a random matrix such that for all i,j: $\mathbb{E}(\mathbf{A}(i,j)) = \mathbf{A}^*(i,j)$ and Var$(\mathbf{A}(i,j)) \leq \sigma^2$ and*

$$|\mathbf{A}(i,j) - \mathbf{A}^*(i,j)| \leq \frac{\sigma\sqrt{m+n}}{log^3(m+n)}$$

For any $m+n \geq 20$, with probability at least $1 - 1/(m+n)$,

$$\|\mathbf{A} - \mathbf{A}^*\|_2 < 7\sigma\sqrt{m+n}$$

With our data sparsification algorithm, it is easy to observe that $\mathbf{A}(i,j)$ follows a binomial distribution with expectation $\mathbf{A}^*(i,j)$ and variance $\mathbf{A}^*(i,j)(1-p)$. We can thus apply Theorem 3.3 to estimate the error bound with a maximum variance $\sigma = (1-p)\text{max}_{i,j}(\mathbf{A}^*(i,j))$. Each application can choose a desirable sampling probability p based on the estimated error bounds, to trade off between processing overhead and approximation error.

Matrix Decomposition

Once we construct the adjacency matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the next step is to compactly summarize it. This is the key component of our process, where various matrix decomposition methods can be applied to the input matrix $\mathbf{A}$ for generating a low-rank approximation. As we mentioned, we consider SVD, CUR and CMD as potential candidates: SVD because it is the traditional, optimal method for low-rank approximation; CUR because it preserves the sparsity property; and CMD because, as we show, it achieves significant performances gains over both previous methods.

Error Measure

The last step of our framework involves measuring the quality of the low rank approximations. An approximation error is useful for certain applications, such as anomaly detection, where a sudden large error may suggest structural changes in the data. A common metric to quantify the error is the sum-square-error (SSE), defined as SSE$= \sum_{i,j}(\mathbf{A}(i,j) - \tilde{\mathbf{A}}(i,j))^2$. In many cases, a relative SSE (SSE/ $\sum_{i,j}(\mathbf{A}(i,j)^2$), computed

Algorithm 3.6: ESTIMATESSE($\mathbf{A} \in \mathbb{R}^{n\times m}$,$\mathbf{C} \in \mathbb{R}^{m\times c}$,$\mathbf{U} \in \mathbb{R}^{c\times r}$,$\mathbf{R} \in \mathbb{R}^{r\times n}$)

Output: Approximation error $S\tilde{S}E$

1 rset = sr random numbers from 1:m

2 cset = sr random numbers from 1:n

3 $\tilde{\mathbf{A}}_S = \mathbf{C}(\text{rset}, :) \cdot \mathbf{U} \cdot \mathbf{R}(:, \text{cset})$

4 $\mathbf{A}_S = \mathbf{A}(\text{rset}, \text{cset})$

5 $S\tilde{S}E$ = $\frac{m\cdot n}{sr\cdot sc}$SSE($\mathbf{A}_S$, $\tilde{\mathbf{A}}_S$)

as a fraction of the original matrix norm, is more informative because it does not depend on the dataset size.

How to compute SSE usually depends on the matrix decomposition method. For example, in SVD, it is easy to compute SSE which is the difference of $\|\mathbf{A}\|$ and $\|\mathbf{\Sigma}\|$. For CUR and CMD, it is more expensive because computing SSE requires one to evaluate the product of $\mathbf{CUR}$. Further, $\mathbf{U}$ is a dense matrix, the product of $\mathbf{CUR}$ is likely to be dense too, incurring more space overhead. Direct computation of SSE requires us to calculate the norm of two big matrices, namely, $\mathbf{X}$ and $\mathbf{X} - \tilde{\mathbf{X}}$ which is expensive. We propose an approximation algorithm to estimate SSE (Algorithm 15) more efficiently. The intuition is to compute the sum of squared errors using only a subset of the entries. The results are then scaled to obtain the estimated $S\tilde{S}E$.

With our approximation, the true SSE and the estimated $S\tilde{S}E$ converge to the same value on expectation based on the following lemma [4]. In our experiments, this algorithm can achieve small approximation errors with only a small sample size.

Lemma 3.1. *Given the matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ and its estimate $\tilde{\mathbf{A}} \in \mathbb{R}^{m\times n}$ such that $\mathbb{E}(\tilde{\mathbf{A}}(i,j)) = \mathbf{A}(i,j)$ and Var$(\tilde{\mathbf{A}}(i,j)) = \sigma^2$ and a set S of sample entries, then*

$$\mathbb{E}(SSE) = \mathbb{E}(S\tilde{S}E) = mn\sigma^2$$

where SSE $= \sum_{i,j}(\mathbf{A}(i,j) - \tilde{\mathbf{A}}(i,j))^2$ and S$\tilde{S}$E $= \frac{mn}{|S|}\sum_{(i,j)\in S}(\mathbf{A}(i,j) - \tilde{\mathbf{A}}(i,j))^2$

Proof.

$$\begin{aligned}\mathbb{E}(\text{SSE}) &= \sum_{i,j}\mathbb{E}(\mathbf{A}(i,j) - \tilde{\mathbf{A}}(i,j))^2 \\ &= mn\mathbb{E}(\mathbf{A}(i,j) - \tilde{\mathbf{A}}(i,j))^2 \\ &= mn\sigma^2\end{aligned}$$

[4]The variance of SSE and $S\tilde{S}E$ can also be estimated but requires higher moment of $\tilde{\mathbf{A}}$.

$$\begin{aligned}\mathbb{E}(\tilde{\text{SSE}}) &= \frac{mn}{|S|}\sum_{(i,j)\in S}\mathbb{E}(\mathbf{A}(i,j)-\tilde{\mathbf{A}}(i,j))^2 \\ &= mn\mathbb{E}(\mathbf{A}(i,j)-\tilde{\mathbf{A}}(i,j))^2 \\ &= mn\sigma^2\end{aligned}$$

□

3.2.4 Experiments

In this section, we evaluate both CMD and our mining framework, using two large datasets with different characteristics. The candidates for comparison include SVD and CUR. The evaluation focuses on 1) space requirement, 2) CPU time, 3) Accuracy estimation cost as well as 4) sparsification effect.

Overall, CMD performs much better than both SVD and CUR as shown in Algorithm 3.3[5].

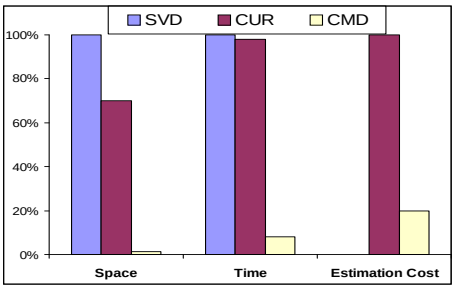


Figure 3.3: Compared to SVD and CUR, CMD achieves lower space and time requirement as well as fast estimation latency. Note that every thing is normalized by the largest cost in that category when achieving 90% accuracy. e.g., The space requirement of CMD is 1.5% of SVD, while that of CUR is 70%.

Experimental Setup

In this section, we first describe the two datasets; then we define the performance metrics used in the experiment.

[5]These experiments are based on network traffic dataset with accuracy 90%. Note that the estimation cost is not applicable to SVD.

data	dimension	$\|E\|$	nonzero entries
Network flow	22K-by-22K	12K	0.0025%
DBLP data	428K-by-3.6K	64K	0.004%
Enron email	34K-by-34K	41K	$.0003\%$
Transactional data	9180-by-784	62K	0.87%

Table 3.2: Dataset summary

The Network Flow Dataset. The traffic trace consists of TCP flow records collected at the backbone router of a class-B university network. Each record in the trace corresponds to a directional TCP flow between two hosts with timestamps indicating when the flow started and finished.

With this traffic trace, we study how the communication patterns between hosts evolve over time, by reading traffic records from the trace, simulating network flows arriving in real time. We use a window size of Δt seconds to construct a source-destination matrix every Δt seconds, where $\Delta t = 3600$ (one hour). For each matrix, the rows and the columns correspond to source and destination IP addresses, respectively, with the value of each entry (i, j) representing the total number of TCP flows (packets) sent from the i-th source to the j-th destination during the corresponding Δt seconds. Because we cannot observe all the flows to or from a non-campus host, we focus on the intranet environment, and consider only campus hosts and intra-campus traffic. The resulting trace has over 0.8 million flows per hour (i.e., sum of all the entries in a matrix) involving 21,837 unique campus hosts. The average percentage of nonzero entries for each matrix is 2.5×10^{-5}.

The distribution of the entry values is very skewed (a power law distribution). Most of hosts have zero traffic, with only a few of exceptions which were involved with high volumes of traffic (over 10^4 flows during that hour). Given such skewed traffic distribution, we rescale all the non-zero entries by taking the natural logarithm (actually, $\log(x + 1)$, to account for $x = 0$), so that the matrix decomposition results will not be dominated by a small number of very large entry values.

Non-linear scaling the values is very important: experiments on the original, bursty data would actually give excellent compression results, but poor anomaly discovery capability: the 2-3 most heavy rows (speakers) and columns (listeners) would dominate the decompositions, and everything else would appear insignificant.

The DBLP Bibliographic Dataset. Based on DBLP data [3], we generate an author-conference graph for every year from year 1980 to 2004 (one graph per year). An edge (a, c) in such a graph indicates that author a has published in conference c during that year. The weight of (a, c) (the entry (a, c) in the matrix $\mathbf{A}$) is the number of papers a published at conference c during that year. In total, there are 428,398 authors and 3,659 conferences.

The average percentage of nonzero entries is 4×10^{-5}. Note that the DBLP matrix is much denser than the network flow one.

The graph for DBLP is less sparse compared with the source-destination traffic matrix. However, we observe that the distribution of the entry values is still skewed, although not as much skewed as the source-destination graph. Intuitively, network traffic is concentrated in a few hosts, but publications in DBLP are more likely to spread out across many different conferences. Most authors, who has one or more paper that year, only publishes in one conference. A few authors publishes in several (maximum is 25). Compared with host-by-host flow matrix, the author-conference matrix is less sparse and skewed.

The Enron Emails Dataset. This consists of the email communications in Enron Inc. from Jan 1999 to July 2002 [2]. We construct sender-to-recipient graphs on a monthly basis. The graphs have m = n = 34, 280 senders/recipients (the number of nodes) with average of 4130 distinct sender-recipient pairs (the number of edges) every month.

The Transactional Dataset. The Transactional dataset has 9180 transactions over 784 accounts of a company for one year. 62,922 nonzero transaction-account pairs exist in the data. The matrix is densest among all four datasets. Furthermore, it does not exhibit a low-rank structure like the rest.

Performance Metric. We use the following three metrics to quantify the mining performance:

- **Approximation accuracy:** This is the key metric that we use to evaluate the quality of the low-rank matrix approximation output. It is defined as:

$$\text{accuracy} = 1 - \text{relative SSE}$$

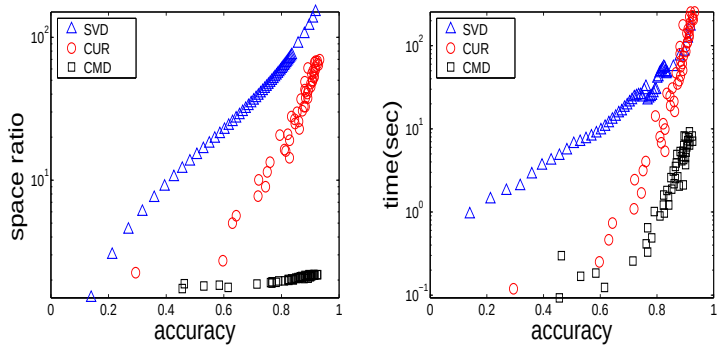
- **Space ratio:** We use this metric to quantify the required space usage. It is defined as the ratio of the number of output matrix entries to the number of input matrix entries. So a larger space ratio means more space consumption.

- **CPU time:** We use the CPU time spent in computing the output matrices as the metric to quantify the computational expense.

All the experiments are performed on the same dedicated server with four 2.4GHz Xeon CPUs and 12GB memory. For each experiment, we repeat it 10 times, and report the mean.

The Performance of CMD

In this section, we compare CMD with SVD and CUR, using static graphs constructed from the four datasets. No sparsification process is required for statically constructed graphs. We vary the target approximation accuracy, and compare the space and CPU time used by the three methods.



(a) space vs accuracy (b) time vs accuracy

Figure 3.4: Network: CMD takes the least amount of space and time to decompose the source-destination matrix; the space and time required by CUR increases fast as the accuracy increases due to the duplicated columns and rows.

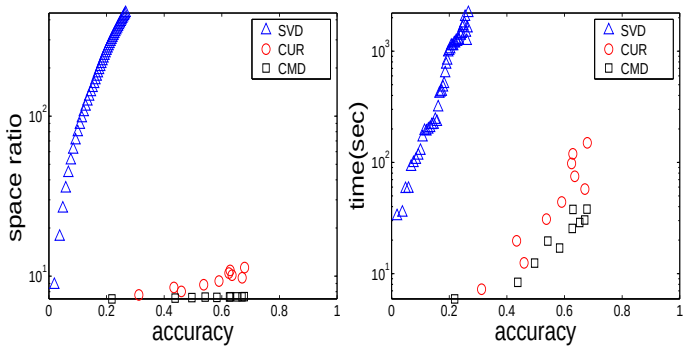
Network Dataset.

- **Network-Space**: We first evaluate the space consumption for three different methods to achieve a given approximation accuracy. Figure 3.4(a) shows the space ratio (to the original matrix) as the function of the approximation accuracy for network flow data. Note the Y-axis is in log scale. SVD uses the most amount of space (over 100X larger than the original matrix). CUR uses smaller amount of space than SVD, but it still has huge overhead (over 50X larger than the original space), especially when high accuracy estimation is needed. Among the three methods, CMD uses the least amount of space consistently and achieves over orders of magnitudes space reduction.

  The reason that CUR performs much worse for high accuracy estimation is that it has to keep many duplicate columns and rows in order to reach a high accuracy, while CMD decides to keep only unique columns and rows and scale them carefully to retain the accuracy estimation.

- **Network-Time**: In terms of CPU time (see Figure 3.4(b)), CMD achieves much more savings than SVD and CUR (e.g., CMD uses less 10% CPU-time compared to SVD and CUR to achieve the same accuracy 90%.). There are two reasons: first, CMD compressed sampled rows and columns, and second, no expensive SVD is needed on the entire matrix (graph). CUR is as bad as SVD for high accuracy estimation due to excessive computation cost on duplicate samples. The majority of time spent by CUR is in performing SVD on the sampled columns (see the algorithm in 13)[6] . Note that the overtaking point in Figure 3.4(b) (around 0.8 accuracy).This implies that sometimes (0.8 accuracy in this case) even the space consumption of CUR becomes more than SVD, but because of the efficient computation steps, CUR is still faster than SVD. Again, CMD is the winner throughout.

DBLP Dataset.



(a) space vs accuracy (b) time vs. accuracy

Figure 3.5: DBLP: CMD uses the least amount of space and time. Notice the huge space and time that SVD requires. The gap between CUR and CMD is smaller because the underlying distribution of data is less skewed, which implies fewer duplicate samples are required.

- **DBLP-Space**: We observe similar performance trends using the DBLP dataset. CMD requires the least amount of space among the three methods (see Figure 3.5(a)). Notice that we do not show the high-accuracy points for SVD, because of its huge memory requirements.

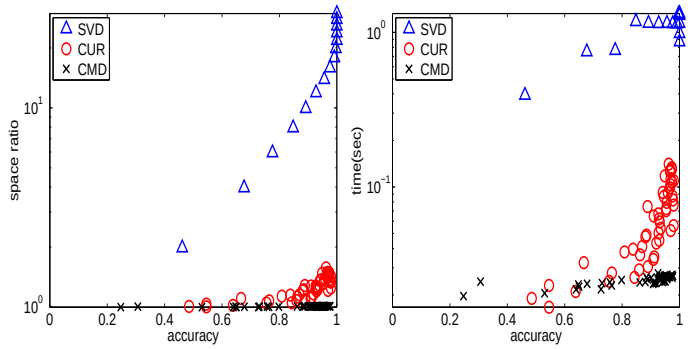
[6]We use LinearTimeCUR algorithm in [54] for all the comparisons. There is another ConstantTimeCUR algorithm proposed in [54], however, the accuracy approximation of it is too low to be useful in practice, which is left out of the comparison.

Overall, SVD uses more than 2000X more space than the original data, even with a low accuracy (less than 30%). The huge gap between SVD and the other two methods is mainly because: (1) the data distribution of DBLP is not as skewed as that of network flow, therefore the low-rank approximation of SVD needs more dimensions to reach the same accuracy, and (2) the dimension for DBLP (428,398) is much bigger than that for network flow (21,837), which implies a much higher cost to store the result for DBLP than for network flow. These results demonstrates the importance of preserving sparsity in the result.

On the other hand, the difference between CUR and CMD in DBLP becomes smaller than that with network flow trace (e.g., CMD is 40% better than CUR for DBLP instead of an order of magnitude better for network.). The reason is that the data distribution is less skewed. There are fewer duplicate samples in CUR.

- **DBLP-Time**: The computational cost of SVD is much higher compared to CMD and CUR (see Figure 3.5(b)). This is because the underlying matrix is denser and the dimension of each singular vector is bigger, which explains the high operation cost on the entire graph. CMD, again, has the best performance in CPU time for DBLP data.

Enron Dataset.



(a) space vs accuracy (b) time vs. accuracy

Figure 3.6: Enron: CMD uses the least amount of space and time. Notice the huge space and time that SVD requires. The gap between CUR and CMD is smaller because the underlying distribution of data is less skewed, which implies fewer duplicate samples are required. The overall accuracy is higher than DBLP and Network because Enron data exhibits a low-rank structure that can be summarized well using a few basis vectors.

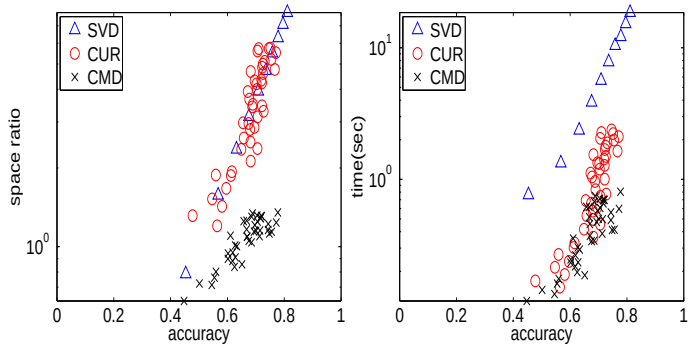
- **Enron-Space**: We observe similar performance trends using the DBLP dataset. CMD requires the least amount of space among the three methods (see Figure 3.6(a)).

  Due to the low-rank structure of Enron data, SVD can achieve fairly good approximation with a low dimensionality (90% accuracy with 6 dimensions). However, SVD still uses more than 10X more space than the original data because the singular vectors are dense. While CUR and CMD preserves sparsity in the final result, it, therefore, requires much smaller space. These results again illustrates the importance of preserving sparsity in the result.

  Because of smoother distribution (versus DBLP), the difference between CUR and CMD in Enron is small than that with network flow trace (e.g., CMD is 2X better than CUR for Enron instead of an order of magnitude better for network.).

- **Enron-Time**: The computational cost of SVD is much higher compared to CMD and CUR (see Figure 3.6(b)). This is because the underlying matrix is denser and the dimension of each singular vector is bigger, which explains the high operation cost on the entire graph. CMD, again, has the best performance in CPU time for Enron data.

Transaction Dataset.



(a) space vs accuracy (b) time vs. accuracy

Figure 3.7: Transaction: CMD uses the least amount of space and time. The space and time required by CUR increases fast as the accuracy increases due to the duplicated columns and rows.

- **Transaction-Space**: Once again, CMD requires the least amount of space among the three methods (see Figure 3.7(a)).

Unlike the rest of datasets, Transaction data does not have a low-rank structure. Because of that, a fairly high dimensionality is required for SVD as well as CUR, CMD in order to achieve relatively high accuracy. In particular, CUR requires about the same space as SVD, while CMD still performs the best but the gap (about 5X difference) is smaller that that of the other datasets due to the lack of low rank structure. The message is that all three methods aim at the datasets with low rank structure. Without that, simple linear dimensionality reductions such as SVD, CUR and CMD are often no longer the proper tools.

- **Transaction-Time**: Overall CMD is still the fastest among the three on Transaction data, but the gap is small. The reason is that because of the lack of low rank structure, an almost full SVD has to be performed for all three methods in order to capture enough energy in the original matrix. As a result, the dominant cost lies in SVD computation.

Accuracy Estimation

In this section, we evaluate the performance of our accuracy estimation algorithm described in Section 3.2.3. Note the estimation of relative SSEs is only required with CUR and CMD. For SVD, the SSEs can be computed easily using the sum of the singular values.

Unlike SVD, CUR and CMD do not have a shortcut to compute the relative SSE, or the accuracy (i.e., 1- relative SSE). Fortunately, there is a way to approximate that as described in Section 3.2.3.

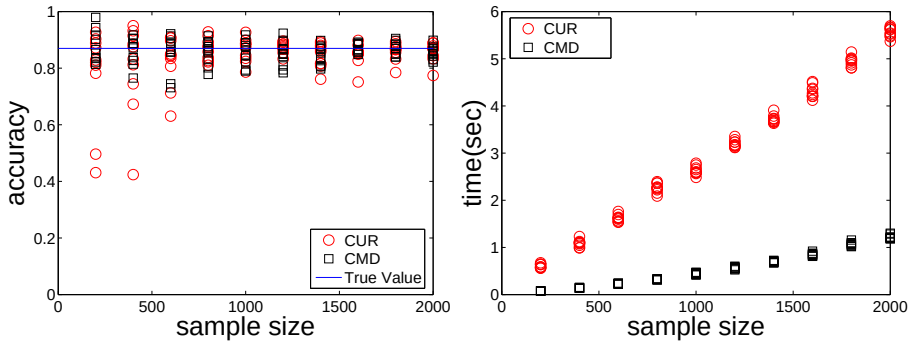
Using the Network dataset, we plot in Figure 3.8 (a) both the estimated accuracy and the true accuracy by varying the sample size used for error estimation (i.e., number of columns or rows). Similar trends observed for all the other datasets, therefore, the results are omitted for brevity.

For every sample size, we repeat the experiment 10 times with both CUR and CMD, and show all the 20 estimated errors. The targeted low-rank approximation accuracy is set to 90%.

We observe that the estimated accuracies (i.e., computed based on the estimated error using $1 - \tilde{\text{SSE}}$) are close to the true accuracy (*unbiased*), with the variance dropping quickly as the sample size increases (small variance).

The time used for estimating the error is linear to the sample size (see Figure 3.8). We observe that CMD requires much smaller time to compute the estimated error than CUR (5 times faster). For both methods, the error estimation can finish within several

seconds. As a comparison, it takes longer than 1,000 seconds to compute a true accuracy for the same matrix. Thus for applications that can tolerate a small amount of inaccuracy in accuracy computation, our estimation method provides a solution to dramatically reduce the computation latency.



(a) Estimated accuracy (b) Estimation latency

Figure 3.8: Accuracy Estimation: (a) The estimated accuracy are very close to the true accuracy; (b) Accuracy estimation performs much faster for CMD than CUR

Robustness to Sparsification

We now proceed to evaluate our framework, beginning with the performance of the sparsification module. As described in 14, our proposed sparsification constructs an approximate matrix instead of using the true matrix. Our goal is thus to see how much accuracy we lose using sparsified matrices, versus using the true matrix constructed from all available data. We use the Network dataset. Figure 3.9 plots the sparsification ratio p vs. accuracy of the final approximation output by the entire framework, using the three different methods, SVD, CUR, and CMD. In other words, the accuracy is computed with respect to the true adjacency matrix constructed with all updates. We also plot the accuracy of the sparsified matrices compared with the true matrices. This provides an upper bound as the best accuracy that could be achieved ideally after sparsification.

Once we get the sparsified matrices, we fix the amount of space to use for the three different methods. For example, the curve on the top of Figure 3.9 is the accuracy of the approximate adjacency matrix to the true one, which is also the upper bound of all the other methods. We observe that the accuracy of CMD is very close to the upper bound ideal case. The accuracies achieved by all three methods do not drop much as the sparsification ratio decreases, suggesting the robustness of these methods to missing data. These

results indicate that we can dramatically reduce the number of raw event records to sample without affecting the accuracy much.

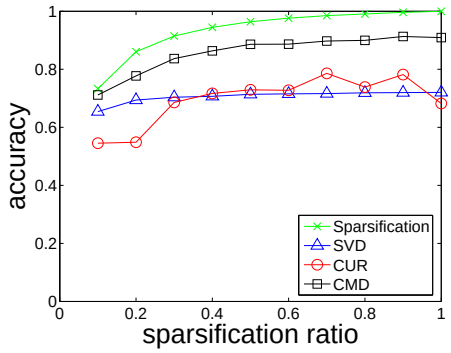


Figure 3.9: Sparsification: it incurs small performance penalties, for all methods.

In summary, CMD consistently outperforms traditional method SVD and the state of art method CUR on all experiments. Next we will illustrate some applications of CMD in practice.

3.2.5 Applications and Mining Case Study

In this section, we illustrate how CMD and our framework can be applied in practice using two example applications: (1) anomaly detection on a single matrix (i.e., a *static graph*) and (2) storage, historical analysis, and real-time monitoring of multiple matrices evolving over time (i.e., *dynamic graphs*). For each application, we perform case studies using real data sets.

Anomaly Detection

Given a large static graph, how do we efficiently determine if certain nodes are outliers, that is, which rows or columns are significantly different than the rest? And how do we identify them? In this section, we consider anomaly detection on a static graph, with the goal of finding abnormal rows or columns in the corresponding adjacency matrix. CMD can be easily applied for mining static graphs. We can detect static graph anomalies using the SSE along each row or column as the potential indicators after matrix decomposition.

A real world example is to detect abnormal hosts from a static traffic matrix, which has often been an important but challenging problem for system administrators. Detecting

Ratio	20%	40%	60%	80%	100%
Source IP	0.9703	0.9830	0.9727	0.8923	0.8700
Destination IP	0.9326	0.8311	0.8040	0.7220	0.6891

Table 3.3: Network anomaly detection: precision is high for all sparsification ratios (the detection false positive rate $= 1 -$ precision).

abnormal behavior of host communication patterns can help identify malicious network activities or mis-configuration errors. In this case study, we focus on the static source-destination matrices constructed from network traffic (every column and row corresponds to a source and destination, respectively), and use the SSEs on rows and columns to detect the following two types of anomalies:

- **Abnormal source hosts**: Hosts that send out abnormal traffic, for example, port-scanners, or compromised "zombies". One example of abnormal source hosts are scanners that send traffic to a large number of different hosts in the system. Scanners are usually hosts that are already compromised by malicious attacks such as worms. Their scanning activities are often associated with further propagating the attack by infecting other hosts. Hence it is important to identify and quarantine these hosts accurately and quickly. We propose to flag a source host as "abnormal", if its row has a high reconstruction error. the corresponding row is significantly larger than the rest of hosts, i.e., $||\mathbf{A}(i,:) - A(\tilde{i},:)|| > \alpha$ where $\mathbf{A}(i,:)$ is the i-th row of matrix $\mathbf{A}$.
- **Abnormal destination hosts**: Examples include targets of *denial of service* attacks (DoS), or targets of *distributed denial of service* (DDoS). Hosts that receive abnormal traffic. An example abnormal destination host is one that has been under denial of service attacks by receiving a high volume of traffic from a large number of source hosts. Similarly, our criterion is the (column) reconstruction error. Similarly, an abnormal destination host can be detected if the reconstruction error of the corresponding column is large, i.e., $||\mathbf{A}(:,j) - \mathbf{A}(:,\tilde{j})|| > \alpha$, where $\mathbf{A}(:,j)$ is the j-th column of matrix $\mathbf{A}$.

Experimental setup: We randomly pick an adjacency matrix from normal periods with no known attacks. Due to the lack of detailed anomaly information, we manually inject anomalies into the selected matrix using the following method: (1)*single entry anomaly:* We randomly select 100 non-zero matrix entries, and increase their values k times, where k is an experiment parameter defined as the *increasing ratio* ranging from 1 to 20. (1)*Abnormal source hosts:* We randomly select a source host and then set all the corresponding

row entries to 1, simulating a scanner host that sends flows to every other host in the network. (2)*Abnormal destination hosts:* Similar to scanner injection, we randomly pick a column and set 90% of the corresponding column entries to 1, assuming the selected host is under denial of service attack from a large number of hosts.

There are two additional input parameters: sparsification ratio and the number of sampled columns and rows. We vary the sparsification ratio from 20% to 100% and set the sampled columns (and rows) to 500.

Performance metrics: We use detection precision as our metric. We sort hosts based their row SSEs and column SSEs, and extract the smallest number of top ranked hosts (say k hosts) that we need to select as suspicious hosts, in order to detect all injected abnormal host (i.e., recall = 100% with no false negatives). Precision thus equals $1/k$, and the false positive rate equals $1 -$ precision.

We inject only one abnormal host each time. And we repeat each experiment 100 times and take the mean.

Results: Table 3.3(a) and (b) show the precision vs. sparsification ratio for detecting *abnormal source hosts* and *abnormal destination hosts*, respectively. Although the precision remains high for both types of anomaly detection, we achieve a higher precision in detecting abnormal source hosts than detecting the abnormal destinations. One reason is that scanners talk to almost all other hosts while not all hosts will launch DOS attacks to a targeted destination. In other words, there are more abnormal entries for a scanner than for a host under denial of service attack. Most of the false positives are actually from servers and valid scanning hosts, which can be easily removed based on the prior knowledge of the network structure.

Our purpose of this case study is not to present the best algorithm for anomaly detection, but to show the great potential of using efficient matrix decomposition as a new method for anomaly detection. Such approach may achieve similar or better performance than traditional methods but without expensive analysis overhead.

Time-Evolving Monitoring

In this section, we consider the application of monitoring dynamic graphs. Using our proposed process, we can dynamically construct and analyze time-evolving graphs from real-time application data. One usage of the output results is to provide compact storage for historical analysis. In particular, for every timestamp t, we can store only the sampled columns and rows as well as the estimated approximation error $S\tilde{S}E_t$ in the format of a tuple $(\mathbf{C}_t, \mathbf{R}_t, S\tilde{S}E_t)$.

Furthermore, the approximation error (SSE) is useful for monitoring dynamic graphs, since it gives an indication of how much the global behavior can be captured using the samples. In particular, we can fix the sparsification ratio and the CMD sample size, and then compare the approximation error over time. A timestamp with a large error or a time interval (multiple timestamps) with a large average error implies structural changes in the corresponding graph, and is worth additional investigation.

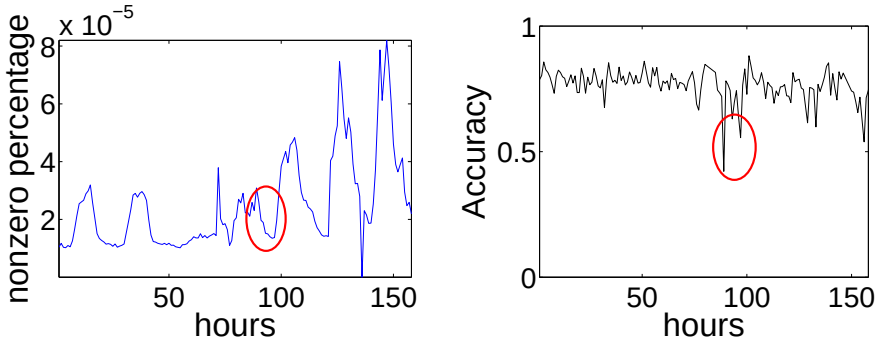
To make our discussion concrete, we illustrate the application of time-evolving monitoring using both the network traffic matrices and the DBLP matrices.

Network over time: For network traffic, normal host communication patterns in a network should roughly be similar to each other over time. A sudden change of approximation accuracy (i.e., $1 - \tilde{SE}$) suggests structural changes of communication patterns since the same approximation procedure can no longer keep track of the overall patterns. We can thus monitor dynamic host-by-host matrices to detect the existence of abnormal communication patterns and alert the administrators.

Figure 3.10(b) shows the approximation accuracy over time, using 500 sampled rows and columns without duplicates (out of 21K rows/columns). The overall accuracy remains high. But an unusual accuracy drop occurs during the period from hour 80 to 100. We manually investigate into the trace further, and indeed find the onset of worm-like hierarchical scanning activities. For comparison, we also plot the percentage of non-zero matrix entries generated each hour over time in Figure 3.10(a), which is a standard method for network anomaly detection based on traffic volume or distinct number of connections. Although such statistic is relatively easier to collect, the total number of traffic entries is not always an effective indicator of anomaly. Notice that during the same period of hour 80 to 100, the percentage of non-zero entries is not particularly high. Only when the infectious activity became more prevalent (after hour 100), we can see an increase of the number of non-zero entries. Our framework can thus potentially help detect abnormal events at an earlier stage.

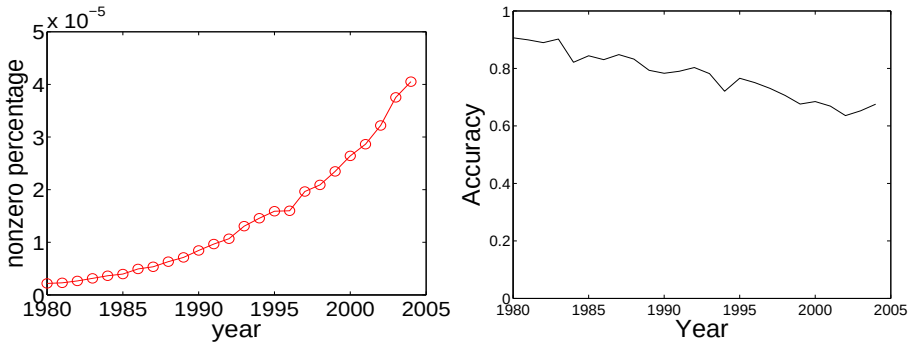
DBLP over time: For the DBLP setting, we monitor the accuracy over the 25 years by sampling 300 conferences (out of 3,659 conferences) and 10 K authors (out of 428K authors) each year. Figure 3.11(b) shows that the accuracy is high initially, but slowly drops over time. The interpretation is that the number of authors and conferences (nonzero percentage) increases over time (see Figure 3.11(a)), suggesting that we need to sample more columns and rows to achieve the same high approximation accuracy.

In summary, our exploration of both applications suggest that CMD has great potential for discovering patterns and anomalies for dynamic graphs too.



(a) Nonzero entries over time (b) Accuracy over time

Figure 3.10: Network flow over time: we can detect anomalies by monitoring the approximation accuracy (b), while traditional method based on traffic volume cannot do (a).



(a) Nonzero entries over time (b) Accuracy over time

Figure 3.11: DBLP over time: The approximation accuracy drops slowly as the graphs grow denser.

3.2.6 Summary

We studied the problem of efficiently discovering patterns and anomalies from large graphs, like traffic matrices, both in the static case, as well as when they evolve over time. The contributions are the following:

- *New matrix decomposition method:* CMD generates low-rank, sparse matrix approximations. We proved that CMD gives exactly the same accuracy like CUR, but in much less space (Theorem 3.1).

- *High-rate time evolving graphs:* Extension of CMD, with careful sampling, and fast estimation of the reconstruction error, to spot anomalies.

- *Speed and space:* Experiments on several real datasets, one of which is >100Gb of real traffic data, show that CMD achieves up to *10 times* less space and less time than the competition.

- *Effectiveness:* CMD found anomalies that were verified by domain experts, like the anomaly in Figure 3.10

3.3 GraphScope: Parameter-Free Mining of Large Time-Evolving Graphs

How to monitor the changes of communities over time? The aspect of time has begun to receive attention [91, 119]. Some examples of the time-evolving graphs include: (a) Network traffic events indicate ongoing communication between source and destination hosts, similar to the `NETWORK` dataset in our experiments; (b) Email networks associate a sender and a recipient at a given date, like the `ENRON` data set [2] in the experiments; (c) Call detail records in telecommunications networks associate a caller with a callee. The set of all conversation pairs over each week forms a graph that evolves over time, like the publicly available '`CELLPHONE`' dataset of MIT users calling each other [1]; (d) Transaction data: in a financial institution, who accessed what account, and when [18]; (e) In a database compliance setting [12], again we need to record which user accessed what data item and when.

Large amounts of data such as those in the above examples are continuously collected and patterns are also changing over time. Therefore, batch methods for pattern discovery are not sufficient. We need tools that can incrementally find the communities and monitor the changes. In summary, there are two sub-problems involved:

(P1) **Community discovery:** Which groups or communities of nodes are associated with each other?

(P2) **Change detection:** When does the community structure change and how to quantify the change?

Moreover, we want to answer these questions (a) *without* requiring any user-defined parameters, and (b) in a *streaming* fashion.

For example, we want to answer questions such as: How do the network hosts interact with each other? What kind of host groups are there, e.g., inactive/active hosts; servers;

scanners? Who emails whom? Do the email communities in a organization such as ENRON remain stable, or do they change between workdays (e.g., business-related) and weekends (e.g., friend and relatives), or during major events (e.g.,the FBI investigation and CEO resignation)?

We propose GraphScope, which addresses both of the above problems simultaneously. More specifically, GraphScope is an efficient, adaptive mining scheme on time-evolving graphs. Unlike many existing techniques, it requires no user-defined parameters, and it operates completely automatically, based on the Minimum Description Length (MDL) principle. Furthermore, it adapts to the dynamic environment by automatically finding the communities and determining good *change-points* in time.

In this section we consider bipartite graphs, which treat source and destination nodes separately (see example in Figure 3.12). As will become clear later, we discover separate source and destination partitions, which are desirable in several application domains. Nonetheless, our methods can be easily modified to deal with unipartite graphs, by constraining the source-partitions to be the same as the destination-partitions [26].

The main insight of dealing with such graphs is to group "similar" sources together into *source-groups* (or *row-groups*), and also "similar" destinations together, into *destination-groups* (or *column-groups*). Examples in Section 3.3.4 show how much more orderly (and easier to compress) the adjacency matrix of a graph is, after we strategically re-order its rows and columns. The exact definition of "similar" is actually simple, and rigorous: the most similar source-partition for a given source node is the one that leads to small encoding cost (see Section 3.3.2 for more details).

Furthermore, if these communities (source and destination partitions) do not change much over time, consecutive snapshots of the evolving graphs have similar descriptions and can also be grouped together into a time segment, to achieve better compression. Whenever a new graph snapshot cannot fit well into the old segment (in terms of compression), GraphScope introduces a change-point, and starts a new segment at that time-stamp. Those change points often detect drastic discontinuities in time.

Contributions Our proposed approach, GraphScope, monitors communities and their changes in a stream of graphs efficiently. It has the following key properties:

- *Parameter-free:* GraphScope is completely automatic, requiring no parameters from the user (like number of communities, thresholds to assess community drifts, and so on). Instead, it is based on sound information-theoretic principles, specifically, MDL.

- *Adaptive:* It can effectively track communities over time, discovering both communities as well as change-points in time, that agree with human intuition.
- *Streaming:* It is fast, incremental and scalable for the streaming environment.

We demonstrate the efficiency and effectiveness of our approach in discovering and tracking communities in real graphs from several domains.

3.3.1 Problem definition

In this section, we formally introduce the necessary notations and formulate the problems.

Notation and definition. Calligraphic letters always denote *graph streams* or *graph stream segments* (consisting of one or more graph snapshots), while individual graph snapshots are denoted by non-calligraphic, upper-case letters. Superscripts in parentheses denote either timestamps t or graph segment indices s, accordingly. Similarly, subscripts denote either individual nodes i, j or node partitions p, q.

Definition 3.1 (Graph stream)**.** *A* graph stream $\mathcal{G}$ *is a sequence of graphs* $G^{(t)}$*, i.e.,*

$$\mathcal{G} := \{G^{(1)}, G^{(2)}, \ldots, G^{(t)}, \ldots\},$$

which grows indefinitely over time. Each of these graphs links m *source nodes to* n *destination nodes.*

For example in Figure 3.12, the first row shows the first three graphs in a graph stream, where $m = 4$ and $n = 3$. Furthermore, the graphs are represented as sparse matrices in the bottom of Figure 3.12 (a black entry is 1, which indicates an edge between the corresponding nodes; likewise a white entry is 0).

In general, each graph may be viewed as an $m \times n$ binary adjacency matrix, where rows $1 \leq i \leq m$ correspond to source nodes and columns $1 \leq j \leq n$ correspond to destination nodes. We use sparse representation of the matrix (i.e., only non-zero entries are stored) whose space consumption is similar to adjacency list representation. Without loss of generality, we assume m and n are the same for all graphs in the stream; if not, we can introduce all-zero rows or columns in the adjacency matrices.

One of our goals is to track how the structure of the graphs $G^{(t)}$, $t \geq 1$, evolves over time. To that end, we will group consecutive timestamps into segments.

Sym.	Definition
$\mathcal{G}$, $\mathcal{G}^{(s)}$	Graph stream, Graph segment
t	Timestamp, $t \geq 1$.
m, n	Number of source(destination) nodes.
$G^{(t)}$	Graph at time t ($m \times n$ adjacency matrix).
i, j	Node indices, $1 \leq i \leq m$, $1 \leq j \leq n$.
$G^{(t)}_{i,j}$	Indicator for edge (i, j) at time t.
s	Graph segment index, $s \geq 1$.
t_s	Starting time of the s-th segment.
k_s,ℓ_s	Number of source (dest.) partitions for segment s.
p, q	Partition indices, $1 \leq p \leq k_s$, $1 \leq q \leq \ell_s$.
$I^{(s)}_p$	Set of sources belonging to the p-th partition, during the s-th segment.
$J^{(s)}_q$	Similar to $I^{(s)}_p$, but for destination nodes.
$m^{(s)}_p$	Source partition size, $m^{(s)}_p \equiv \lvert I^{(s)}_p \rvert$, $1 \leq p \leq k_s$.
$n^{(s)}_p$	Dest. partition size, $n^{(s)}_p \equiv \lvert J^{(s)}_p \rvert$, $1 \leq p \leq \ell_s$.
$\mathcal{G}^{(s)}_{p,q}$	Subgraphs induced by p-th and q-th partitions of segment s, i.e., subgraph segment
$\lvert\mathcal{G}^{(s)}_{p,q}\rvert$	Size of subgraphs segment, $\lvert\mathcal{G}^{(s)}_{p,q}\rvert := m^{(s)}_p n^{(s)}_q (t_{s+1} - t_s)$.
$\lvert E\rvert^{(s)}_{p,q}$	Number of edges in $\mathcal{G}^{(s)}_{p,q}$
$\rho^{(s)}_{p,q}$	Density of $\mathcal{G}^{(s)}_{p,q}$, $\frac{\lvert E\rvert^{(s)}_{p,q}}{\lvert\mathcal{G}^{(s)}_{p,q}\rvert}$
$H(.)$	Shannon entropy function

Table 3.4: Definitions of symbols

Definition 3.2 (Graph stream segment)**.** *The set of graphs between timestamps t_s and $t_{s+1} - 1$ (inclusive) consist the s-th segment $\mathcal{G}^{(s)}$, $s \geq 1$, which has length $t_{s+1} - t_s$,*

$$\mathcal{G}^{(s)} := \{G^{(t_s)}, G^{(t_s+1)}, \ldots, G^{(t_{s+1}-1)}\}.$$

Intuitively, a "*graph stream segment*" (or just "*graph segment*") is a set of consecutive graphs in a graph stream. For example in Figure 3.12, $\mathcal{G}^{(1)}$ is a graph segment consisting of two graph $G^{(1)}$ and $G^{(2)}$.

Next, within each segment, we will partition the source and destination nodes into source partitions and destination partitions, respectively.

Definition 3.3 (Graph segment partitions)**.** *For each segment $s \geq 1$, we partition source nodes into k_s source partitions and destination nodes into ℓ_s destination partitions. The*

set of source nodes that are assigned into the p-th source partition $1 \leq p \leq k_s$ is denoted by $I_p^{(s)}$. Similarly, the set of destination nodes assigned to the q-th destination partition is denoted by $J_q^{(s)}$, for $1 \leq q \leq \ell_s$.

The sets $I_p^{(s)}$ ($1 \leq p \leq k_s$) partition the source nodes, in the sense that $I_p^{(s)} \cap I_{p'}^{(s)} = \emptyset$ for $p \neq p'$, while $\bigcup_p I_p^{(s)} = \{1, \ldots, m\}$. Similarly, the sets $J_q^{(s)}$ ($1 \leq q \leq l_s$) partition the destination nodes. For example in Figure 3.12, the first graph segment $\mathcal{G}^{(1)}$ has source partitions $I_1^{(1)} = \{1, 2\}$, $I_2^{(1)} = \{3, 4\}$, and destination partitions $J_1^{(1)} = \{1\}$, $J_2^{(1)} = \{2, 3\}$ where $k_1 = 2$, $\ell_1 = 2$. We can similarly define source and destination partition for the second graph segment $\mathcal{G}^{(2)}$, where $k_2 = 3$, $\ell_2 = 3$.

Problem formulation. The ultimate goals are to find communities on time-evolving graphs along with the *change-points*, if any. Thus, the following two problems need to be addressed.

Problem 3.1 (Partition Identification)**.** *Given a graph stream segment $\mathcal{G}^{(s)}$, how to find good partitions of source and destination nodes, which summarize the fundamental community structure.*

The meaning of "good" will be made precise in the next section, which formulates our cost objective function. However, to obtain an answer for the above problem, two important sub-questions need to be answered:

- How to assign the m source and n destination nodes into k_s source and ℓ_s destination partitions?
- How to determine k_s and ℓ_s?

Problem 3.2 (Time Segmentation)**.** *Given a graph stream $\mathcal{G}$, how can we incrementally construct graph segments, by selecting good change points t_s.*

Section 3.3.3 presents the algorithms and formalizes the notion of "good" for both problems above. We name the whole analytic process *GraphScope*.

3.3.2 GraphScope encoding

Our mining framework is based on one form of the Minimum Description Length (MDL) principle and employs a lossless encoding scheme for a graph stream. Our objective function estimates the number of bits needed to encode the full graph stream so far. Our

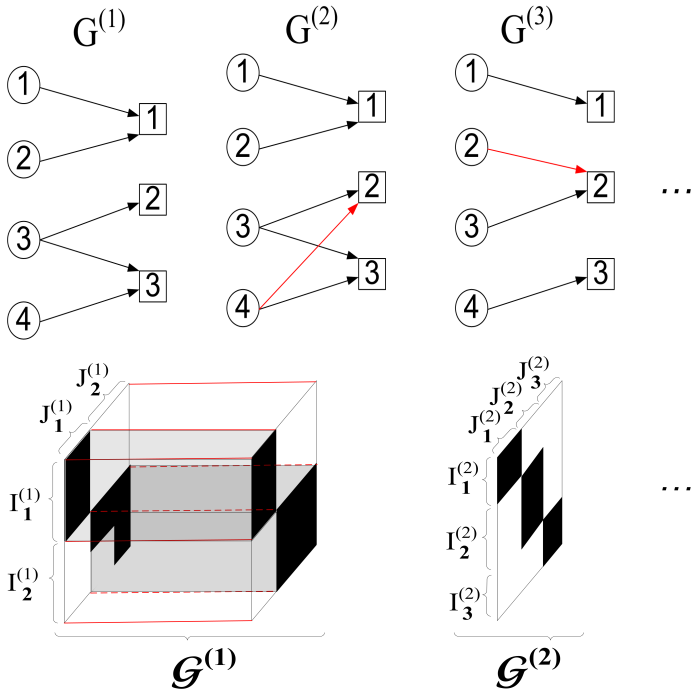


Figure 3.12: Notation illustration: A graph stream with 3 graphs in 2 segments. First graph segment consisting of $G^{(1)}$ and $G^{(2)}$ has two source partitions $I_1^{(1)} = \{1, 2\}$, $I_2^{(1)} = \{3, 4\}$; two destination partitions $J_1^{(1)} = \{1\}$, $J_2^{(1)} = \{2, 3\}$. Second graph segment consisting of $G^{(3)}$ has three source partitions $I_1^{(2)} = \{1\}$, $I_2^{(2)} = \{2, 3\}$, $I_3^{(2)} = \{4\}$; three destination partitions $J_1^{(2)} = \{1\}$, $J_2^{(2)} = \{2\}$, $J_2^{(2)} = \{3\}$.

proposed encoding scheme takes into account both the community structures, as well as their change points in time, in order to achieve a concise description of the data. The fundamental trade-off that decides the "best" answers to problems 1 and 2 in Section 3.3.1 is between (i) the number of bits needed to describe the communities (or, partitions) and their change points (or, segments) and (ii) the number of bits needed to describe the individual edges in the stream, given this information.

We begin by first assuming that the change-points as well the source and destination partitions for each graph segment are given, and we show how to estimate the bit cost to describe the individual edges (part (ii) above). Next, we show how to incorporate the partitions and segments into an encoding of the entire stream (part (i) above).

Graph encoding

In this paper, a graph is presented as a m-by-n binary matrix. For example in Figure 3.12, $G^{(1)}$ is represented as

$$G^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.4}$$

Conceptually, we can store a given binary matrix as a binary string with length mn, along with the two integers m and n. For example, equation 3.4 can be stored as 1100 0010 0011 (in column major order), along with two integers 4 and 3.

To further save space, we can adopt some standard lossless compression scheme (such as Huffman coding, or arithmetic coding [39]) to encode the binary string, which formally can be viewed as a sequence of realizations of a binomial random variable X. The code length for that is accurately estimated as $mnH(X)$ where $H(X)$ is the entropy of variable X. For notational convenience, we also write that as $mnH(G^{(t)})$. Additionally, three integers need to be stored: the matrix sizes m and n, and the number of ones in the matrix (i.e., the number of edges in the graph) denoted as $|E|$ [7]. The cost for storing three integers is $\log^\star |E| + \log^\star m + \log^\star n$ bits, where $\log^\star$ is the universal code length for an integer[8]. Notice that this scheme can be extended to a sequence of graphs in a segment.

More generally, if the random variable X can take values from the set M, with size $|M|$ (a multinomial distribution), the entropy of X is

$$H(X) = -\sum_{x \in M} p(x) \log p(x).$$

where $p(x)$ is the probability that $X = x$. Moreover, the maximum of $H(X)$ is $\log |M|$ when $p(x) = \frac{1}{|M|}$ for all $x \in M$ (pure random, most difficult to compress); the minimum is 0 when $p(x) = 1$ for a particular $x \in M$ (deterministic and constant, easiest to compress). For the binomial case, if all symbols are all 0 or all 1 in the string, we do not have to store anything because by knowing the number of ones in the string and the sizes of matrix, the receiver is already able to decode the data completely.

With this observation in mind, the goal is to organize the matrix (graph) into some homogeneous sub-matrices with low entropy and compress them separately, as we will describe next.

[7] $|E|$ is needed for computing the probability of ones or zeros, which is required for several encoding scheme such as Huffman coding

[8] To encode a positive integer x, we need $\log^\star x \approx \log_2 x + \log_2 \log_2 x + \ldots$, where only the positive terms are retained and this is the optimal length, if the range of x is unknown [111]

Graph Segment encoding

Given a graph stream segment $\mathcal{G}^{(s)}$ and its partition assignments, we can precisely compute the cost for transmitting the segment as two parts: 1) *Partition encoding cost*: the model complexity for partition assignments, 2) *Graph encoding cost*: the actual code for the graph segment.

Partition encoding cost. The description complexity for transmitting the partition assignments for graph segment $\mathcal{G}^{(s)}$ consists of the following terms:

First, we need to send the number of source and destination nodes m and n using $\log^\star m + \log^\star n$ bits. Note that, this term is constant, which has no effect on the choice of final partitions.

Second, we shall send the number of source and destination partitions which is $\log^\star k_s + \log^\star \ell_s$.

Third, we shall send the source and destination partition assignments. To exploit the non-uniformity across partitions, the encoding cost is $mH(P) + nH(Q)$ where P is a multinomial random variable with the probability $p_i = \frac{m_i^{(s)}}{m}$ and $m_i^{(s)}$ is the size of i-th source partition $1 \leq i \leq k_s$). Similarly, Q is another multinomial random variable with $q_i = \frac{n_i^{(s)}}{n}$ and $n_i^{(s)}$ is the size of i-th destination partition, $1 \leq i \leq \ell_s$.

For example in Figure 3.12, the partition sizes for first segment $\mathcal{G}^{(1)}$ are $m_1^{(1)} = m_2^{(1)} = 2$, $n_1^{(1)} = 1$, and $n_2^{(1)} = 2$; the partition assignments for $\mathcal{G}^{(1)}$ costs $-4(\frac{2}{4}\log(\frac{2}{4}) + \frac{2}{4}\log(\frac{2}{4})) - 3(\frac{1}{3}\log(\frac{1}{3}) + \frac{2}{3}\log(\frac{2}{3}))$ bits.

In summary, the partition encoding cost for graph segment $\mathcal{G}^{(s)}$ is

$$C_p^{(s)} := \log^\star m + \log^\star n + \log^\star k_s + \log^\star \ell_s + mH(P) + nH(Q) \tag{3.5}$$

where P and Q are multinomial random variables for source and destination partitions, respectively.

Graph encoding cost. After transmitting the partition encoding, the actual graph segment $\mathcal{G}^{(s)}$ is transmitted as $k_s\ell_s$ subgraph segments. To facilitate the discussion, we define the entropy term for a subgraph segment $\mathcal{G}_{p,q}^{(s)}$ as

$$H(\mathcal{G}_{p,q}^{(s)}) = -\big(\rho_{p,q}^{(s)} \log \rho_{p,q}^{(s)} + (1 - \rho_{p,q}^{(s)}) \log(1 - \rho_{p,q}^{(s)})\big) \tag{3.6}$$

where $\rho_{p,q}^{(s)} = \frac{|E|_{p,q}^{(s)}}{|\mathcal{G}_{p,q}^{(s)}|}$ is the density of subgraph segment $\mathcal{G}_{p,q}^{(s)}$. Intuitively, $H(\mathcal{G}_{p,q}^{(s)})$ quantifies

how difficult it is to compress the subgraph segment $\mathcal{G}^{(s)}_{p,q}$. In particular, if the entire subgraph segment is all 0 or all 1 (the density is exactly 0 or 1), the entropy term becomes 0.

With this, the graph encoding cost is

$$C^{(s)}_g := \sum_{p=1}^{k_s} \sum_{q=1}^{\ell_s} \big(\log^\star |E|^{(s)}_{p,q} + |\mathcal{G}^{(s)}_{p,q}| \cdot H(\mathcal{G}^{(s)}_{p,q}) \big). \tag{3.7}$$

where $|E|^{(s)}_{p,q}$ is the number of edges in the (p, q) sub-graphs of segment s; $|\mathcal{G}^{(s)}_{p,q}|$ is the size of sub-graph segment, i.e, $m^{(s)}_p n^{(s)}_q (t_{s+1} - t_s)$, and $H(\mathcal{G}^{(s)}_{p,q})$ is the entropy of the subgraph segment defined in equation 3.6.

In the sub-graph segment $\mathcal{G}^{(1)}_{2,2}$ of Figure 3.12, the number of edges $|E|^{(1)}_{2,2} = 3{+}4$, $\mathcal{G}^{(1)}_{2,2}$ has the size $|\mathcal{G}^{(1)}_{2,2}| = 2{\times}\,2{\times}2$, the density $\rho^{(1)}_{2,2} = \frac{7}{8}$, and the entropy $H(\mathcal{G}^{(1)}_{2,2}) = -(\frac{7}{8}\log\frac{7}{8}{+}\frac{1}{8}\log\frac{1}{8})$.

Putting everything together, we obtain the segment encoding cost as the follows:

Definition 3.4 (Segment encoding cost)**.**

$$C^{(s)} := \log^\star(t_{s+1} - t_s) + C^{(s)}_p + C^{(s)}_g. \tag{3.8}$$

where $t_{s+1}{-}t_s$ is the segment length, $C^{(s)}_p$ is the partition encoding cost, $C^{(s)}_g$ is the graph encoding cost.

Graph stream encoding

Given a graph stream $\mathcal{G}$, we partition it into a number of graph segments $\mathcal{G}^{(s)} (s \geq 1)$ and compress each segment separately such that the total encoding cost is small.

Definition 3.5 (Total cost)**.** *The total encoding cost is*

$$C := \textstyle\sum_s C^{(s)}. \tag{3.9}$$

where $C^{(s)}$ is the encoding cost for the s-th graph stream segment.

For example in Figure 3.12, the encoding cost C up to timestamp 3 is the sum of the costs of two graph stream segments $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$.

Intuition. Intuitively, our encoding cost objective tries to decompose the graph into subgraphs that are homogeneous, i.e., close to either fully-connected (cliques) or fully-disconnected. Additionally, if such cliques are stable over time, then it places subsequent

graphs into the same segment. The encoding cost penalizes a large number of cliques or lack of homogeneity. Hence, our model selection criterion favors simple enough decompositions that adequately capture the essential structure of the graph over time.

Having defined the objective precisely in equation 3.6 and equation 3.7, the next step is to search for optimal partition and time segmentation. However, finding the optimal solution is NP-hard[9]. Next, in Section 3.3.3, we present an alternating minimization method coupled with an incremental segmentation process to perform the overall search.

3.3.3 GraphScope

In this section we describe our method, GraphScope by solving the two problems proposed in Section 3.3.1. The goal is to find the appropriate number and position of change-points, and the number and membership of source and destination partitions so that the cost of (3.9) is minimized. Exhaustive enumeration is prohibitive, and thus we resort to alternating minimization. Note that we drop the subscript s on k_s and ℓ_s whenever it is clear from the context.

Specifically, we have two steps: (a) how to find good communities (source and destination partitions), for a given set of graph snapshots that belong to the same segment. (b) when to declare a time-tick as a *change point* and start a new graph segment. We describe each next.

Partition Identification

Here we explain how to find source and destination partitions for a given graph segment $\mathcal{G}^{(s)}$. In order to do that, we need to answer the following two questions:

- How to find the best partitioning given the number of source and destination partitions?
- How to search for the appropriate number of source and destination partitions?

Next, we present the solution for each step.

Finding the best partitioning. Given the number of the best source and destination partitions k and ℓ, we want to re-group sources and destinations into the better partitions. Typically this regrouping procedure alternates between source and destination nodes. Namely,

[9]It is NP-hard since, even allowing only column re-ordering, a reduction to the TSP problem can be found [79].

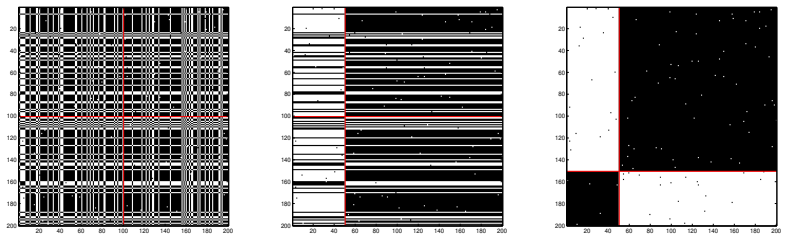
Algorithm 3.7: REGROUP(Graph Segment $\mathcal{G}^{(s)}$; partition size k,ℓ; initial partitions $I^{(s)}$, $J^{(s)}$)

Compute density $\rho^{(s)}_{p,q}$ for all p, q based on $I^{(s)}$, $J^{(s)}$. **repeat**
  forall *source* s *in* $\mathcal{G}^{(s)}$ **do**
    `// assign` s `to the most similar partition`
    s is split in ℓ parts
    compute source density p_i for each part
    assign s to source partition with the minimal encoding cost (Equation 3.11).
  Update destination partitions similarly
until *no change*;

we update the source partitions with respect to the current destination partitions, and vice versa. More specifically, we alternate the following two steps until it converges:

- **Update source partitions:** for each source (a row of the graph matrix), consider assigning it to the source partition that incurs the smallest encoding cost
- **Update destination partitions:** Similarly, for each destination (column), consider assigning it to the destination partition that yields smaller encoding cost.

The cost of assigning a row to a row-group is discussed later (see (3.11)). The pseudo-code is listed in Algorithm 3.7. The initialization of Algorithm 3.7 is discussed separately in Section 3.3.3.



original graph destination partition source partition

Figure 3.13: Alternating partition on source and destination nodes on a graph with 2 communities with size 150 and 50 plus 1% noise. For $k = \ell = 2$, the correct partitions are identified after one pass.

Determining the number of partitions. Given different values for k and ℓ, we can easily run Algorithm 3.7 and choose those leading to a smaller encoding cost. However, the search space for k and ℓ is still too large to perform exhaustive tests. Fortunately for the time-evolving graphs, the best k and ℓ are closely related to the k_{s-1} and ℓ_{s-1}. We experimented with a number of different heuristics for quickly adjusting k and ℓ, and obtained good results with Algorithm 3.8. The central idea is to do local search around some initial partition assignments, and adjust the number of partitions k and ℓ as well as the partition assignments based on the encoding cost. Figure 3.14 illustrates the search process in action. Starting the search with $k = \ell = 1$, it successfully, finds the correct number of partitions for this graph with 3 sub-matrices with size 100, 80 and 20.

Algorithm 3.8: SEARCHKL(Graph Segment $\mathcal{G}^{(s)}$; initial partition size k,ℓ; initial partitions $I^{(s)}$, $J^{(s)}$)

```
repeat
    // try to merge source partitions
    repeat
        Find the source partition pair (x, y) s.t. merging x and y gives smallest
        encoding cost for G^(s).
        if total encoding decrease then merge x,y
    until no more merge;
    // try to split source partition
    repeat
        Find source partition x with largest average entropy per node.
        foreach source s in x do
            if average entropy reduces without s then
                assign s to the new partition
        ReGroup(G^(s), updated partitions)
    until no decrease in encoding cost;
    Search destination partitions similarly
until no changes;
```

Cost computation for partition assignments. Here we present the details of how to compute the encoding cost of assigning a node to a particular partition. Our discussion focuses on assigning a source node to a source partition. The assignment for a destination node is symmetric.

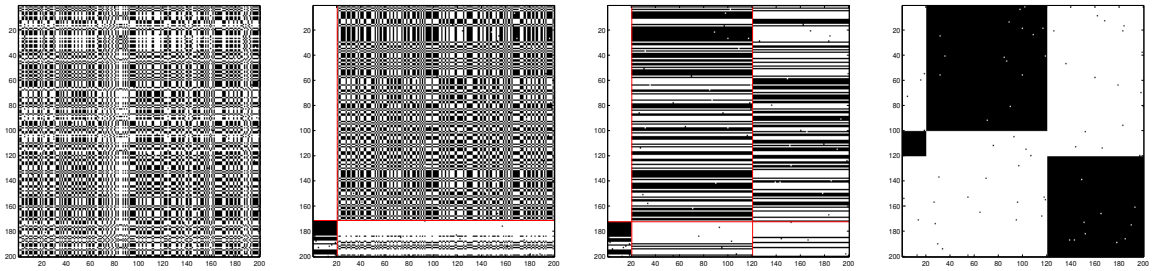


Figure 3.14: Search for best k and ℓ for a graph with 3 communities with size 100, 80, 20 plus 1 noise. The algorithm progressively improves the partition quality (reduces the encoding cost) by changing the k and ℓ.

Recall a graph segment $\mathcal{G}^{(s)}$ consists of $(t_{s+1} - t_s)$ graphs, $G^{(t_s)}, \ldots, G^{(t_{s+1}-1)}$. For example in Figure 3.12, $\mathcal{G}^{(1)}$ consists of 2 graphs, $G^{(1)}$ and $G^{(2)}$. Likewise, every source node in a graph segment $\mathcal{G}^{(s)}$ is associated with $(t_{s+1} - t_s)$ sets of edges in these $(t_{s+1} - t_s)$ graphs. Therefore, the total number of possible edges out of one source node in $\mathcal{G}^{(s)}$ is $(t_{s+1} - t_s)n$.

Furthermore, the destination partitions $J_i^{(s)}$ divide the destination nodes into ℓ disjoint sets with size $n_i^{(s)}$ ($1 \leq i \leq \ell$, $\sum_i n_i^{(s)} = n$). For example, $\mathcal{G}^{(1)}$ of Figure 3.12 has two destination partitions ($\ell = 2$), where the first destination partition $J_1^{(1)} = \{1\}$, and the second destination partition $J_2^{(1)} = \{2, 3\}$.

Similarly, all the edges from a single source node in graph segment $\mathcal{G}^{(s)}$ are also split into these ℓ sets. In $\mathcal{G}^{(1)}$ of Figure 3.12, the edges from the 4-th source node are split into two sets, where the first set $J_1^{(1)}$ has 0 edges and the second set $J_2^{(1)}$ 3 edges[10].

More formally, the edge pattern out of a source node is generated from ℓ binomial distributions $\mathbf{p}_i (1 \leq i \leq \ell)$ with respect to ℓ destination partitions. Note that $\mathbf{p}_i(1)$ is the density of the edges from that source node to the destination partition $J_i^{(s)}$, and $\mathbf{p}_i(0) = 1 - \mathbf{p}_i(1)$. In $\mathcal{G}^{(1)}$ of Figure 3.12, the 4-th source node has $\mathbf{p}_1(1) = 0$ since there are 0 edges from 4 to $J_1^{(1)} = \{1\}$, and $\mathbf{p}_1(1) = \frac{3}{4}$ since 3 out of 4 possible edges from 4 to $J_1^{(2)} = \{2, 3\}$.

One possible way of encoding the edges of one source node is based on precisely these distributions $\mathbf{p}_i$, but as we shall see later, this is not very practical. More specifically,

[10]One edge from 4 to 3 in $G^{(1)}$, two edges from 4 to 2 and 3 in $G^{(2)}$ in Figure 3.12.

using the "true" distributions $\mathbf{p}_i$, the encoding cost of the source node's edges in the graph segment $\mathcal{G}^{(s)}$ would be

$$C(\mathbf{p}) = (t_{s+1} - t_s) \sum_{i=1}^{\ell} n_i H(\mathbf{p}_i) \tag{3.10}$$

where $(t_{s+1} - t_s)$ is the number of graphs in the graph segment, n is the number of possible edges out of a source node for each graph[11], $H(\mathbf{p}_i) = \sum_{x=\{0,1\}} \mathbf{p}_i(x) \log \mathbf{p}_i(x)$ is the entropy for the each source node's partition.

In $\mathcal{G}^{(1)}$ of Figure 3.12, the number of graphs is $t_{s+1} - t_s = 3 - 1 = 2$; the number of possible edges out of the 4-th source node $n = 3$; therefore, the 4-th source node costs $2 \times 3 \times (0 + \frac{3}{4} \log \frac{3}{4} + \frac{1}{4} \log \frac{1}{4}) = 2.25$. Unfortunately, this is not practical to do so for every source node, because the model complexity is too high. More specifically, we have to store additional $m\ell$ integers in order to decode all source nodes.

The practical option is to group them into a handful of source partitions and to encode/decode one partition at a time instead of one node at a time. Similar to a source node, the edge pattern out of a source partition is also generated from ℓ binomial distributions $\mathbf{q}_i$ $(1 \leq i \leq \ell)$. Now we encode the i-th source node based on the distribution $\mathbf{q}_i$ for a partition instead of the "true" distribution $\mathbf{p}_i$ for the node. The encoding cost is

$$C(\mathbf{p}, \mathbf{q}) = (t_{s+1} - t_s) \sum_{i=1}^{\ell} n_i H(\mathbf{p}_i, \mathbf{q}_i) \tag{3.11}$$

where $H(\mathbf{p}_i, \mathbf{q}_i) = \sum_{x=\{0,1\}} \mathbf{p}_i(x) \log \mathbf{q}_i(x)$ is the cross-entropy. Intuitively, the cross-entropy is the encoding cost when using the distribution $\mathbf{q}_i$ instead of the "true" distribution $\mathbf{p}_i$. In $\mathcal{G}^{(1)}$ of Figure 3.12, the cost of assigning the 4-th node to second source partition $I_2^{(1)}$ is $2 \times 3 \times (0 + \frac{3}{4} \log \frac{7}{8} + \frac{1}{4} \log \frac{1}{8}) = 2.48$ which is slightly higher than using the true distribution that we just computed (2.25). However, the model complexity is much lower, i.e., $k\ell$ integers are needed instead of $m\ell$.

Time Segmentation

So far, we have discussed how to partition the source and destination nodes given a graph segment $\mathcal{G}^{(s)}$. Now we present the algorithm to construct the graph segments incrementally when new graph snapshots at arrive every time-tick. Intuitively, we want to group "similar" graphs from consecutive timestamps into one graph segment and encode them all together. For example, in Figure 3.12, graphs $G^{(1)}$ and $G^{(2)}$ are similar (only one different edge), and therefore we group them into one graph segment, $\mathcal{G}^{(1)}$. On the other hand, $G^{(3)}$ is quite different from the previous graphs, and hence we start a new segment $\mathcal{G}^{(2)}$ whose first member is $G^{(3)}$.

[11] $(t_{s+1} - t_s)n$ is the total number of possible edges of a source node in the graph segment

The guiding principle here is still the encoding cost. More specifically, the algorithm will combine the incoming graph with the current graph segment if there is a storage benefit, otherwise we start a new segment with that graph. The meta-algorithm is listed in Algorithm 3.9. Figure 3.15 illustrates the algorithm in action. A graph stream consists of three graphs, where $G^{(1)}$ and $G^{(2)}$ have two groups of size 150 and 50, $G^{(3)}$ three groups of size 100, 80 and 20, and every graph contains 1% noise. The algorithm decides to group $G^{(1)}$, $G^{(2)}$ into the first graph segment, and put $G^{(3)}$ into another. During this process, the correct partitions are also identified as show in the bottom of Figure 3.15. Furthermore, within each segment, the correct partition assignments are identified.

Algorithm 3.9: GRAPHSCOPE(Graph Segment $\mathcal{G}^{(s)}$; Encoding cost c_o; New Graph $G^{(t)}$

output: updated graph segment, new partition assignment $I^{(s)}$, $J^{(s)}$

Compute new encoding c_n of $\mathcal{G}^{(s)} \bigcup \{G^{(t)}\}$

Compute encoding cost c for just $G^{(t)}$

`// check if there is any encoding benefit`

if $c_n - c_o < c$ **then**

  `// add` $G^{(t)}$ `in` $\mathcal{G}^{(s)}$

  $\mathcal{G}^{(s)} \leftarrow \mathcal{G}^{(s)} \bigcup \{G^{(t)}\}$

  SearchKL for updated $\mathcal{G}^{(s)}$

else

  `// start a new segment from` $G^{(t)}$

  $\mathcal{G}^{(s+1)} := \{G^{(t)}\}$

  SearchKL for new $\mathcal{G}^{(s+1)}$

Initialization

Once we decide to start a new segment, how should we initialize the number and membership of its partitions? There are several ways to do the initialization. Trading-off convergence speed versus compression quality, we propose and study two alternatives:

Fresh-start. One option is to start from a small k and ℓ, typically $k = 1$ and $\ell = 1$, and progressively increase them (see Algorithm 3.8) as well as re-group sources and destinations (see Algorithm 3.7). From our experiments, this scheme is very effective in leading to a good result. In terms of computational cost, it is relatively fast, since we start with small k and ℓ.

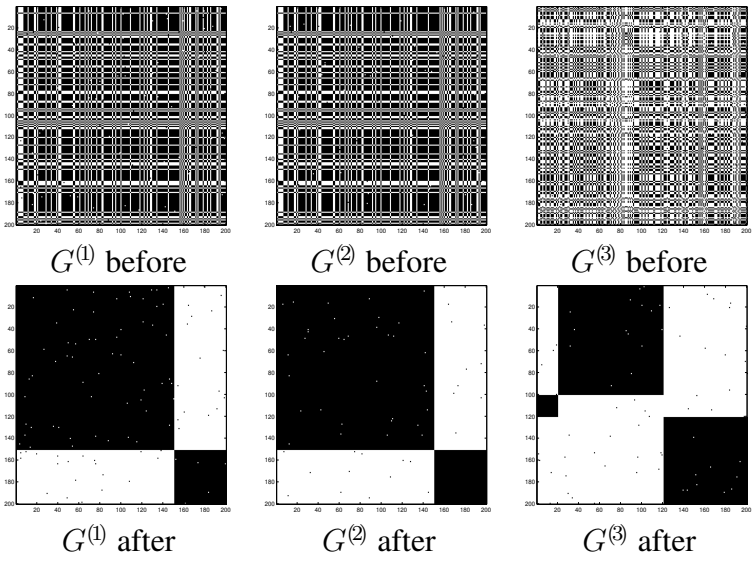


Figure 3.15: A graph stream with three graphs: The same communities appear in graph $G^{(1)}$ and $G^{(2)}$, therefore, they are grouped into the same graph segment. However, $G^{(3)}$ has different community structure, therefore, a new segment starts from $G^{(3)}$.

Resume. For time evolving graphs, consecutive graphs often have a strong similarity. We can leverage this similarity in the search process by starting from old partitions. More specifically, we initialize k_{s+1} and ℓ_{s+1} to k_s and ℓ_s, respectively. Additionally, we initialize $I^{(s+1)}$ and $J^{(s+1)}$ to $I^{(s)}$ and $J^{(s)}$. We study the relative CPU performance of *fresh-start* and *resume* in Section 3.3.4.

3.3.4 Experiments

In this section, we will evaluate the result on both *community discovery* and *change detection* of GraphScope using several real, large graph datasets. We first describe the datasets in Section 3.3.4. Then we present our experiments, which are designed to answer the following two questions:

- *Mining Quality:* How good is our method in terms of finding meaningful communities and change points (Section 3.3.4).
- *Speed:* How fast is it, and how does it scale up (Section 3.3.4).

Finally, we present some additional mining observations that our method automatically identifies. To the best of our knowledge, no other parameter-free and incremental method for time-evolving graphs has been proposed to date. Our goal is to automatically determine the best change-points in time, as well as the best node partitioning, which concisely reveal the basic structure of both communities as well as their change over time. It is not clear how parameters of other methods (e.g., number of partitions, graph similarity thresholds, etc) should be set for these methods to attain this goal. GraphScope is fully automatic and, as we will show, still able to find meaningful communities and change points.

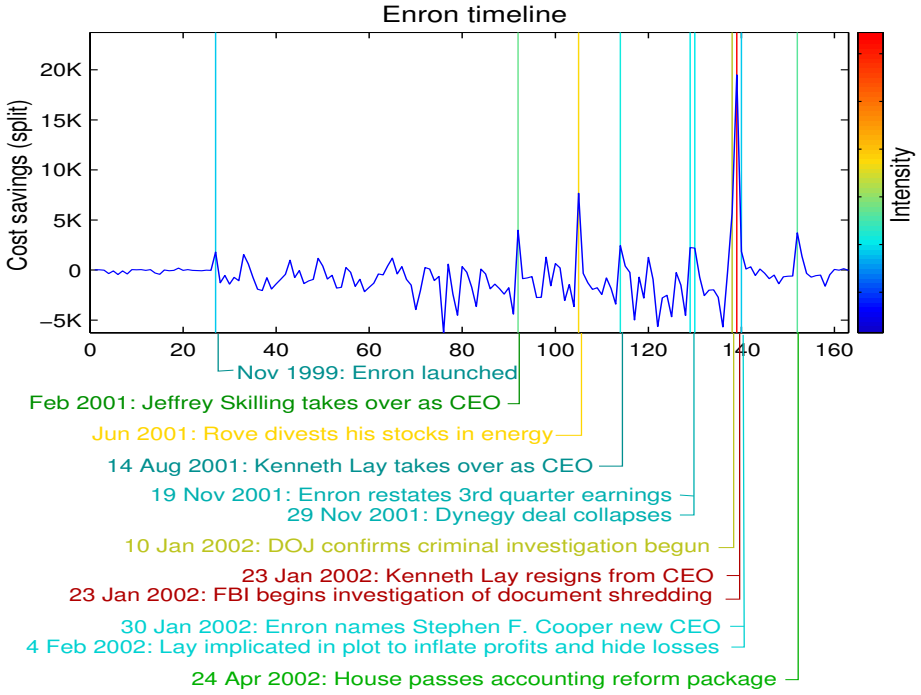


Figure 3.16: ENRON dataset (Best viewed in color). Relative compression cost versus time. Large cost indicates change points, which coincide with the key events. E.g., at time-tick 140 (Feb 2002), CEO Ken Lay was implicated in fraud.

Datasets

In this section, we describe the datasets in our experiments.

The NETWORK Dataset. The traffic trace consists of TCP flow records collected at the backbone router of a class-B university network. Each record in the trace corresponds to a

name	m**-by-**n	**avg.**$\lvert E \rvert$	**time T**
NETWORK	29K-by-29K	12K	$1,222$
ENRON	34k-by-34k	15K	165
CELLPHONE	97-by-3764	430	46
DEVICE	97-by-97	689	46
TRANSACTION	28-by-28	132	51

Table 3.5: Dataset summary

directional TCP flow between two hosts, with timestamps indicating when the flow started and finished. With this traffic trace, we use a window size of one hour to construct the source-destination graph stream. Each graph is represented by a sparse adjacency matrix with the rows and the columns corresponding to source and destination IP addresses, respectively. An edge in a graph $G^{(t)}$ means that there exist TCP flows (packets) sent from the i-th source to the j-th destination during the t-th hour. The graphs involve m=n=21,837 unique campus hosts (the number of source and destination nodes) with a per-timestamp average of over 12K distinct connections (the number of edges). The total number of timestamps T is 1,222. Figure 3.17(a) shows an example of superimposing[12] all source-destination graphs in one time segment of 18 hours. Every row/column corresponds to a source/destination; the dot there indicates there is at least a packet from the source to the destination during that time segment. The graphs are correlated, with most of the traffic to or from a small set of server-like hosts.

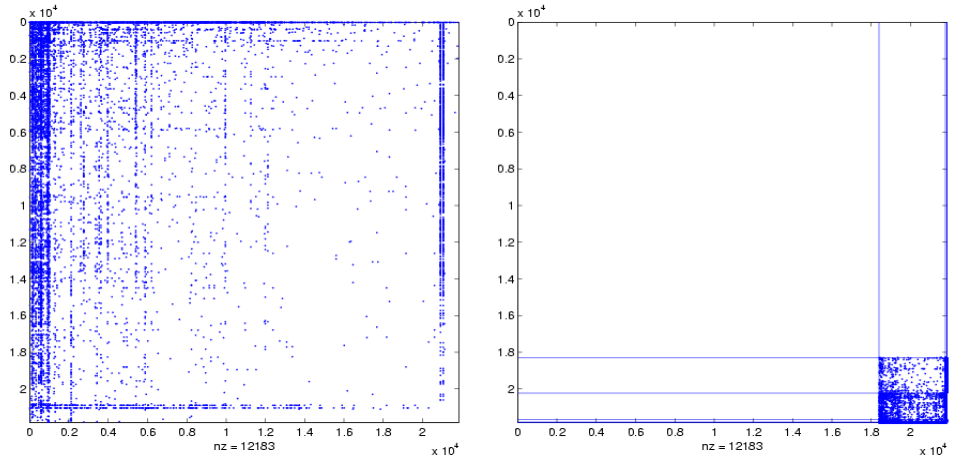
GraphScope automatically exploits the sparsity and correlation by organizing the sources and destinations into homogeneous groups as shown in Figure 3.17(b).

The ENRON **Dataset.** This consists of the email communications in Enron Inc. from Jan 1999 to July 2002 [2]. We construct sender-to-recipient graphs on a weekly basis. The graphs have $m = n = 34,275$ senders/recipients (the number of nodes) with an average of 1,479 distinct sender-receiver pairs (the number of edges) every week.

Like the NETWORK dataset, the graphs in ENRON are also correlated. GraphScope can reorganize the graph into homogeneous partitions (see the visual comparison in Figure 3.18).

The CELLPHONE **Dataset.** The CELLPHONE dataset records the cellphone activity for m=n=97 users from two different labs in MIT [1]. Each graph snapshot corresponds to a week, from Jan 2004 to May 2005. We thus have T=46 graphs, one for each week,

[12] Two graphs are superimposed together by taking the union of their edges.



(a) before (b) after

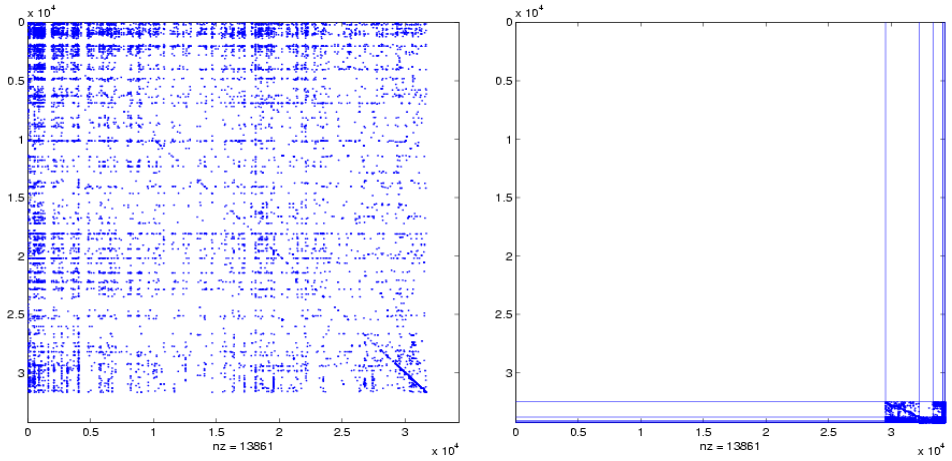
Figure 3.17: NETWORK before and after GraphScope for the graph segment between Jan 7 1:00, 2005 and Jan 7 19:00, 2005. GraphScope successfully rearrange the sources and destinations such that the sub-matrices are much more homogeneous.

excluding weeks with no activity.

We plot the superimposed graphs of weeks 38 to 42 in 2004 at Figure 3.19(a), which looks much more random than NETWORK and ENRON. However, GraphScope is still able to extract the hidden structure from the graph as shown in Figure 3.19(b), which looks much more homogeneous (more details in Section 3.3.4).

The DEVICE dataset. DEVICE dataset is constructed on the same 97 users whose cellphones periodically scan for nearby phones and computers over Bluetooth. The goal is to understand people's behavior from their proximity to others. Figure 3.20(a) plots the superimposed user-to-user graphs for one time segment where every dot indicates that the two corresponding users are physically near each other. Note that the first row represents all the devices that do not belong to any of the 97 individual users (mainly laptop computers, PDAs and other peoples' cellphones). Figure 3.20(b) shows the resulting user partitions for that time segment, where cluster structure is revealed (see Section 3.3.4 for details).

The Transaction Dataset. The TRANSACTION dataset has m=n=28 accounts of a company, over 2,200 days. An edge indicates that the source account had funds transfered to the destination account. Each graph snapshot covers transaction activities over a



(a) before (b) after

Figure 3.18: ENRON before and after GraphScope for the graph segment of week 35, 2001 to week 38, 2001. GraphScope can achieve significant compression by partitioning senders and recipients into homogeneous groups

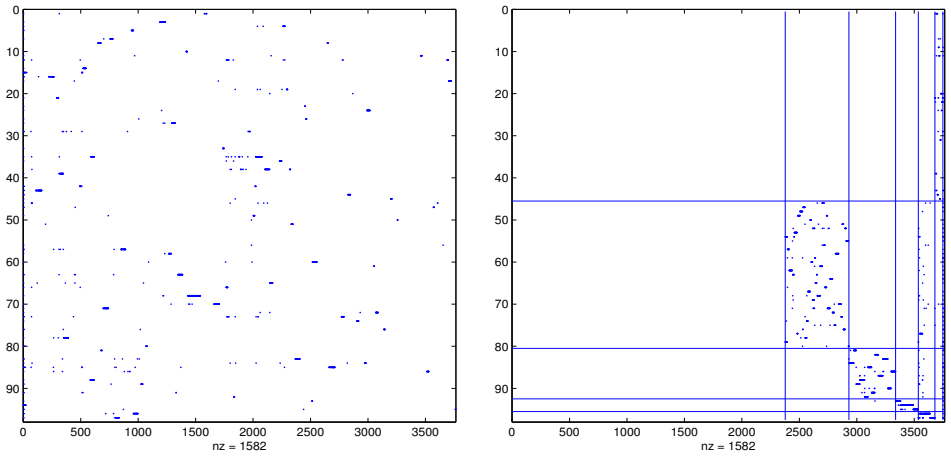
window of 40 days, resulting in T=51 time-ticks for our dataset.

Figure 3.26(a) shows the transaction graph for one timestamp. Every black square at the (i, j) entry in Figure 3.26(a) indicates there is at least one transaction debiting the ith account and crediting the jth account. After applying GraphScope on that timestamp (see Figure 3.26(b)), the accounts are organized into very homogeneous groups with a few exceptions.

Mining Case-studies

Now we qualitatively present the mining observation on all the datasets. More specifically, we illustrate that (1) source and destination groups correspond to semantically meaningful clusters; (2) the groups evolve over time; (3) time segments indicate interesting change-points.

NETWORK**: Interpretable groups.** Despite the bursty nature of network traffic, GraphScope can successfully cluster the source and destination hosts into meaningful groups. Figure 3.21(a) and (b) show the active source and destination nodes organized by groups for two different time segments. Note that Figure 3.21 is in log-log scale to visualize those small partitions. For example, source nodes are grouped into (1) active hosts which talk to



(a) before (b) after

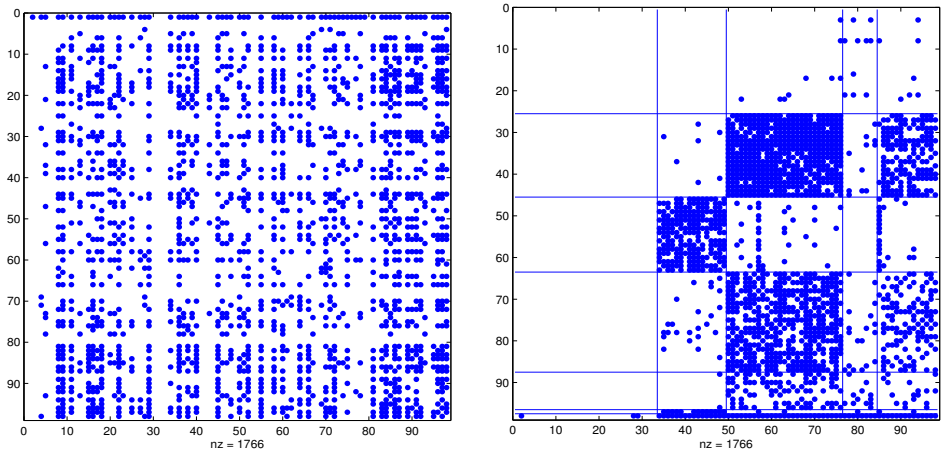
Figure 3.19: CELLPHONE before and after GraphScope, for the period of week 38 to 42 in 2004

a small number of hosts, (2) P2P hosts that scan a number of hosts, and (3) administrative scanning hosts[13] which scan many hosts. Similarly, destination hosts are grouped into (1) active hosts, (2) cluster servers at which many students login remotely to work on different tasks, (3) web servers which hosts the websites of different schools, and (4) mail servers that have the most incoming connections. The main difference between Figure 3.21(a) and (b) is that a source group of unusual scanners emerges in the latter. GraphScope can automatically identify the change and decide to split into two time segments.

CELLPHONE**: Evolving groups.** As in NETWORK, we also observe meaningful groups in CELLPHONE. Figure 3.22 (a) illustrate the calling patterns in the fall semester of 2004, where two strong user partitions (G1 and G2) exist. The dense small partition G3 is the service call in campus, which has a lot of incoming calls from everyone. Figure 3.22 (b) illustrates that the calling patterns changed during the winter break which follows the fall semester.

DEVICE**: Evolving groups.** The evolving group behavior is also observed in the DEVICE dataset. In particular, two dense partitions appear in Figure 3.23(a): after inspecting the user ids and their attributes, we found that the users in group $U1$ are all from the same school with similar schedule, probably taking the same class; the users in $U2$ all work in

[13]The campus network is constantly running some port-scanning program to identify potential vulnerabilities of the in-network hosts.



(a) before (b) after

Figure 3.20: DEVICE before and after GraphScope for the time segment between week 38, 2004 and week 42, 2004. Interesting communities are identified

the same lab. In a later time segment (see Figure 3.23(b)), the partition $U1$ disappeared, while the partition $U2$ is unchanged.

TRANSACTION**.** As shown in Figure 3.26(b), GraphScope successfully organizes the 28 accounts into three partitions. Upon closer inspection, these groups correspond to the different functional groups of the accounts (e.g., 'marketing', 'sales')[14]. In Figure 3.26(b), the interaction between first source partition (from the top) and second destination partition (from the left) correspond to mainly the transactions from assets accounts to liability and revenue accounts, which obeys common business practice.

ENRON**: Change-point detection.** The source and destination partitions usually correspond to meaningful clusters for the given time segment. Moreover, the time segments themselves usually encode important information about changes. Figure 3.16 plots the encoding cost difference between incorporating the new graph into the current time segment vs. starting a new segment. The vertical lines on Figure 3.16 are the top 10 splits with largest cost savings when starting a new segment, which actually correspond to the key events related to Enron Inc. Moreover, the intensity in terms of magnitude and frequency dramatically increases around Jan 2002 which coincides with several key incidents such as the investigation on document shredding, and the CEO resignation.

[14] Due to anonymity requirements, the account types are obfuscated.

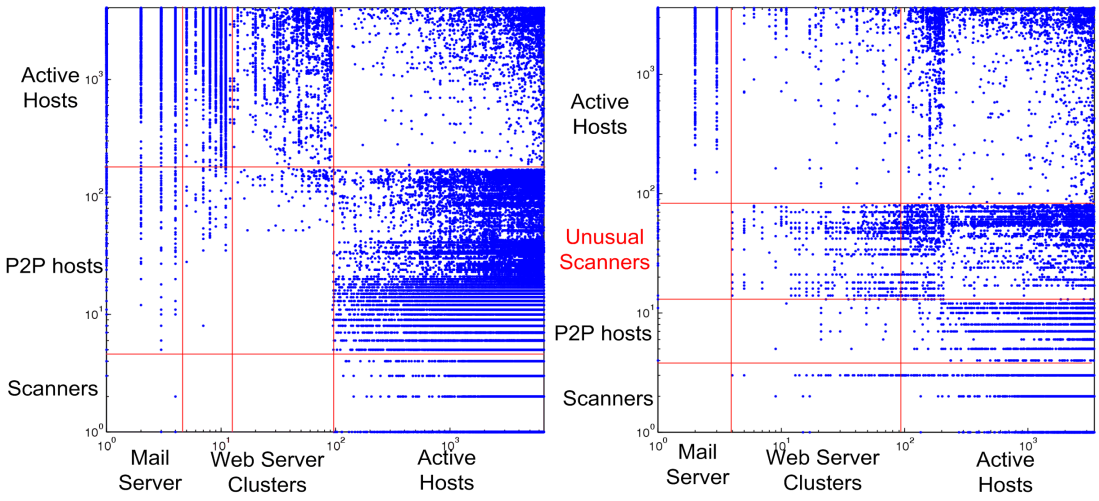


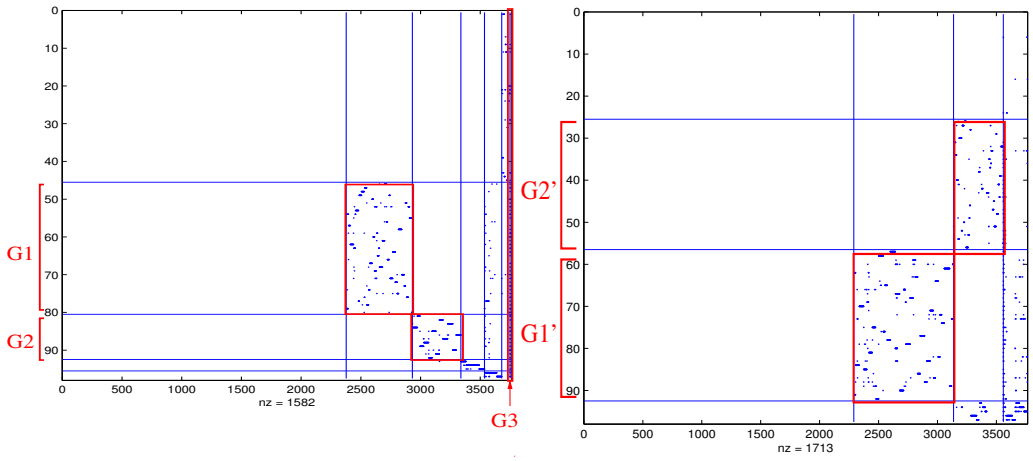
Figure 3.21: NETWORK zoom-in (log-log plot): (a) Source nodes are grouped into active hosts and security scanning program; Destination nodes are grouped into active hosts, clusters, web servers and mail servers. (b) on a different time segment, a group of unusual scanners appears, in addition to the earlier groups.

Quality and Scalability

We compare *fresh-start* and *resume* (see Section 3.3.3) in terms of compression benefit, against the global compression estimate and the space requirement for the original graphs, stored as sparse matrices (adjacency list representation). Figure 3.24 shows that both *fresh-start* and *resume* GraphScope achieve high compression gain (less than 4% of the original space), which is even better than the global compression on the graphs (the 3rd bar for each dataset). Our two variations require about the same space.

Now we illustrate the CPU cost (scalability) of *fresh-start* and *resume*. As shown in Figure 3.25(a) for NETWORK (similar result are achieved for the other datasets, hence omitted), the CPU cost per timestamp/graph is stable over time for both *fresh-start* and *resume*, which suggests that both proposed methods are scalable to streaming environments.

Furthermore, *resume* is much faster than *fresh-start* as shown in Figure 3.25(b), especially for large graphs such as in NETWORK. There, *resume* only uses 10% of CPU time compared to *fresh-start*.



(a) fall semester (b) winter break

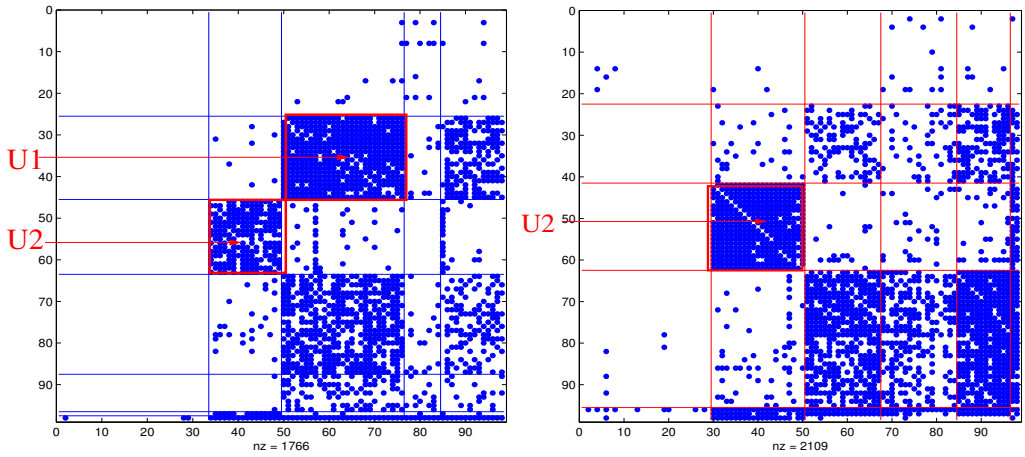
Figure 3.22: CELLPHONE: a) Two calling groups appear during the fall semester; b) Call groups changed in the winter break. The change point corresponds to the winter break.

3.3.5 Summary

We propose GraphScope, a parameter-free scheme to mine streams of graphs. Our method has all of the following desired properties: 1) It is rigorous and automatic, with no need for user-defined parameters. Instead, it uses the Minimum Description Language (MDL) principle, to decide how to form communities, and when to modify them. 2) It is fast and scalable, carefully designed to work in a streaming setting. 3) It is effective, discovering meaningful communities and meaningful transition points.

We also present experiments on several real datasets, spanning 500 Gigabytes. The datasets were from widely diverse applications (university network traffic, email from the Enron company, cellphone call logs and Bluetooth connections). Because of its generality and its information theoretic underpinnings, GraphScope is able to find meaningful groups and patterns in all the above settings, without any specific fine-tuning on our side.

Future research directions include extensions to create hierarchical groupings, both of the communities, as well as of the time segments.



(a) two groups (b) one group disappeared

Figure 3.23: DEVICE: (a) two groups are prominent. Users in $U1$ are all from the same school with similar schedule possibly taking the same class; Users in $U2$ are all working in the same lab. (b) $U1$ disappears in the next time segment, while $U2$ remains unchanged.

3.4 Chapter summary: graph mining

Time-evolving graphs or second-order tensor streams have a variety of applications in social network analysis, Internet monitoring, financial auditing and computational biology. The key questions are the following: How to summarize time-evolving graphs? How to find communities and track them over time? How to identify anomalies in the graphs? The challenges are the huge size, sparse property and dynamic nature in the graphs.

We introduced two scalable techniques that exploit two different aspects of the real graphs. First, we presented Compact Matrix Decomposition (CMD) which summarizes the graphs as low-rank approximations similar to SVD but preserving the sparsity in the process. In particular, CMD achieves orders of magnitudes improvement on both speed and space compared to the state of arts. Also we illustrated the success of CMD on the anomaly detection in network forensic application. Second, we developed GraphScope, an online algorithm that incrementally clusters the time-evolving graphs in a parameter-free fashion. GraphScope can track the community structure over time and identify the change points. We applied GraphScope successfully on several real, large datasets including Enron email communication graphs. In particular, the change points on the Enron dataset coincide with many real events such as FBI investigation on the document shredding.

Next, we will present general techniques to handle higher-order tensor streams.

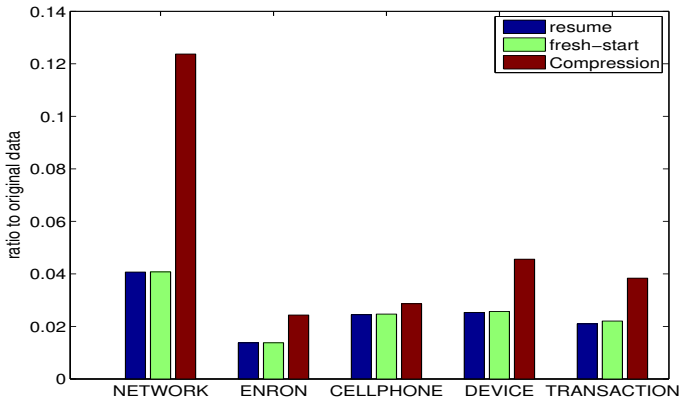
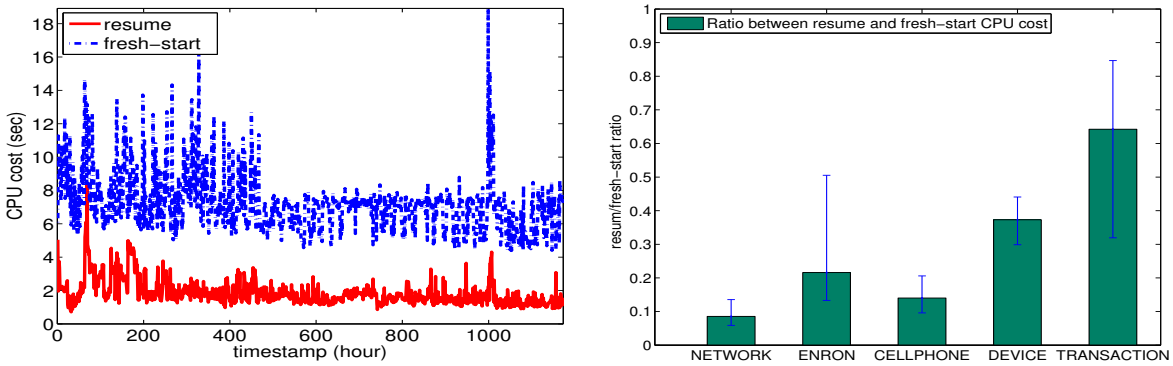


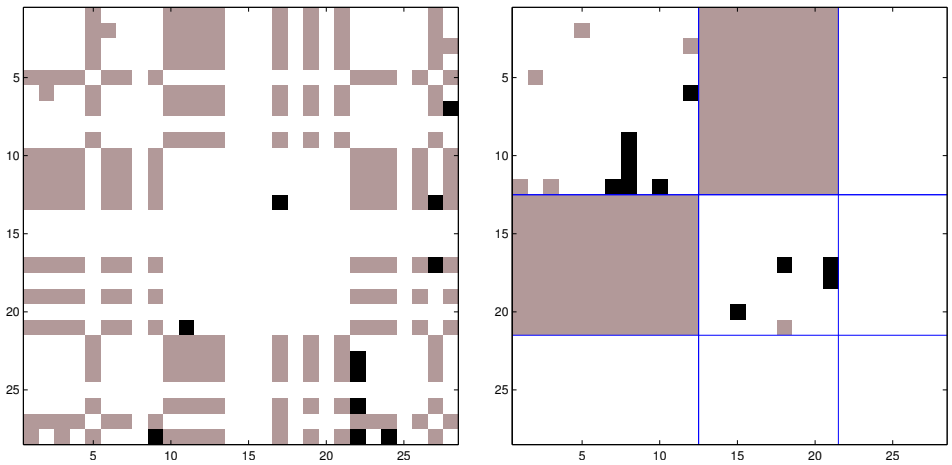
Figure 3.24: Relative Encoding Cost: Both *resume* and *fresh-start* methods give over an order of magnitude space saving compared to the raw data and are much better than global compression on the raw data.



(a) CPU cost (NETWORK)

(b) Relative CPU

Figure 3.25: CPU cost: (a) the CPU costs for both *resume* and *fresh-start* GraphScope are stable over time; (b) *resume* GraphScope is much faster than *fresh-start* GraphScope on the same datasets (the error bars give 25% and 75% quantiles);



(a) before (b) after

Figure 3.26: TRANSACTION before and after GraphScope for a time segment of 5 months. GraphScope is able to group accounts into partitions based on their types. Darker color indicates multiple edges over time.

Chapter 4

Tensor Mining

"How to summarize high-order data cubes (tensor)? How to incrementally update those patterns over time?"

Given a keyword-author-timestamp-conference bibliography, how can we find patterns and latent concepts? Given Internet traffic data (who sends packets to whom, on what port, and when), how can we find anomalies, patterns and summaries? Anomalies could be, e.g., port-scanners, patterns could be of the form "workstations are down on weekends, while servers spike at Fridays for backups". Summaries like the one above are useful to give us an idea what is the past (which is probably the norm), so that we can spot deviations from it in the future.

Powerful as they may be, matrix-based tools such as PCA and SVD can handle neither of the two problems we stated in the beginning. The crux is that matrices have only two "dimensions" (e.g., "customers" and "products"), while we may often need more, like "authors", "keywords", "timestamps", "conferences". This is exactly what a *tensor* is, and of course, a tensor is a generalization of a matrix (and of a vector, and of a scalar). We propose to envision all such problems as tensor problems, to use the vast literature of tensors to our benefit, and to introduce new tensor analysis tools, tailored for streaming applications.

Using tensors, we can attack an even wider range of problems, that matrices can not even touch. For example, 1) Rich, time-evolving network traffic data, as mentioned earlier: we have tensors of order $M = 3$, with modes "source-IP", "destination-IP" and "port" over time. 2) Labeled graphs and social networks: suppose that we have different types of edges in a social network (eg., who-called-whom, who-likes-whom, who-emailed-whom, who-borrowed-money-from-whom). In that case, we have a 3rd order tensor, with edge-type

being the 3rd mode. Over time, we have multiple 3rd order tensors, which are still within the reach of our upcoming tools. 3) Microarray data, with gene expressions evolving over time [139]. Here we have genes-proteins over time, and we record the expression level: a series of 2nd order tensors. 4) All OLAP and DataCube applications: customer-product-branch sales data is a 3rd order tensor; and so is patient-diagnoses-drug treatment data.

Motivating example:. Let us consider the network monitoring example as shown in Figure 1.3. Here we have network flows arriving very fast and continuously through routers, where each flow consists of source IP, destination IP, port number and the number of packets. How to monitor the dynamic traffic behavior? How to identify anomalies which can signify a potential intrusion, or worm propagation, or an attack? What are the correlations across the various sources, destinations and ports?

Therefore, from the **data model aspect** in this chapter, we focus on a more general and expressive model *tensor stream*. For the network flow example, the 3rd order tensor for a given time period has three modes: source, destination and port, which can be viewed as a 3D data cube (see Figure 1.3(a)). An entry (i, j, k) in that tensor (like the small blue cube in Figure 1.3(a)) has the number of packets from the corresponding source (i) to the destination j through port k, during the given time period. The dynamic aspect comes from the fact that new tensors are arriving continuously over time.

In this chapter, we present a general framework, Incremental Tensor Analysis (ITA) for summarizing tensor streams. ITA incrementally summarizes input tensors in the tensor stream as core tensors associated with projection matrices, one for each mode as shown in Figure 1.3(b). The core tensor can be viewed as a low dimensional summary of an input tensor; The projection matrices specify the transformation between the input tensor and the core tensor. The projection matrices can be updated incrementally over time when a new tensor arrives, and contain valuable information about which, eg., source IP addresses are correlated with which destination IP addresses and destination ports, over time.

From the **algorithmic aspect**, we introduce three different variants of ITA:

- *Dynamic Tensor Analysis (DTA)* incrementally maintains covariance matrices for all modes and uses the leading eigen-vectors of covariance matrices as projection matrices.

- *Streaming Tensor Analysis (STA)* directly updates the leading eigen-vectors of covariance matrices using SPIRIT algorithm.

- *Window-based Tensor Analysis (WTA)* uses similar updates as DTA but perform alternating iteration to further improve the results.

Finally, from the **application aspect**, we illustrate the anomaly detection and *multi-way LSI* through DTA/STA and WTA. For anomaly detection, our method found suspicious Internet activity in a real trace, and it also found the vast majority of injected anomalies (see Section 4.3). For multi-way LSI, we found natural groups in a DBLP bibliography dataset and in an environmental monitoring dataset.

4.1 Tensor background and related work

As mentioned, a tensor of order M closely resembles a Data Cube with M dimensions. Formally, we write an Mth order tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_M}$ as $\mathcal{X}_{[n_1,\ldots,n_M]}$, where n_i ($1 \leq i \leq M$) is the *dimensionality* of the ith mode ("dimension" in OLAP terminology). For brevity, we often omit the subscript $[n_1, \ldots, n_M]$.

We will also follow the typical conventions, and denote matrices with upper case bold letters (e.g., $\mathbf{U}$) column vectors with lower-case bold letters (e.g., $\mathbf{x}$, scalars with lower-case normal font (e.g., n), and tensors with calligraphic font (e.g., $\mathcal{X}$). From the tensor literature we need the following definitions.

4.1.1 Matrix Operators

The Kronecker product (or the tensor product), Khatri-Rao product and Hadamard product are important matrix operators related to tensors.

Definition 4.1 (Kronecker Product)**.** *The Kronecker product of matrices* $\mathbf{A} \in \mathbb{R}^{m \times n}$ *and* $\mathbf{B} \in \mathbb{R}^{p \times q}$ *is denoted as matrix* $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{mp \times nq}$*:*

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}$$

The Kronecker product $\mathbf{A} \otimes \mathbf{B}$ creates many copies of matrix $\mathbf{B}$ and scales each one by the corresponding entry of $\mathbf{A}$. Kronecker product has the following properties (see [97] for more complete discussion).

Property 4.1 (Kronecker Product)**.** *Let* $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{B} \in \mathbb{R}^{p \times q}$, $\mathbf{C} \in \mathbb{R}^{n \times o}$ *and* $\mathbf{D} \in \mathbb{R}^{q \times r}$

then

$$(\mathbf{A} \otimes \mathbf{B})^{\mathsf{T}} = \mathbf{B}^{\mathsf{T}} \otimes \mathbf{A}^{\mathsf{T}}$$
$$(\mathbf{A} \otimes \mathbf{B})^{\dagger} = \mathbf{B}^{\dagger} \otimes \mathbf{A}^{\dagger}$$
$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}$$
$$(\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} = \mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C})$$

where $\mathbf{A}^{\mathsf{T}}$ *and* $\mathbf{A}^{\dagger}$ *are the transpose and pseudo-inverse of matrix* $\mathbf{A}$*, respectively.*

Definition 4.2 (Khatri-Rao Product)**.** *The Khatri-Rao product of matrices* $\mathbf{A} \in \mathbb{R}^{m \times n}$ *and* $\mathbf{B} \in \mathbb{R}^{p \times n}$ *is denoted as* $\mathbf{A} \odot \mathbf{B} \in \mathbb{R}^{mp \times n}$*:*

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b}_1 & \mathbf{a}_2 \otimes \mathbf{b}_2 & \cdots & \mathbf{a}_n \otimes \mathbf{b}_n \end{bmatrix}$$

The Khatri-Rao product requires the two matrices with the same number of columns. The Khatri-Rao product can be viewed as a column-wise Kronecker product. Note that the Khatri-Rao product of two vectors is identical to the Kronecker product of two vectors.

Definition 4.3 (Hadamard Product)**.** *The Hadamard product of matrices* $\mathbf{A} \in \mathbb{R}^{m \times n}$ *and* $\mathbf{B} \in \mathbb{R}^{m \times n}$ *is denoted as* $\mathbf{A} * \mathbf{B} \in \mathbb{R}^{n \times m}$*:*

$$\mathbf{A} * \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \cdots & a_{mn}b_{mn} \end{bmatrix}$$

Hadamard product performs element-wise multiplication between two matrices of the same size.

Property 4.2 (Khatri-Rao Product)**.** *Let* $\mathbf{A} \in \mathbb{R}^{m \times n}$*,* $\mathbf{B} \in \mathbb{R}^{p \times n}$ *and* $\mathbf{C} \in \mathbb{R}^{q \times n}$ *then*

1. $\mathbf{A} \odot \mathbf{B} \odot \mathbf{C} = (\mathbf{A} \odot \mathbf{B}) \odot \mathbf{C} = \mathbf{A} \odot (\mathbf{B} \odot \mathbf{C})$
2. $(\mathbf{A} \odot \mathbf{B})^{\mathsf{T}}(\mathbf{A} \odot \mathbf{B}) = (\mathbf{A}^{\mathsf{T}}\mathbf{A}) * (\mathbf{B}^{\mathsf{T}}\mathbf{B})$
3. $(\mathbf{A} \odot \mathbf{B})^{\dagger} = ((\mathbf{A}^{\mathsf{T}}\mathbf{A}) * (\mathbf{B}^{\mathsf{T}}\mathbf{B}))^{\dagger}(\mathbf{A} \odot \mathbf{B})^{\mathsf{T}}$

The properties of Khatri-Rao product are important for reducing the computation cost of tensor decomposition, especially PARAFAC.

Definition 4.4 (Outer product)**.** *The outer product of* M *vectors* $\mathbf{a}^{(i)}|_{i=1}^{M} \in \mathbb{R}^{n_i}$ *is denoted as* $\mathbf{a}^{(1)} \circ \mathbf{a}^{(2)} \circ \cdots \circ \mathbf{a}^{(M)}$*:*

$(\mathbf{a}^{(1)} \circ \mathbf{a}^{(2)} \circ \cdots \circ \mathbf{a}^{(M)})_{i_1 i_2 \ldots i_M} = a^{(1)}(i_1)a^{(2)}(i_2) \cdots a^{(M)}(i_M)$ *for* $1 \leq i_j \leq n_j, 1 \leq j \leq M$

4.1.2 Tensor Operators

Definition 4.5 (Mode Product)**.** *The d-mode product of a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$ with a matrix $\mathbf{A} \in \mathbb{R}^{r \times n_d}$ is denoted as $\mathcal{X} \times_d \mathbf{A}$ which is defined element-wise as*

$$(\mathcal{X} \times_d \mathbf{A})_{i_1 \ldots i_{d-1} j i_{d+1} \ldots i_M} = \sum_{i_d=1}^{n_d} x_{i_1 \times i_2 \times \cdots \times i_M} a_{j i_d}$$

Figure 4.1 shows an example of 3rd order tensor $\mathcal{X}$ mode-1 multiplies a matrix $\mathbf{U}$. The process is equivalent to first matricizing $\mathcal{X}$ along mode-1, then to doing matrix multiplication of $\mathbf{U}$ and $\mathbf{X}_{(1)}$, finally folding the result back as a tensor.

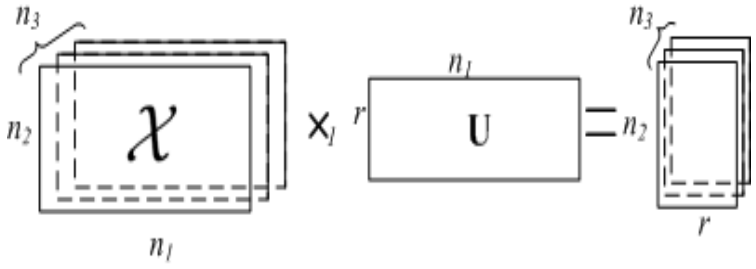


Figure 4.1: 3rd order tensor $\mathcal{X}_{[n_1,n_2,n_3]} \times_1 \mathbf{U}$ results in a new tensor in $\mathbb{R}^{r \times n_2 \times n_3}$

In general, a tensor $\mathcal{Y} \in \mathbb{R}^{r_1 \times \cdots \times r_M}$ can multiply a sequence of matrices $\mathbf{U}^{(i)}|_{i=1}^{M} \in \mathbb{R}^{n_i \times R_i}$ as: $\mathcal{Y} \times_1 \mathbf{U}_1 \cdots \times_M \mathbf{U}_M \in \mathbb{R}^{n_1 \times \cdots \times n_M}$,which can be written as $\mathcal{Y} \prod_{i=1}^{M} \times_i \mathbf{U}_i$ for clarity. Furthermore, the notation for $\mathcal{Y} \times_1 \mathbf{U}_1 \cdots \times_{i-1} \mathbf{U}_{i-1} \times_{i+1} \mathbf{U}_{i+1} \cdots \times_M \mathbf{U}_M$ (i.e. multiplication of all $\mathbf{U}_j$s except the i-th) is simplified as $\mathcal{Y} \prod_{j \neq i} \times_j \mathbf{U}_j$.

Property 4.3 (Mode Product)**.** *Let $\mathcal{Y} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$, then*

1. *Given $\mathbf{A} \in \mathbb{R}^{p \times n_i}$ and $\mathbf{B} \in \mathbb{R}^{q \times n_j}$, $\mathcal{Y} \times_i \mathbf{A} \times_j \mathbf{B} = (\mathcal{Y} \times_i \mathbf{A}) \times_j \mathbf{B} = (\mathcal{Y} \times_j \mathbf{B}) \times_i \mathbf{A}$*
2. *Given $\mathbf{A} \in \mathbb{R}^{p \times n_i}$ and $\mathbf{B} \in \mathbb{R}^{q \times p}$, then $\mathcal{Y} \times_i \mathbf{A} \times_i \mathbf{B} = \mathcal{Y} \times_i (\mathbf{B}\mathbf{A})$*
3. *Given $\mathbf{A} \in \mathbb{R}^{p \times n_i}$ with full column rank, then $\mathcal{X} = \mathcal{Y} \times_i \mathbf{A} \Rightarrow \mathcal{Y} = \mathcal{X} \times_i \mathbf{A}^{\dagger}$*
4. *A special case of above, if $\mathbf{A} \in \mathbb{R}^{p \times n_i}$ is orthonormal with full column rank, then $\mathcal{X} = \mathcal{Y} \times_i \mathbf{A} \Rightarrow \mathcal{Y} = \mathcal{X} \times_i \mathbf{A}^{\mathsf{T}}$*

Definition 4.6 (Mode-n Matricization)**.** *The mode-d matricization or matrix unfolding of an Mth order tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_M}$ are vectors in $\mathbb{R}^{n_d}$ obtained by keeping index d fixed and varying the other indices. Therefore, the mode-d matricization $\mathbf{X}_{(d)}$ is in $\mathbb{R}^{(n_d \times \prod_{i \neq d} n_i)}$.*

The mode-d matricization $\mathcal{X}$ is denoted as unfold($\mathcal{X}$,d)= $\mathbf{X}_{(d)}$. Similarly, the inverse operation is denoted as fold($\mathbf{X}_{(d)}$). In particular, we have $\mathcal{X} = \text{fold}(\text{unfold}(\mathcal{X}, d))$. Figure 4.2 shows an example of mode-1 matricizing of a 3rd order tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ to the $(n_2 \times n_3) \times n_1$-matrix $\mathbf{X}_{(1)}$. Note that the shaded area of $\mathbf{X}_{(1)}$ in Figure 4.2 the slice of the 3rd mode along the 2nd dimension. A more general case of matricization forms a matrix by partitioning all the modes into row and column groups [86].

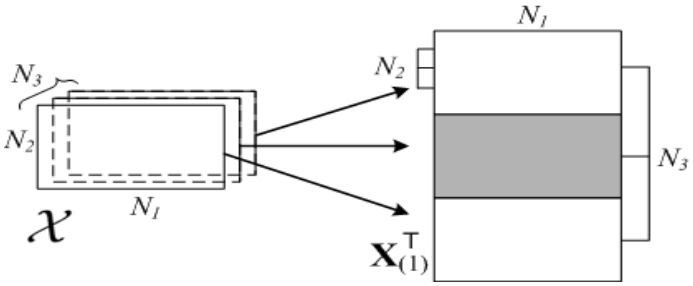


Figure 4.2: 3rd order tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ is matricized along mode-1 to a matrix $\mathbf{X}_{(1)} \in \mathbb{R}^{(n_2 \times n_3) \times n_1}$. The shaded area is the slice of the 3rd mode along the 2nd dimension.

Property 4.4 (Matricization)**.** *Given $\mathcal{Y} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$*

1. *If $\mathbf{A} \in \mathbb{R}^{k \times n_d}$, then $\mathcal{X} = \mathcal{Y} \times_d \mathbf{A} \Rightarrow \mathbf{X}_{(d)} = \mathbf{A}\mathbf{Y}_{(d)}$*
2. *If $\mathbf{A}^{(d)} \in \mathbb{R}^{k_d \times n_d}|_{d=1}^{M}$, then $\mathcal{X} = \mathcal{Y} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_M \mathbf{A}^{(M)} \Leftrightarrow \mathbf{X}_{(d)} = \mathbf{A}^{(d)}\mathbf{Y}_{(d)}(\mathbf{A}^{(M)} \otimes \cdots \otimes \mathbf{A}^{(n+1)}\mathbf{A}^{(n-1)} \otimes \cdots \otimes \mathbf{A}^{(1)})^{\mathsf{T}}$*

Definition 4.7 (Inner Product)**.** *The inner product of two tensors $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_M}$ is denoted as $\langle \mathcal{X}, \mathcal{Y} \rangle$:*

$$\langle \mathcal{X}, \mathcal{Y} \rangle = \text{vec}(\mathcal{X})^{\mathsf{T}}\text{vec}(\mathcal{Y})$$

where $\text{vec}(\mathcal{X})$ is the vectorized version of $\mathcal{X}$.

Therefore, the norm of a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_M}$ is defined as

$$\| \mathcal{X} \|^2 = \langle \mathcal{X}, \mathcal{X} \rangle$$

Property 4.5 (Inner Product)**.** *Assume all the sizes match,*

1. $\langle \mathcal{X}, \mathcal{Y} \times_d \mathbf{A} \rangle = \langle \mathcal{X} \times_d \mathbf{A}^\mathsf{T}, \mathcal{Y} \rangle$
2. *If* $\mathbf{Q}$ *is an orthonormal matrix, then* $\| \mathcal{X} \| = \| \mathcal{X} \times_d \mathbf{Q} \|$.

4.1.3 Tensor Decomposition

We introduce two popular tensor decompositions, namely the Tucker decomposition, and the PARAllel FACtor Analysis (PARAFAC).

Tucker Decomposition

The main objectives of Tucker decomposition [125] is to approximate a large tensor using a small tensor through a change of basis.

Definition 4.8 (Tucker Decomposition)**.** *Given an input tensor* $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$ *and core tensor sizes* $\{r_1, \ldots, r_M\}$*, find a core tensor* $\mathcal{Y} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_M}$ *and a sequence of projection matrices* $\mathbf{U}^{(d)}|_{d=1}^{M} \in \mathbb{R}^{n_d \times r_d}$ *such that* $\left\| \mathcal{X} - \mathcal{Y} \times_1 \mathbf{U}^{(1)} \times \cdots \times_M \mathbf{U}^{(M)} \right\|$ *is small, i.e.,* $\mathcal{X} \approx \mathcal{Y} \times_1 \mathbf{U}^{(1)} \cdots \times_M \mathbf{U}^{(M)}$.

The approximation can be exact if $r_d = n_d$, $\mathbf{U}^{(d)}$ is full rank for $1 \leq d \leq M$. Essentially, Tucker decomposition changes the bases for every mode through the projection matrix $\mathbf{U}^{(d)}|_{d=1}^{M}$.

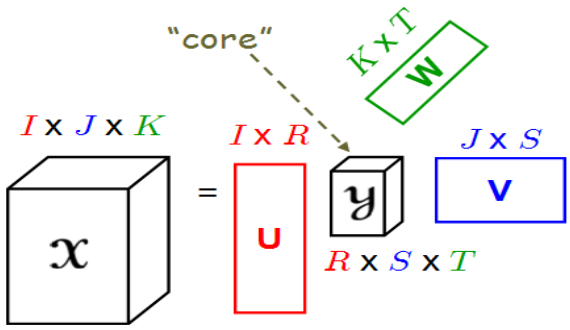


Figure 4.3: 3rd order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K} \approx \mathcal{Y} \times_1 \mathbf{U} \times_2 \mathbf{V} \times_3 \mathbf{W}$ where $\mathcal{Y} \in \mathbb{R}^{R \times S \times T}$ is the core tensor, $\mathbf{U}, \mathbf{V}, \mathbf{W}$ the projection matrices.

Algorithm 4.1: HOSVD(tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$, core sizes $r_d|_{d=1}^{M}$)

Output: projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M}$ and core tensor $\mathcal{Y} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_M}$

for $d = 1$ *to* M **do**

    // Find the covariance matrix of mode-d matricization

    $\mathbf{C} = \mathbf{X}_{(d)}\mathbf{X}_{(d)}^{\mathsf{T}}$

    Set $\mathbf{U}^{(d)}$ to be the r_d leading left eigenvectors of $\mathbf{C}$

$\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}^{(1)\mathsf{T}} \cdots \times_M \mathbf{U}^{(M)\mathsf{T}}$ // Compute the core tensor

Property 4.6 (Tucker Decomposition)**.** *The following presentations are equivalent:*

1. Mode-n multiplication*:* $\mathcal{X} \approx \mathcal{Y} \times_1 \mathbf{U}^{(1)} \cdots \times_M \mathbf{U}^{(M)}$
2. Matricization*:* $\mathbf{X}_{(d)} \approx \mathbf{U}^{(d)}\mathbf{Y}_{(d)}(\mathbf{U}^{(M)} \otimes \cdots \otimes \mathbf{U}^{(d+1)} \otimes \mathbf{U}^{(d-1)} \otimes \cdots \otimes \mathbf{U}^{(1)})^{\mathsf{T}}$
3. Outer product*:* $\mathcal{X} \approx \sum_{i_1=1}^{n_1}\sum_{i_2=1}^{n_2}\cdots\sum_{i_M=1}^{n_M} y_{i_1,i_2,\ldots,i_M}\mathbf{u}_{i_1}^{(1)} \circ \mathbf{u}_{i_2}^{(2)} \circ \cdots \circ \mathbf{u}_{i_M}^{(M)}$

In general, the Tucker decomposition is not unique. In particular, any r-by-r orthonormal transformation can be included without affecting the fit. e.g., $\mathcal{Y} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} = (\mathcal{Y} \times \mathbf{B}) \times_1 \mathbf{U}^{(1)}\mathbf{B}^{\mathsf{T}} \times_2 \mathbf{U}^{(2)}$

High-Order SVD (HOSVD) and High-Order Orthogonal Iteration (HOOI) are two popular algorithms to find the Tucker decomposition. HOSVD treats all the modes independently and performs matrix SVD on every matricization of the tensor; while HOOI performs alternating optimization to find better projection matrices iteratively. In general, HOSVD can be considered as a special case of HOOI with one iteration only.

PARAFAC

A special case of the Tucker decomposition is the case where the core tensor is superdiagonal, also known as PARAFAC.

Definition 4.9 (PARAFAC)**.** *Given an input tensor* $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$ *and the core size* r*, find* r *rank-one tensors in the form of* $\lambda_i \mathbf{u}_i^{(1)} \circ \cdots \circ \mathbf{u}_i^{(M)}$ *where* $\mathbf{u}_i^{(d)}|_{d=1}^{M} \in \mathbb{R}^{n_d}$ *for* $1 \le i \le r$ *such that* $\left\| \mathcal{X} - \sum_{i=1}^{r} \lambda_i \mathbf{u}_i^{(1)} \circ \cdots \circ \mathbf{u}_i^{(M)} \right\|$ *is small, i.e.,* $\mathcal{X} \approx \sum_{i=1}^{r} \lambda_i \mathbf{u}_i^{(1)} \circ \cdots \circ \mathbf{u}_i^{(M)}$

Note that the projection matrix $\mathbf{U}^{(i)} = [\mathbf{u}_1^{(i)}, \ldots, \mathbf{u}_r^{(i)}] \in \mathbb{R}^{n_i \times r}$ is in general not orthonormal.

Algorithm 4.2: HOOI(tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$, core sizes $r_d|_{d=1}^{M}$)

Output: projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M}$ and core tensor $\mathcal{Y} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_M}$

Initialize projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M}$ using HOSVD(Algorithm 4.1)

while *not converged* **do**

  for $d = 1$ *to* M **do**

    `// Project onto all but the d-th projection matrices`

    $\mathcal{Z} = \mathcal{X} \times_1 \mathbf{U}^{(1)\mathsf{T}} \cdots \times_{d-1} \mathbf{U}^{(d-1)\mathsf{T}} \times_{d+1} \mathbf{U}^{(d+1)\mathsf{T}} \cdots \times_M \mathbf{U}^{(M)\mathsf{T}}$

    Set $\mathbf{U}^{(d)}$ to be the r_d leading left eigenvectors of $\mathbf{Z}_{(d)}\mathbf{Z}_{(d)}^{\mathsf{T}}$

$\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}^{(1)\mathsf{T}} \cdots \times_M \mathbf{U}^{(M)\mathsf{T}}$ `// Compute the core tensor`

Property 4.7 (PARAFAC)**.** *The following presentations are equivalent,*

1. Mode-n multiplication*:* $\mathcal{X} \approx \mathcal{Y} \times_1 \mathbf{U}^{(1)} \cdots \times_M \mathbf{U}^{(M)}$ *where is a super-diagonal tensor with the diagonal entries* $\lambda_1 \ldots \lambda_r$.

2. Matricization*:* $\mathbf{X}_{(d)} \approx \mathbf{U}^{(d)}\mathbf{\Lambda}(\mathbf{U}^{(M)} \odot \cdots \odot \mathbf{U}^{(d+1)} \odot \mathbf{U}^{(d-1)} \odot \cdots \odot \mathbf{U}^{(1)})^{\mathsf{T}}$ *where* $\mathbf{\Lambda}$ *is a diagonal matrix with diagonal entries* $\lambda_1 \ldots \lambda_r$.

3. Outer product*:* $\mathcal{X} \approx \sum_{i=1}^{r} \lambda_i \mathbf{u}_i^{(1)} \circ \cdots \circ \mathbf{u}_i^{(M)}$

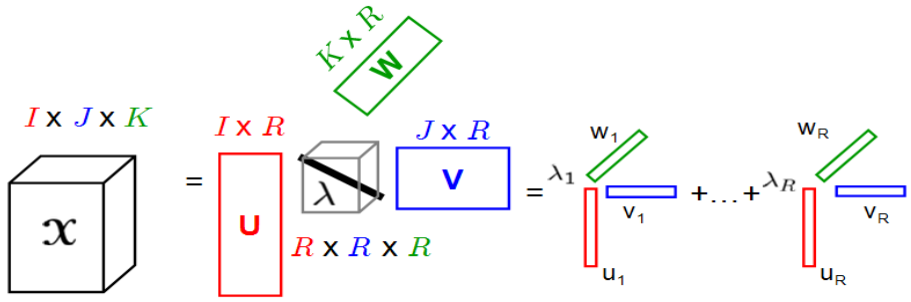


Figure 4.4: 3rd order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K} \approx \sum_{i=1}^{R} \lambda_i \mathbf{u}^{(i)} \circ \mathbf{v}^{(i)} \circ \mathbf{w}^{(i)}$

Solving the PARAFAC involves a similar alternating optimization procedure. In particular, it searches for the d-th projection matrix by fixing the rest, or more formally, the product of $\mathbf{U}^{(d)}\Lambda$ equals $\arg\min_{\mathbf{B} \in \mathbb{R}^{n_d \times R}} \left\| \mathbf{X}_{(n)} - \mathbf{B}(\mathbf{U}^{(M)} \odot \cdots \odot \mathbf{U}^{(d+1)} \odot \mathbf{U}^{(d-1)} \odot \cdots \odot \mathbf{U}^{(1)})^{\mathsf{T}} \right\|$

Algorithm 4.3: PARAFAC(tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$, core size r)

Output: projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M}$ and scaling factors $\lambda_i|_{i=1}^{r}$

Initialize projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M}$ using HOSVD

$\mathbf{S}$ is an $r \times r$ all-one matrix

for $d = 1$ *to* M **do**

  $\mathbf{C}^{(d)} = \mathbf{U}^{(d)\mathsf{T}}\mathbf{U}^{(d)}$

  $\mathbf{S} = \mathbf{S} * \mathbf{C}^{(d)}$

while *not converged* **do**

  for $d = 1$ *to* M **do**

    `// Element-wise division to remove old` $\mathbf{C}^{(d)}$`,` ϵ `for handling divided-by-0 exception`

    $\mathbf{S} = \mathbf{S}./(\mathbf{C}^{(d)} + \epsilon)$

    $\mathbf{B}^{\mathsf{T}} = \mathbf{S}^{\dagger}(\mathbf{U}^{(M)} \odot \cdots \odot \mathbf{U}^{(d+1)} \odot \mathbf{U}^{(d-1)} \odot \cdots \odot \mathbf{U}^{(1)})^{\mathsf{T}}{\mathbf{X}_{(d)}}^{\mathsf{T}}$

    $\mathbf{U}^{(d)}$ equals the column-normalized $\mathbf{B}$

    $\lambda_1 \ldots \lambda_r$ are the scaling factors

    $\mathbf{S} = \mathbf{S} * \mathbf{C}^{(d)}$ `// update` $\mathbf{S}$

which can be solved using Least square with one trick based on the property 4.2. I.e,

$$\begin{aligned}\mathbf{B}^{*\mathsf{T}} &= (\mathbf{U}^{(M)} \odot \cdots \odot \mathbf{U}^{(d+1)} \odot \mathbf{U}^{(d-1)} \odot \cdots \odot \mathbf{U}^{(1)})^{\dagger}{\mathbf{X}_{(d)}}^{\mathsf{T}} \\ &= \mathbf{S}^{\dagger}(\mathbf{U}^{(M)} \odot \cdots \odot \mathbf{U}^{(d+1)} \odot \mathbf{U}^{(d-1)} \odot \cdots \odot \mathbf{U}^{(1)})^{\mathsf{T}}{\mathbf{X}_{(d)}}^{\mathsf{T}}\end{aligned}$$

where $\mathbf{S} = (\mathbf{U}^{(M)\mathsf{T}}\mathbf{U}^{(M)}) * \cdots * (\mathbf{U}^{(d+1)\mathsf{T}}\mathbf{U}^{(d+1)}) * (\mathbf{U}^{(d-1)\mathsf{T}}\mathbf{U}^{(d-1)}) * \cdots * (\mathbf{U}^{(1)\mathsf{T}}\mathbf{U}^{(1)}) \in \mathbb{R}^{r \times r}$

Once the optimal $\mathbf{B}^{*}$ is obtained, $\mathbf{U}^{(d)}$ is the column normalized $\mathbf{B}^{*}$ and $\mathbf{\Lambda}$ is a diagonal matrix with the diagonal entries the normalized constants. The pseudocode is listed in Algorithm 4.3.

4.1.4 Other tensor related work

Supervised tensor learning: Shashua and Levin [114] first applied the rank-one tensor decomposition in linear image coding for image representation and classification. Motivated by the successes of the rank-one tensor decomposition in image analysis, Tao et al. [122] extended it for classification. Ye et al. [135] used the tensor representation to

reduce the small samples size problem in linear discriminant analysis for 2nd order tensors. Tao et al. [123] developed a general tensor discriminant analysis, which incorporated the tensor analysis and a generalized linear discriminant analysis. All these methods are offline methods and assume labeled data, which are not available for many applications. Therefore, we focus on online unsupervised setting.

Unsupervised tensor learning: Vasilescu and Terzopoulos [127] introduced the tensor singular value decomposition for face recognition. Ye [134] presented the generalized low rank approximations which extends PCA from the vectors (1st-order tensors) into matrices (2nd order tensors). Ding and Ye [49] proposed an approximation of [134]. Similar approach is also proposed in [72]. Xu et al. [132] formally presented the tensor representation for principal component analysis and applied it for face recognition. Drineas and Mahoney [56] present an algorithm that approximates the tensor SVD using biased sampling. Kolda et al. [87] apply PARAFAC on web graphs to generalize the hub and authority scores for web ranking through anchor text information.

4.2 Incremental Tensor Analysis Framework

In this section, we first formally define the notations of *tensor sequence* and *tensor stream* for streaming data. Then we introduce the Incremental Tensor Analysis (ITA) framework.

4.2.1 Data model

Definition 4.10 (Tensor Sequence)**.** *A sequence of Mth order tensors $\boldsymbol{\mathcal{X}_1} \ldots \boldsymbol{\mathcal{X}_T}$, where each $\boldsymbol{\mathcal{X}_t} \in \mathbb{R}^{n_1 \times \cdots \times n_M}$ ($1 \leq t \leq T$), is called tensor sequence. And T is the cardinality of the tensor sequence.*

In fact, we can view an Mth order tensor sequence $\boldsymbol{\mathcal{X}_1} \ldots \boldsymbol{\mathcal{X}_T}$ as a single (M+1)th order tensor with the dimensionality T on the additional mode.

Definition 4.11 (Tensor Stream)**.** *A sequence of Mth order tensor $\boldsymbol{\mathcal{X}_1} \ldots \boldsymbol{\mathcal{X}_T}$, where each $\boldsymbol{\mathcal{X}_t} \in \mathbb{R}^{n_1 \times \cdots \times n_M}$ ($1 \leq t \leq T$), is called a tensor stream if T is the maximum index that increases with time.*

Intuitively, we can consider a tensor stream is growing incrementally over time. And $\boldsymbol{\mathcal{X}_T}$ is the latest tensor in the stream. In the network monitoring example, a new 3rd order tensor arrives every hour. For the simplicity of presentation, all the tensors are of the same size, which can be relaxed in practice by zero-padding.

4.2.2 Offline Tensor Analysis

After defining the data models, now we introduce the core mining operations. First, we introduce Offline Tensor Analysis for tensor sequences which is a generalization of PCA for higher order tensors[1]. Unlike PCA, which requires the input to be vectors (1st-order tensors), OTA can accept general Mth order tensors for dimensionality reduction.

Definition 4.12 (Offline Tensor Analysis (OTA))**.** *Given a sequence of tensors* $\mathcal{X}_1 \ldots \mathcal{X}_T$*, where each* $\mathcal{X}_t|_{t=1}^{T} \in \mathbb{R}^{n_1 \times \cdots \times n_M}$*, find the projection matrices* $\mathbf{U}^{(i)}|_{i=1}^{M} \in \mathbb{R}^{n_i \times r_i}$ *and a sequence of core tensors* $\mathcal{Y}_t|_{t=1}^{T} \in \mathbb{R}^{r_1 \times \cdots \times r_M}$*, such that the reconstruction error* e *is minimized:* $e = \sum_{t=1}^{T} \|\mathcal{X}_t - \mathcal{Y}_t \prod_{i=1}^{M} \times_i \mathbf{U}^{(i)}\|$

Figure 4.5 shows an example of OTA over n 2nd order tensors.

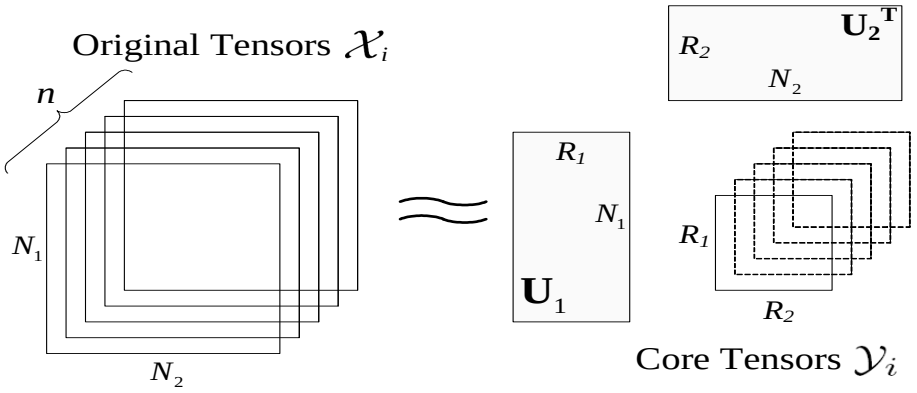


Figure 4.5: OTA projects n large 2nd order tensors $\mathcal{X}_i$ into n smaller 2nd order tensors $\mathcal{Y}_i$ with two projection matrices $\mathbf{U}_1$ and $\mathbf{U}_2$ (one for each mode).

Unfortunately, the closed form solution for OTA does not exist. An alternating projection is used to approach to the optimal projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M}$. The iterative algorithm converges in finite number of steps because each sub-optimization problem is convex. The detailed algorithm is given in Algorithm 4.4. Intuitively, it projects and matricizes along each mode; and it performs PCA to find the projection matrix for that mode. This process potentially needs more than one iteration.

In practice, this algorithm converges in a small number of iterations. In fact, Ding and Ye [49] show the near-optimality of a similar algorithm for 2nd order tensors. Therefore,

[1]Similar ideas have been proposed for 2nd- or 3rd order tensors [89, 132, 72].

Algorithm 4.4: OTA(tensor $\mathcal{X}_t|_{t=1}^{T} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$, core sizes $r_d|_{d=1}^{M}$)

Output: projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M} \in \mathbb{R}^{n_d \times r_d}$ and core tensor $\mathcal{Y}_t|_{t=1}^{T} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_M}$

Initialize projection matrix $\mathbf{U}^{(d)}|_{d=1}^{M}$ to be n_d-by-r_d truncated identity
while *not converged* **do**
  for $d = 1$ *to* M **do**
    $\mathbf{C}^{(d)} = n_d$-by-n_d all zeros matrix
    for $t = 1$ *to* T **do**
      `// Project onto all but d-th projection matrices`
      $\mathcal{Z} = \mathcal{X}_t \times_1 \mathbf{U}^{(1)\mathsf{T}} \cdots \times_{d-1} \mathbf{U}^{(d-1)\mathsf{T}} \times_{d+1} \mathbf{U}^{(d+1)\mathsf{T}} \cdots \times_M \mathbf{U}^{(M)\mathsf{T}}$
      `// Update the variance matrix`
      $\mathbf{C}^{(d)} = \mathbf{C}^{(d)} + \mathbf{Z}_{(d)}\mathbf{Z}_{(d)}^{\mathsf{T}}$
    Set $\mathbf{U}^{(d)}$ be the top r_d eigenvectors of $\mathbf{C}^{(d)}$.
for $t = 1$ *to* T **do**
  $\mathcal{Y}_n = \mathcal{X}_n \times_1 \mathbf{U}^{(1)\mathsf{T}} \cdots \times_M \mathbf{U}^{(M)\mathsf{T}}$ `// Compute the core tensors`

for time-critical applications, a single iteration is usually sufficient to obtain the near-optimal projection matrices $\mathbf{U}_d|_{d=1}^{M}$. In that case, the algorithm essentially unfolds the tensor along each mode and applies PCA on those unfolded matrices separately.

Note that OTA requires all tensors available up front, which is not possible for dynamic environments (i.e., new tensors keep coming). Even if we want to apply OTA every time that a new tensor arrives, it is prohibitively expensive or merely impossible since the computation and space requirement are unbounded. Next we present an algorithm for the dynamic setting.

4.2.3 Incremental Tensor Analysis

For tensor streams, it is not feasible to find the optimal projection matrices because new tensors keep arriving. Furthermore, it is not sensible to have the same projection matrices all the time because the characteristics of the data are likely to change over time. Instead, Incremental Tensor Analysis (ITA) assumes the continuity over time.

Definition 4.13 (Incremental Tensor Analysis (ITA))**.** *Given a new tensor* $\mathcal{X}_t \in \mathbb{R}^{n_1 \times \cdots \times n_M}$ *and old projection matrices, find the new projection matrices* $\mathbf{U}^{(i)} \in \mathbb{R}^{n_i \times r_i}|_{i=1}^{M}$ *and the*

core tensor $\mathcal{Y}_t$ *such that the reconstruction error* e_t *is small:* $e_t = \|\mathcal{X}_t - \mathcal{Y}_t \prod_{i=1}^{M} \times_i \mathbf{U}^{(i)}\|$

ITA continuously updates the model as new input tensors arrive. Compared to OTA, the differences are the following:

- *Computational efficiency:* ITA only processes each tensor once, while OTA may process input tensors multiple times. In particular, OTA performs alternating optimization over all data. ITA updates the model only based on the current data, which is more efficient.
- *Time dependency:* The model of ITA (projection matrices in particular) is changing over time while OTA computes a fixed model for all input tensors. Hence, the model of ITA is time dependent.

In terms of computational efficiency, we adopt two different techniques to efficiently update the model, namely, 1) incremental update of the covariance matrices as used in DTA(Section 4.2.4) and WTA(Section 4.2.6); 2) incremental update of projection matrices as used in STA(Section 4.2.5). In terms of time dependency, we adopt two schemes: 1) exponential forgetting in DTA and STA and 2) sliding window in WTA.

4.2.4 Dynamic Tensor Analysis

Here we present the dynamic tensor analysis (DTA), an incremental algorithm for tensor dimensionality reduction.

Intuition: The idea behind the incremental algorithm is to exploit two facts:

1. In general OTA can be computed relatively quickly once the covariance matrices[2]are available;
2. Covariance matrices can be incrementally updated without storing any historical tensor.

The algorithm processes each mode of the tensor at a time. In particular, the covariance matrix of the dth mode is updated as:

$$\mathbf{C}_d \leftarrow \mathbf{C}_d + \mathbf{X}_{(d)}^T \mathbf{X}_{(d)}$$

[2]Recall the covariance matrix of $\mathbf{X}_{(d)} \in \mathbb{R}^{(\prod_{i \neq d} n_i) \times n_d}$ is defined as $\mathbf{C} = \mathbf{X}_{(d)}^T \mathbf{X}_{(d)} \in \mathbb{R}^{n_d \times n_d}$.

Algorithm 4.5: DTA(new tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$, old projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M}$, energy matrices $\mathbf{S}^{(d)}|_{d=1}^{M}$, forgetting factor λ)

Output: projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M} \in \mathbb{R}^{n_d \times r_d}$ and core tensor $\mathcal{Y}_t|_{t=1}^{T} \in \mathbb{R}^{r_1 \times r_2 \times \cdots r_M}$

for $d = 1$ *to* M **do**

  `// Reconstruct the old covariance`

  $\mathbf{C}^{(d)} \leftarrow \mathbf{U}^{(d)}\mathbf{S}^{(d)}\mathbf{U}^{(d)\mathsf{T}}$

  `// Update the covariance matrix`

  $\mathbf{C}^{(d)} = \lambda\mathbf{C}^{(d)} + \mathbf{X}_{(d)}\mathbf{X}_{(d)}{}^{\mathsf{T}}$

  Set $\mathbf{U}^{(d)}$ be the top r_d eigenvectors of $\mathbf{C}^{(d)}$.

$\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}^{(1)\mathsf{T}} \cdots \times_M \mathbf{U}^{(M)\mathsf{T}}$ `// Compute the core tensor`

where $\mathbf{X}_{(d)} \in \mathbb{R}^{(\prod_{i \neq d} n_i) \times n_d}$ is the mode-d matricizing of the tensor $\mathcal{X}$. The updated projection matrices can be computed by diagonalization: $\mathbf{C}_d = \mathbf{U}_d\mathbf{S}_d\mathbf{U}_d^T$, where $\mathbf{U}_d$ is an orthogonal matrix and $\mathbf{S}_d$ is a diagonal matrix. The pseudo-code is listed in Algorithm 4.5. The process is also visualized in Figure 4.6.

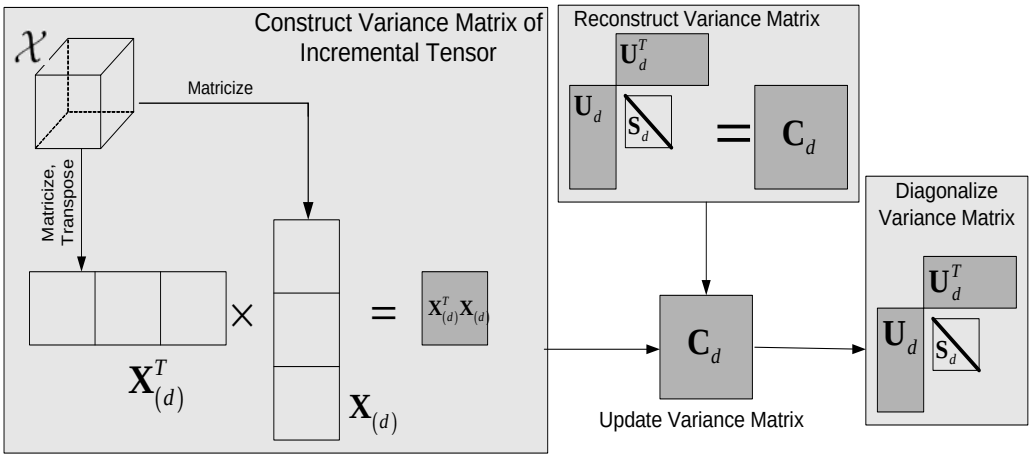


Figure 4.6: New tensor $\mathcal{X}$ is matricized along the dth mode. Then covariance matrix $\mathbf{C}_d$ is updated by $\mathbf{X}_{(d)}^T\mathbf{X}_{(d)}$. The projection matrix $\mathbf{U}_d$ is computed by diagonalizing $\mathbf{C}_d$.

Forgetting factor: Dealing with time dependent models, we usually do not treat all the timestamps equally. Often the recent timestamps are more important than the ones far away from the past [3]. More specifically, we introduce a forgetting factor into the model.

[3]Unless there is a seasonality in the data, which is not discussed in the paper.

In terms of implementation, the only change to Algorithm 4.5 is

$$\mathbf{C}_d \leftarrow \lambda \mathbf{C}_d + \mathbf{X}_{(d)}^T \mathbf{X}_{(d)}$$

where λ is the forgetting factor between 0 and 1. Herein $\lambda = 0$ when no historical tensors are considered, while $\lambda = 1$ when historical tensors have the same weights as the new tensor $\mathcal{X}$. Note that the forgetting factor is a well-known trick in signal processing and adaptive filtering [71].

Update Rank: One thing missing from Algorithm 4.5 is how to update the rank r_i. In general the rank can be different on each mode or can change over time. The idea is to keep the smallest rank r_i such that the energy ratio $\|\mathbf{S}_i\|_F / \|\mathbf{X}\|_F^2$, is above the threshold [80]. In all experiments, we set the threshold as 0.9 unless mentioned otherwise.

Complexity: The space consumption for incremental algorithm is $\prod_{i=1}^{M} n_i$ + $\sum_{i=1}^{M} n_i \times r_i$ + $\sum_{i=1}^{M} r_i$. The dominant factor is from the first term O($\prod_{i=1}^{M} n_i$). However, standard OTA requires O($n \prod_{i=1}^{M} n_i$) for storing all tensors up to time n, which is unbounded.

The computation cost is $\sum_{i=1}^{M} r_i n_i^2$ + $\sum_{i=1}^{M} n_i \prod_{j=1}^{M} n_j$ + $\sum_{i=1}^{M} r_i' n_i^2$. Note that for a medium or low order tensor (i.e., order $M \leq 5$), the diagonalization is the main cost. Section 4.2.5 introduces a faster approximation of DTA that avoids diagonalization.

For high order tensors (i.e., order $M > 5$), the dominant cost becomes O($\sum_{i=1}^{M} n_i \prod_{j=1}^{M} n_j$) from updating the covariance matrix (line 3). Nevertheless, the improvement of DTA is still tremendous compared to OTA O($T \prod_{i=1}^{M} n_i$) where T is the number of all tensors up to the current time.

4.2.5 Streaming Tensor Analysis

In this section, we present the *streaming tensor analysis* (STA), a fast algorithm to approximate DTA without diagonalization. We first introduce the key component of tracking a projection matrix. Then a complete algorithm is presented for STA.

Tracking a projection matrix: For most of time-critical applications, the diagonalization process in DTA for every new tensor can be expensive. Specifically, when the change of the covariance matrix is small, we can avoid the cost of diagonalization. The idea is to continuously track the changes of projection matrices using the SPIRIT algorithm described in Section 2.2.

Streaming tensor analysis: The goal of STA is to adjust the projection matrices smoothly as the new tensor comes in. Note that the tracking process has to be run on all modes of the new tensor. For a given mode, we first matricize the tensor $\mathcal{X}$ into a matrix $\mathbf{X}_{(d)}$ (line

Algorithm 4.6: STA(new tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$, old projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M}$, forgetting factor λ)

Output: projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M} \in \mathbb{R}^{n_d \times r_d}$ and core tensor $\mathcal{Y}_t|_{t=1}^{T} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_M}$

for $d = 1$ *to* M **do**
  foreach *column vector* $\mathbf{x}$ *in* $\mathbf{X}_{(d)}$ **do**
    Update $\mathbf{U}^{(d)}$ using Algorithm 2.1 by $\mathbf{x}$ with forgetting factor λ
$\mathcal{Y} = \mathcal{X}_{\times_1} \mathbf{U}^{(1)\mathsf{T}} \cdots \times_M \mathbf{U}^{(M)}$ `// Compute the core tensor`

2 of Algorithm 4.6), then adjust the projection matrix $\mathbf{U}_d$ by applying SPIRIT over the columns of $\mathbf{X}_{(d)}$. The full algorithm is in Algorithm 4.6. And the process is depicted in Figure 4.7.

To further reduce the time complexity, we can select only a subset of the vectors in $\mathbf{X}_{(d)}$. For example, we can sample vectors with high norms, because those potentially give higher impact to the projection matrix.

Complexity: The space complexity of STA is the same as DTA, which is only the size of the new tensor. The computational complexity is $O((\sum_i R_i) \prod_i N_i)$ which is smaller than DTA (when $R_i \ll N_i$). The STA can be further improved with a random sampling technique, i.e., by using only a subset of rows of $\mathbf{X}_{(d)}$ for updating.

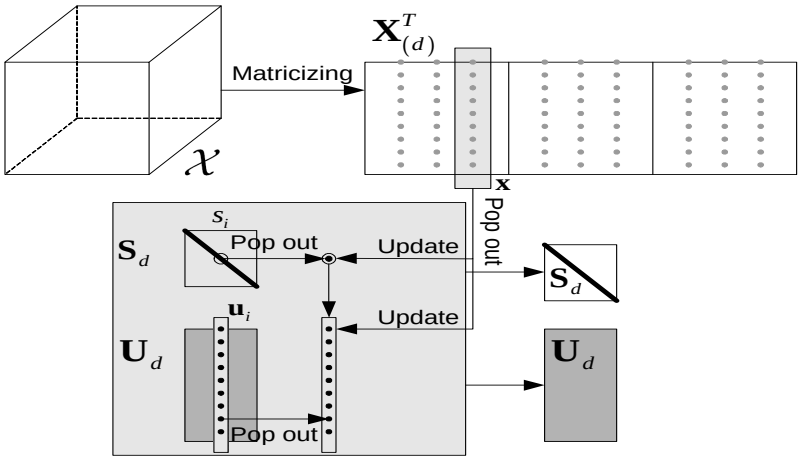


Figure 4.7: New tensor $\mathcal{X}$ is matricized along the dth mode. For every row of $\mathbf{X}_d$, we update the projection matrix $\mathbf{U}_d$. And $\mathbf{S}_d$ helps determine the update size.

4.2.6 Window-based tensor analysis

Unlike DTA or STA that use the exponential forgetting factor, WTA handles time dependency through a sliding window. Compared to the exponential forgetting scheme, the sliding window method gives the same weight to all timestamps in the window. Now we present WTA that incrementally maintains a sliding window model.

Definition 4.14 (Tensor window)**.** *A tensor window $\mathcal{D}(n, W)$ consists of a subset of a tensor stream ending at time n with size W. Formally $\mathcal{D}(n, w) \in \mathbb{R}^{W \times n_1 \times \ldots \times n_M} = \{\boldsymbol{\mathcal{X}}_{\boldsymbol{n-w+1}}, \ldots, \boldsymbol{\mathcal{X}}_{\boldsymbol{n}}\}$ where each $\boldsymbol{\mathcal{X}}_{\boldsymbol{i}} \in \mathbb{R}^{n_1 \times \cdots \times n_M}$.*

A tensor window localizes the tensor stream into a (smaller) tensor sequence with cardinality W and at particular time n. The current tensor window refers to the window ending at the current time.

After introducing the basic definitions, we now formalize the core problem, *window-based tensor analysis (WTA)*. The goal of WTA is to incrementally extract patterns from tensor streams. Two versions of WTA are presented as follows:

Problem 4.1 (Independent-window tensor analysis)**.** *Given a tensor window $\boldsymbol{\mathcal{D}} \in \mathbb{R}^{W \times n_1 \times \cdots \times n_M}$, find the projection matrices $\mathbf{U}^{(0)} \in \mathbb{R}^{W \times r_0}$ and $\mathbf{U}^{(i)}|_{i=1}^{M} \in \mathbb{R}^{n_i \times r_i}$ such that the reconstruction error is minimized: $e = \|\boldsymbol{\mathcal{D}} - \boldsymbol{\mathcal{D}} \prod_{i=0}^{M} \times_i (\mathbf{U}^{(i)\mathsf{T}} \mathbf{U}^{(i)})\|_F^2$*

The intuition of problem IW is illustrated in Figure 4.8 where a tensor window is summarized into a core tensor associated with projection matrices. The core tensor $\mathcal{Y}$ is defined as $\mathcal{D} \prod_{i=0}^{M} \times_i \mathbf{U}^{(i)\mathsf{T}}$.

Essentially, this formulation treats the tensor windows independently. Next we relax this independence assumption and propose the Moving Window problem which uses the time dependence structure on the tensor windows.

Problem 4.2 (Moving-window tensor analysis)**.** *Given the current tensor window $\boldsymbol{\mathcal{D}}(n, W) \in \mathbb{R}^{W \times n_1 \times \cdots \times n_M}$ and the old result for $\boldsymbol{\mathcal{D}}(n-1, W)$, find the new projection matrices $\mathbf{U}^{(0)} \in \mathbb{R}^{W \times r_0}$ and $\mathbf{U}^{(i)} \in \mathbb{R}^{n_i \times r_i} \; |_{i=1}^{M}$ such that the reconstruction error is minimized: $e = \|\mathcal{D}(n, W) - \mathcal{D}(n, W) \prod_{i=0}^{M} \times_i (\mathbf{U}^{(i)\mathsf{T}} \mathbf{U}^{(i)})\|_F^2$*

Iterative Optimization on tensor windows

Recall that the goal of tensor analysis is to find the set of projection matrices $\mathbf{U}^{(i)}|_{i=0}^{M}$ that minimize the reconstruction error $d(\mathcal{D}, \tilde{\mathcal{D}})$, where $d(\cdot,\cdot)$ is a divergence function between two tensors; $\mathcal{D}$ is the input tensor; $\tilde{\mathcal{D}}$ is the approximation tensor defined as $\mathcal{D}\prod_{i=0}^{M}{}_{\times_i}(\mathbf{U}^{(i)\mathsf{T}}\mathbf{U}^{(i)})$.

These types of problems can usually be solved through iterative optimization techniques, such as Tucker and PARAFAC. The principle is to optimize parameters one at a time by fixing all the rest. The benefit comes from the simplicity and robustness of the algorithms. To develop a concrete algorithm for WTA, three things have to be specified:

Initialization condition: This turns out to be a crucial component for data streams. Two different schemes for this are presented in this section, namely the Independent Window (IW) and the Moving Window (MW) schemes.

Optimization strategy: This is closely related to the divergence function $d(\cdot,\cdot)$. Gradient descent type of methods can be developed in most of cases. However, in this paper, we use the square Frobenius norm $\|\cdot\|_F^2$ as the divergence function, which naturally leads to a simpler iterated method, alternating least squares:

1. Project on all but mode-d: $\mathcal{Z} = \mathcal{D}(\prod_{i\neq d}{}_{\times_i}\mathbf{U}^{(i)\mathsf{T}}) \in \mathbb{R}^{r_0\times\cdots\times n_d\times\cdots\times r_M}$
2. Matricize $\mathcal{Z}$ along mode-d as $\mathbf{Z}_{(d)} \in \mathbb{R}^{n_d\times(\prod_{k\neq d} r_k)}$
3. Construct the covariance matrix $\mathbf{C}_d = \mathbf{Z}_{(d)}\mathbf{Z}_{(d)}{}^{\mathsf{T}}$
4. Set $\mathbf{U}^{(d)}$ as the leading r_d eigenvectors of $\mathbf{C}_d$

Convergence checking: We use the standard approach of monitoring the change of the projection matrices until it is sufficiently small.

In a streaming setting, WTA requires quick updates when a new tensor window arrives. Ultimately, this reduces to quickly setting a good initial condition for the iterative algorithm. In this section, we first introduce the independent-window tensor analysis (IW) as the baseline algorithm. Then the moving-window tensor analysis (MW) is presented that exploits the time dependence structure to quickly set a good initial condition, thereby significantly reducing the computational cost. Finally, we provide some practical guidelines on how to choose the size of the core tensors.

In general, any scheme has to balance the quality of the initial condition with the cost of obtaining it. A simple initialization can be fast, but it may require a large number of

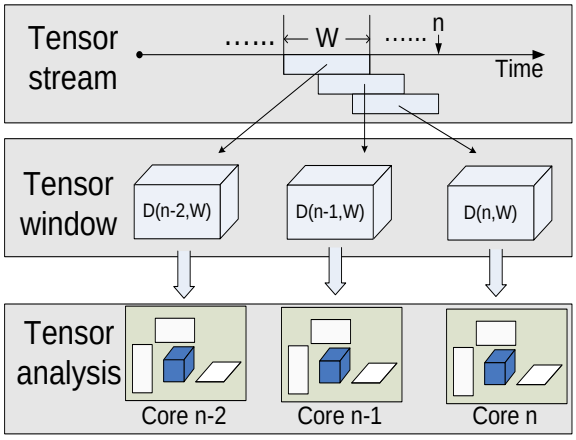


Figure 4.8: IW computes the core tensors and projection matrices for every tensor window separately, despite that fact there can be overlaps between tensor windows.

iterations until convergence. On the other hand, a complex scheme can give a good initial condition leading to much fewer iterations, but the overall benefit can still be diminished due to the time-consuming initialization.

Independent window tensor analysis (IW)

IW is a simple way to deal with tensor windows by fitting the model independently. At every timestamp a tensor window $\mathcal{D}(n, W)$ is formed, which includes the current tensor $\mathcal{D}_n$ and $W-1$ old tensors. Then we can apply the alternating least squares method (Algorithm 4.3) on $\mathcal{D}(n, W)$ to compute the projection matrices $\mathbf{U}_i|_{i=0}^{M}$. The idea is illustrated in Figure 4.8. The projection matrices $\mathbf{U}^{(i)}$ can, in theory, be any orthonormal matrices. For instance, we initialize $\mathbf{U}^{(i)}$ to be a $n_i \times r_i$ truncated identity matrix in the experiment, which leads to extremely fast initialization of the projection matrices. However, the number of iterations until convergence can be large.

Moving-window tensor analysis (MW)

MW explicitly utilizes the overlapping information of the two consecutive tensor windows to update the covariance matrices $\mathbf{C}_d|_{d=1}^{M}$. More specifically, given a tensor window $\mathcal{D}(n, W) \in \mathbb{R}^{W \times n_1 \times \cdots \times n_M}$, we have $M+1$ covariance matrices $\mathbf{C}_i|_{i=0}^{M}$, one for each mode. Note that the current window $\mathcal{D}(n, W)$ removes an old tensor $\mathcal{D}_{n-W}$ and includes a new tensor $\mathcal{D}_n$, compared to the previous window $\mathcal{D}(n-1, W)$ (see Figure 4.9).

Update modes 1 to M:. For all but the time mode, the update can be easily achieved as

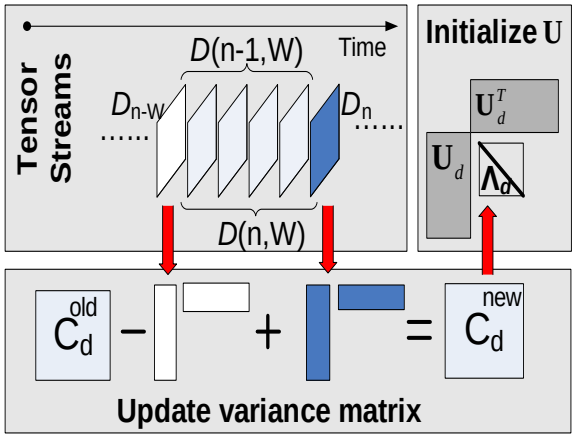


Figure 4.9: The key of MW is to initialize the projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M}$ by diagonalizing the covariance matrices $\mathbf{C}^{(d)}$, which are incrementally maintained. Note that $\mathbf{U}^{(0)}$ for the time mode is not initialized, since it is different from the other modes.

follows:

$$\mathbf{C}^{(d)} \leftarrow \mathbf{C}^{(d)} - \mathbf{X}_{n-W,(d)}\mathbf{X}_{n-W,(d)}{}^{\mathsf{T}} + \mathbf{X}_{n,(d)}\mathbf{D}_{n,(d)}{}^{\mathsf{T}}$$

where $\mathbf{X}_{n-W,(d)}(\mathbf{X}_{n,(d)})$ is the mode-d unfolding matrix of tensor $\mathcal{X}_{n-W}(\mathcal{X}_n)$: see Figure 4.9. Intuitively, the covariance matrix can be updated easily when adding or deleting rows from an unfolded matrix, since all computation only involves the added and deleted rows.

Update mode 0:. For the time mode, the analogous update does not exist. Fortunately, the iterative algorithm actually only needs initialization for all but one mode in order to start. Therefore, after initialization of the other modes, the iterated update starts from the time mode and proceeds until convergence. This gives both quick convergence and fast initialization. The pseudo-code is listed in Algorithm 4.7.

Batch update:. Often the updates for window-based tensor analysis consist of more than one tensors. Either the input tensors are coming in batches, or the processing unit waits until enough new tensors appear and then triggers the updates. In terms of the algorithm, the only difference is that the update to covariance matrices involves more than two tensors (line 4 of Algorithm 4.7).

Choosing R:

The sizes of the core tensor $r_0 \times \ldots \times r_M$ are system parameters. In practice, there are usually two mechanisms for setting them:

Algorithm 4.7: WTA(new tensor $\mathcal{D}_{\boldsymbol{n}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$, old tensor $\mathcal{D}_{\boldsymbol{n-W}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$, old covariance matrices $\mathbf{C}^{(d)}|_{d=1}^{M}$, $\mathbb{R}^{r_0 \times \cdots \times r_M}$)

Output: new covariance matrix $\mathbf{C}^{(d)}|_{d=1}^{M}$, $\mathbb{R}^{r_0 \times \cdots \times r_M}$, projection matrices $\mathbf{U}^{(d)}|_{d=1}^{M} \in \mathbb{R}^{n_d \times r_d}$ and core tensor $\mathcal{Y}_{\boldsymbol{t}}|_{t=1}^{T} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_M}$

```
// Initialize every mode except time
```

for $d = 1$ *to* M **do**

  Mode-d matricize $\mathcal{D}_{\boldsymbol{n-W}}(\mathcal{D}_{\boldsymbol{n}})$ as $\mathbf{D}_{n-W,(d)}(\mathbf{D}_{n,(d)})$

  Update $\mathbf{C}^{(d)} \leftarrow \mathbf{C}^{(d)} - \mathbf{D}_{n-W,(d)}^{T}\mathbf{D}_{n-W,(d)} + \mathbf{D}_{n,(d)}^{T}\mathbf{D}_{n,(d)}$

  Diagonalization $\mathbf{C}^{(d)} = \mathbf{U}^{(d)}\mathbf{\Lambda}_d\mathbf{U}^{(d)\mathsf{T}}$

  Truncate $\mathbf{U}^{(d)}$ to first r_d columns

Apply the iterative algorithm with the new initialization

- **Reconstruction-guided**: The larger r_is are, the better approximation we can get. But the computational and storage cost will increase accordingly (see Figure 4.20 for detailed discussion). Therefore, users can set a desirable threshold for the error, then the proper sizes of r_i can be chosen accordingly.
- **Resource-guided**: If the user has a resource constraint on the computation cost, memory or storage limit, the core tensors can also be adjusted based on that.

In the former case, the size of the core tensors may change depending on the streams' characteristics, while in the latter case, the reconstruction error may vary over time.

4.3 Experiments

In this section, we first evaluate DTA and STA in Section 4.3.1; then study WTA in Section 4.3.2.

4.3.1 Evaluation on DTA and STA

Datasets

The Network Datasets: `IP2D`, `IP3D`

The traffic trace consists of TCP flow records collected at the backbone router of a class-B university network. Each record in the trace corresponds to a directional TCP flow

name	description	dimension	timestamps
IP2D	Network 2D	500-by-500	1200
IP3D	Network 3D	500-by-500-by-100	1200
DBLP	DBLP data	4584-by-3741	11

Figure 4.10: Three datasets

between two hosts through a server port with timestamps indicating when the flow started and finished.

With this traffic trace, we study how the communication patterns between hosts and ports evolve over time, by reading traffic records from the trace, simulating network flows arriving in real time. We use a window size of an hour to construct a source-destination 2nd order tensors and source-destination-port 3rd order tensor. For each 2nd order tensor, the modes correspond to source and destination IP addresses, respectively, with the value of each entry (i, j) representing the total number of TCP flows (packets) sent from the i-th source to the j-th destination during an hour. Similarly, the modes for each 3rd order tensor corresponds to the source-IP address, the destination-IP address and the server port number, respectively.

Because the tensors are very sparse and the traffic flows are skewed towards a small number of dimensions on each mode, we select only n_1=n_2=500 sources and destinations and n_3=100 port numbers with high traffic. Moreover, since the values are very skewed, we scaled them by taking the logarithm (and actually, $\log(x + 1)$, to account for $x = 0$), so that our tensor analysis is not dominated by a few very large entries. All figures are constructed over a time interval of 1,200 timestamps(hours).

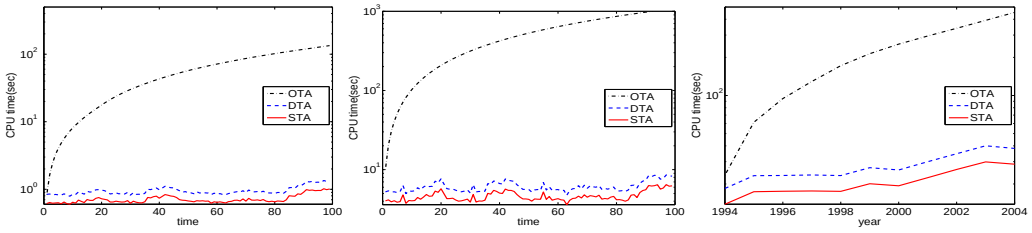
DBLP Bibliographic Data Set:

Based on the DBLP data [3], we generate author-keyword 2nd order tensors of the KDD and VLDB conferences from year 1994 to 2004 (one tensor per year). The entry (a, k) in such a tensor is the number of papers that author a has published using keyword k during that year. In total, there are 4,584 authors and 3,741 keywords. Note that the keywords are generated from the paper title after proper stemming and stop-word removal.

All the experiments are performed on the same dedicated server with four 2.4GHz Xeon CPUs and 12GB memory. For each experiment, we repeat it 10 times, and report the mean.

Computational cost

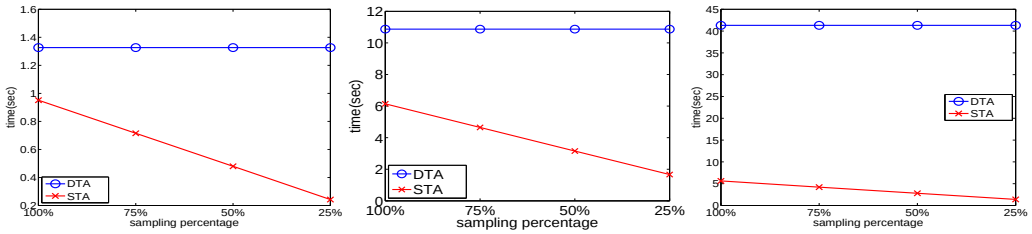
We first compare three different methods, namely, offline tensor analysis (OTA), dynamic tensor analysis (DTA), streaming tensor analysis (STA), in terms of computation time for different datasets. Figure 4.11 shows the CPU time in logarithm as a function of elapse time. Since the new tensors keep coming, the cost of OTA increases linearly[4]; while DTA and STA remains more or less constant. Note that DBLP in Figure 4.11(c) shows lightly increasing trend on DTA and STA because the tensors become denser over time (i.e., the number of published paper per year are increasing over time), which affects the computation cost slightly.



(a)IP2D (b) IP3D (c) DBLP

Figure 4.11: Both DTA and STA use much less time than OTA over time across different datasets

We show that STA provides an efficient way to approximate DTA over time, especially with sampling. More specifically, after matricizing, we sample the vectors with high norms to update the projection matrices. Figure 4.12 shows the CPU time vs. sampling rate, where STA runs much faster compared to DTA.



(a) IP2D (b) IP3D (c) DBLP

Figure 4.12: STA uses much less CPU time than DTA across different datasets

[4]We estimate CPU time by extrapolation because OTA runs out of the memory after a few timestamps.

Accuracy comparison

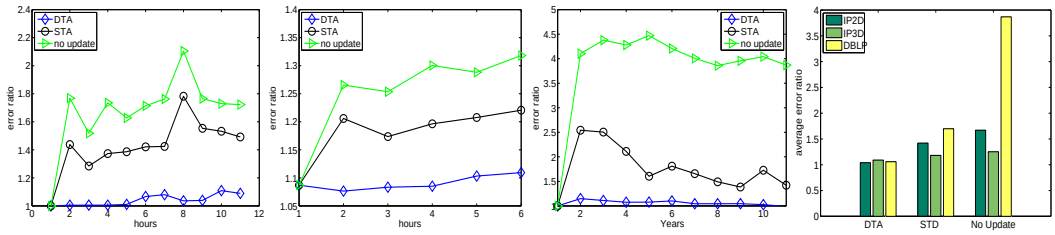
Now we evaluate the approximation accuracy of DTA and STA compared to OTA.

Performance metric: Intuitively, the goal is to be able to compare how accurate each tensor decomposition is to the original tensors. Therefore, reconstruction error provides a natural way to quantify the accuracy. Error can always be reduced when more eigenvectors are included (more columns in the projection matrices). Therefore, we fix the number of eigenvectors in the projection matrices for all three methods such that the reconstruction error for OTA is 20%. And we use the error ratios between DTA/STA to OTA as the performance indices.

Evaluation results: Overall the reconstruction error of DTA and STA are close to the expensive OTA method (see Figure 4.13(d)). Note that the cost for doing OTA is high in both space and time complexity. That is why only a few timestamps are shown in Figure 4.13 since after that point OTA runs out of the memory.

In more details, Figure 4.13(a)-(c) plot the error ratios over time for three datasets. There we also plot the one that never updates the original projection matrices as a lower-bound baseline.

DTA performs very close to OTA, besides much lower computational cost of DTA. An even cheaper method, STA, usually gives a good approximation to DTA (see Figure 4.13(a) and (b) for IP2D and IP3D). But note that STA performs considerably worse in DBLP in Figure 4.13(c) because an adaptive subspace tracking technique as STA cannot keep up the big changes of the DBLP tensors over consecutive timestamps. Therefore, STA is only recommended for the fast incoming with significant time-dependency (i.e., the changes over consecutive timestamps should not be too big).



(a) IP2D (b) IP3D (c) DBLP (d) Average

Figure 4.13: Reconstruction error over time

4.3.2 Evaluation on WTA

Name	Aspects	Dimensionality
SENSOR	(time, node, type)	$(65534, 52, 4)$
MACHINE	(time, machine, type)	$(5102, 9, 21)$

Figure 4.14: Dataset summary

Environmental sensor data (SENSOR):

Description: The sensor data are collected from 52 MICA2 Mote sensors placed in a lab, over a period of a month. Every 30 seconds each Mote sensor sends to the central collector via wireless radio four types of measurements: light intensity, humidity, temperature, and battery voltage. In order to compare different types, we scale each type of measurement into zero mean and unit variance. This calibration process can actually be done online since mean and variance can be easily updated incrementally. Note that we place equal weight on all measurements across all nodes. However, other weighting schemes can be easily applied, based on the application.

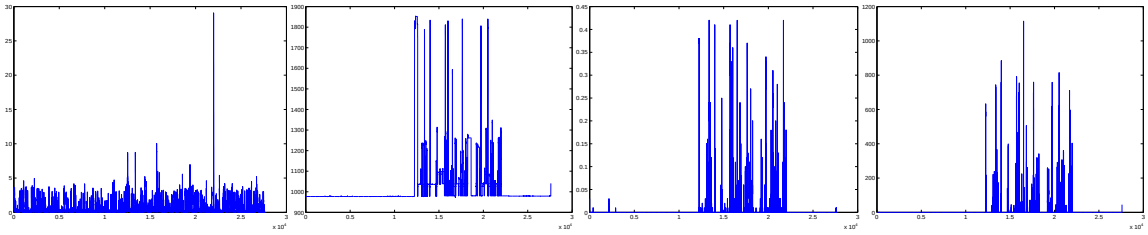
Characteristics: The data are very bursty but still correlated across locations and time. For example, the measurements of same type behave similarly, since all the nodes are deployed in the same lab.

Tensor construction: By scanning through the data once, the tensor windows are incrementally constructed and processed/decomposed. More specifically, every tensor window is a 3-mode tensor with dimensions (W, 52, 4) where W varies from 100 to 5000 in the experiments.

Server monitoring data (MACHINE):

Description: The data are collected from 9 hosts in the same cluster, all of them running a large distributed simulation program during the collection period of three days. In particular, we monitor 21 different metrics for each machine. Four example metrics on machine 1 are plotted in Figure 4.15.

Characteristics: The MACHINE data are even more bursty than SENSOR, with no clear periodicity. However, the data are still correlated locally.



(a) Load (b) Number of tasks (c) User CPU (d) Byte received

Figure 4.15: MACHINE measurements are bursty and correlated but without any clear periodicity.

Tensor construction: Similar to SENSOR, every tensor window is a 3-mode tensor with dimensionality (W, 9, 21), where W varies from 100 to 5000 in the experiments.

Computational efficiency

In this section we compare two different methods, independent-window tensor analysis (IW) and moving-window tensor analysis (MW), in terms of computation time and of number of iterations to convergence, for both SENSOR and MACHINE datasets.

Parameters: This experiment has three parameters:

- **Window size W**: the number of timestamps included in each tensor window.
- **Step ratio S**: the number of newly arrived tensors in the new tensor windows divided by window size W (a ratio between 0 and 1).
- **Core tensor size**: (r_0, r_1, r_2) where r_0 is the size of the time mode.

IW and MW reach the same error level[5] across all the experiments, since we use the same termination criterion for the iterative algorithm in both cases.

Stability over time.: Figure 4.16 shows the CPU time as a function of elapsed time, where we set $W = 1000$, $S = .2$ (i.e. 20% new tensors). Overall, CPU time for both IW and MW exhibits a constant trend. MW achieves about 30% overall improvement compared to IW, on both datasets.

[5]Reconstruction error is $e(\mathcal{D}) = \frac{||\mathcal{D}-\tilde{\mathcal{D}}||_F^2}{||\mathcal{D}||_F^2}$, where the tensor reconstruction is $\tilde{\mathcal{D}} = \mathcal{D} \prod_{\times_i} (\mathbf{U}_i \mathbf{U}_i^T)$

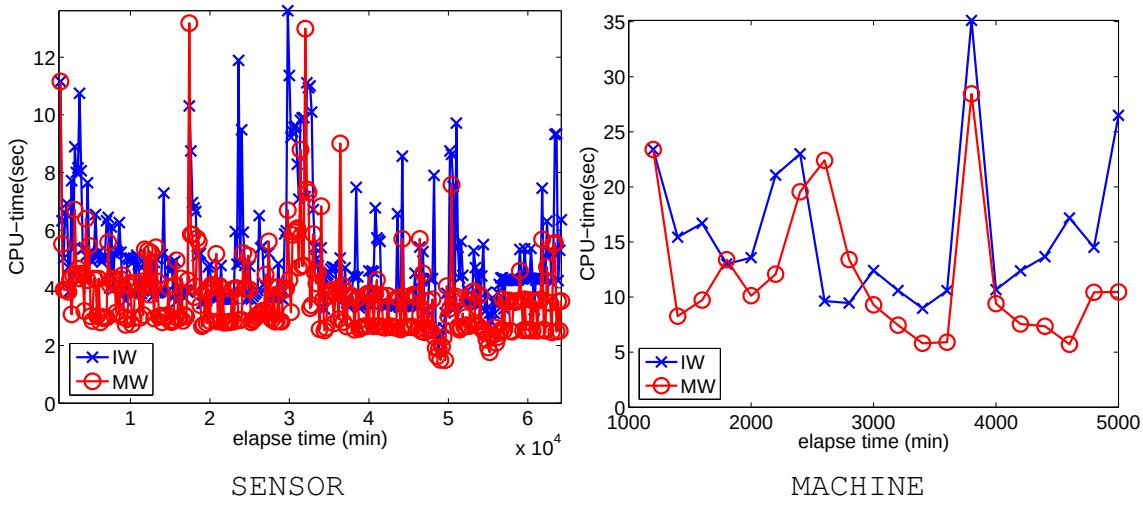


Figure 4.16: CPU cost over time: both IW and MW give a constant trend over time but MW runs 30% faster overall.

The performance gain of MW comes from its incremental initialization scheme. As shown in Figure 4.17, the CPU time is strongly correlated with the number of iterations. As a result of the clever initialization scheme of MW, which reduces the number of iterations needed, MW is much faster than IW.

Consistent across different parameter settings.:

- **Window size**: Figure 4.18 shows CPU time (in log-scale) vs. window size W. As expected, CPU time is increasing with window size. Note that the MW method achieves large computational saving across all sizes, compared to IW.

- **Step size**: Figure 4.19 presents step size vs. CPU time. MW is much faster than IW across all settings, even when there is no overlap between two consecutive tensor windows (i.e., step size equals 1). This clearly shows that the importance of a good initialization for the iterative algorithm.

- **Core tensor size**: We vary the core tensor size along the time-mode and show the CPU time as a function of this size (see Figure 4.20). Again, MW performs much faster than IW, over all sizes. Similar results are achieved when varying the sizes of the other modes.

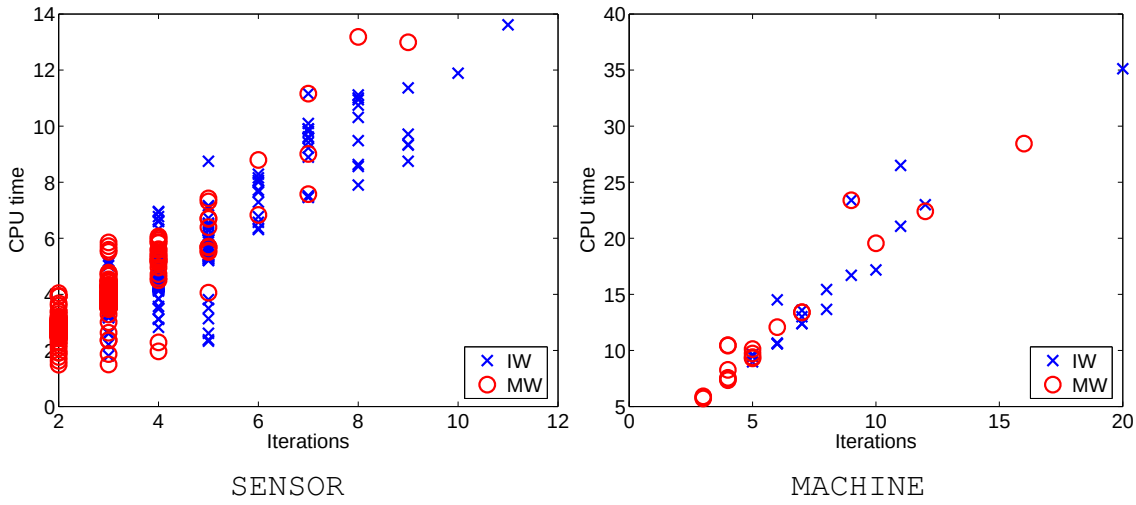


Figure 4.17: Number of iterations is perfectly correlated with CPU time. MW converges using much fewer iterations and CPU time than IW.

4.4 Case studies

In this section, we first illustrate the applications of DTA, STA in Section 4.4.1, then present the applications of WTA on Section 4.4.2.

4.4.1 Applications of DTA and STA

We now illustrate two practical applications of DTA or STA: 1) *Anomaly detection*, which tries to identify abnormal behavior across different tensors as well as within a tensor; 2) *Multi-way latent semantic indexing (LSI)*, which finds the correlated dimensions in the same mode and across different modes.

Anomaly Detection

We envision the abnormal detection as a multi-level screening process, where we try to find the anomaly from the broadest level and gradually narrow down to the specifics. In particular, it can be considered as a three-level process for tensor streams: 1) given a sequence of tensors, identify the abnormal ones; 2) on those suspicious tensors, we locate the abnormal modes; 3) and then find the abnormal dimensions of the given mode. In

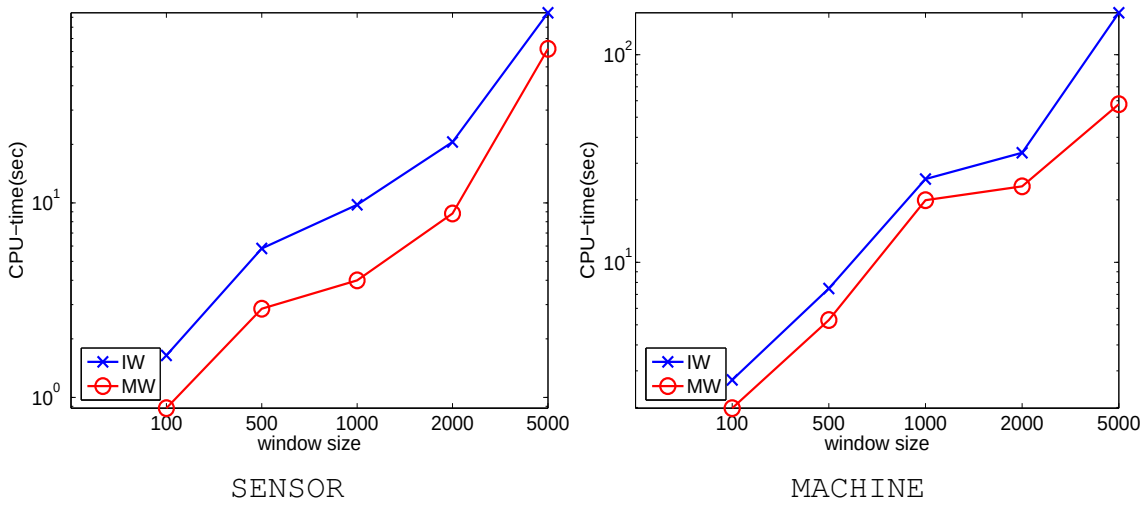


Figure 4.18: CPU cost vs. window size: The CPU time (log-scale) shows the big difference between IW and MW for all W and for both datasets.

the network monitoring example, the system first tries to determine when the anomaly occurs; then it tries to find why it occurs by looking at the traffic patterns from sources, destinations and ports, respectively; finally, it narrows down the problem on specific hosts or ports.

For level one (tensor level), we model the abnormality of a tensor $\mathcal{X}$ by its reconstruction error:

$$e_i = \|\mathcal{X}_i - \mathcal{Y}_i \prod_{l=1}^{M} \times_l \mathbf{U}^{(l)}\|_F = \|\mathcal{X}_i - \mathcal{X}_i \prod_{l=1}^{M} \times_l \mathbf{U}^{(l)\mathsf{T}}\mathbf{U}^{(l)}\|_F^2.$$

For level two (mode level), the reconstruction error of the l-th mode only involves one projection matrix $\mathbf{U}^{(l)}$ for a given tensor $\mathcal{X}$:

$$e_d = \|\mathcal{X} - \mathcal{X} \times_l \mathbf{U}^{(l)\mathsf{T}}\mathbf{U}^{(l)}\|_F^2.$$

For level three (dimension level), the error of dimension d on the l-th mode is just the reconstruction of the tensor slice of dimension d along the lth mode.

How much error is too much? We answer this question in the typical way: If the error at time T is a few (say, 2) standard deviations away from the mean error so far, we declare the tensor $\mathcal{X}_T$ as abnormal. Formally, the condition is as follows:

$$e_{T+1} \geq \text{mean}\left(e_i|_{i=1}^{T+1}\right) + \alpha \cdot \text{std}\left(e_i|_{i=1}^{T+1}\right).$$

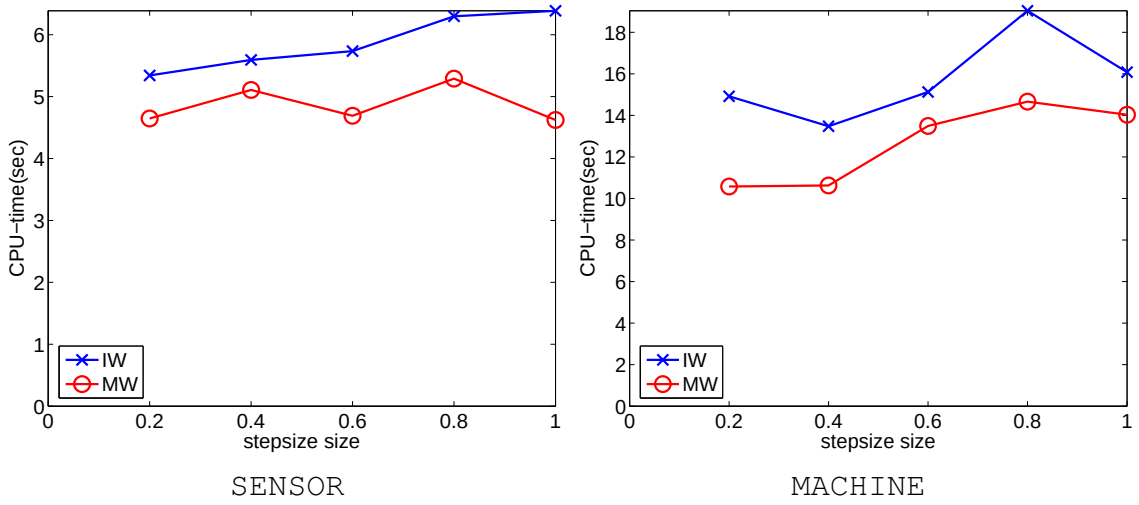


Figure 4.19: CPU cost vs. step size: MW consistently outperforms IW for all step sizes, which indicates the importance of a good initialization for the iterative process.

Authors	Keywords	Year
michael carey,michael stonebraker, h. jagadish,hector garcia-molina	queri,parallel,optimization, concurr,objectorient	1995
surajit chaudhuri,mitch cherniack,michael stonebraker,ugur etintemel	distribut,systems,view,storag, servic,process,cach	2004
jiawei han,jian pei,philip s. yu, jianyong wang,charu c. aggarwal	streams,pattern,support,cluster, index,gener,queri	2004

Table 4.1: Example clusters: first two lines databases groups, last line data mining group.

Multi-way Latent Semantic Indexing

The goal of the multi-way LSI is to find high correlated dimensions within the same mode and across different modes, and monitor them over time. Consider the DBLP example, author-keyword over time, Figure 4.21 shows that initially (in $\mathcal{X}_1$) there is only one group, DB, in which all authors and keywords are related to databases; later on (in $\mathcal{X}_n$) two groups appear, namely, databases (DB) and data mining (DM).

Correlation within a mode: A projection matrix gives the correlation information among dimensions for a given mode. More specifically, the dimensions of the lth mode can be grouped based on their values in the columns of $\mathbf{U}^{(l)}$. The entries with high absolute values

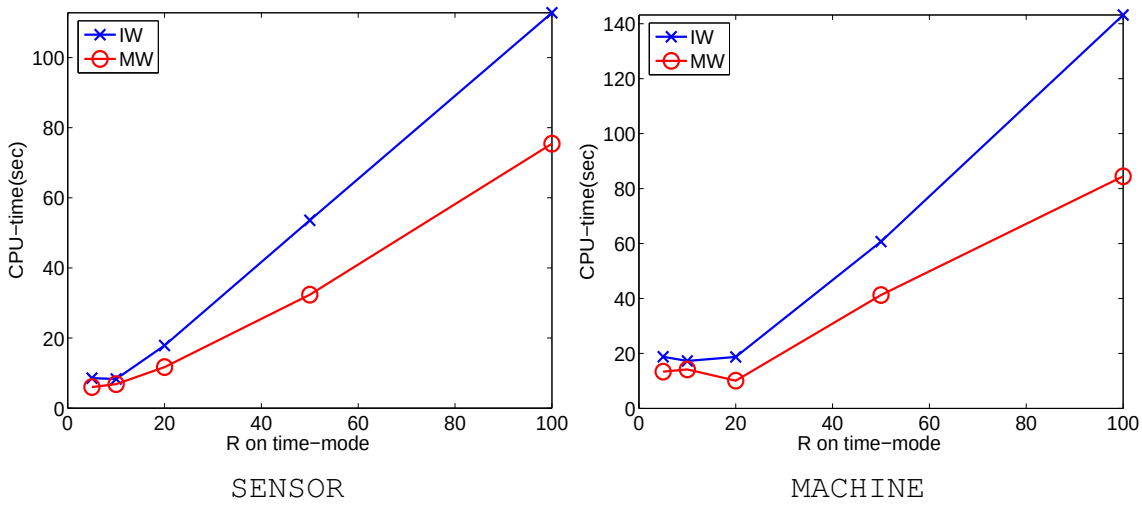


Figure 4.20: CPU time vs. core tensor size: CPU time increases linearly with respect to the core tensor size on time mode.

in a column of $\mathbf{U}^{(l)}$ correspond to the important dimensions in the same "concept".

In the DBLP example shown in Figure 4.21, $\mathbf{U}^{(K)}$ corresponds to the keyword concepts. First and second columns are the DB and DM concepts, respectively. The circles of $\mathbf{U}^{(A)}$ and $\mathbf{U}^{(K)}$ indicate the influential authors and keywords in DM concept, respectively. Similarly, the stars are for the DB concept. An example of actual keywords and authors is in Table 4.1.

Correlations across modes: The interesting aspect of DTA is that the core tensor $\mathcal{Y}$ provides indications on the correlations of different dimensions across different modes. More specifically, a large entry in $\mathcal{Y}$ means a high correlation between the corresponding columns in all modes. For example in Figure 4.21, the large values of $\mathcal{Y}_i$ (in the shaded region) activate the corresponding concepts of the tensor $\mathcal{X}_i$. For simplicity, we described a non-overlapping example, however, groups may overlap which actually happens often in real datasets.

Correlations across time: And the core tensor $\mathcal{Y}_i$s also capture the temporal evolution of concepts. In particular, $\mathcal{Y}_1$ only activates the DB concept; while $\mathcal{Y}_n$ activates both DB and DM concepts.

Note that DTA monitors the projection matrices over time. In this case, the concept space captured by projection matrix $\mathbf{U}^{(i)}$s are also changing over time.

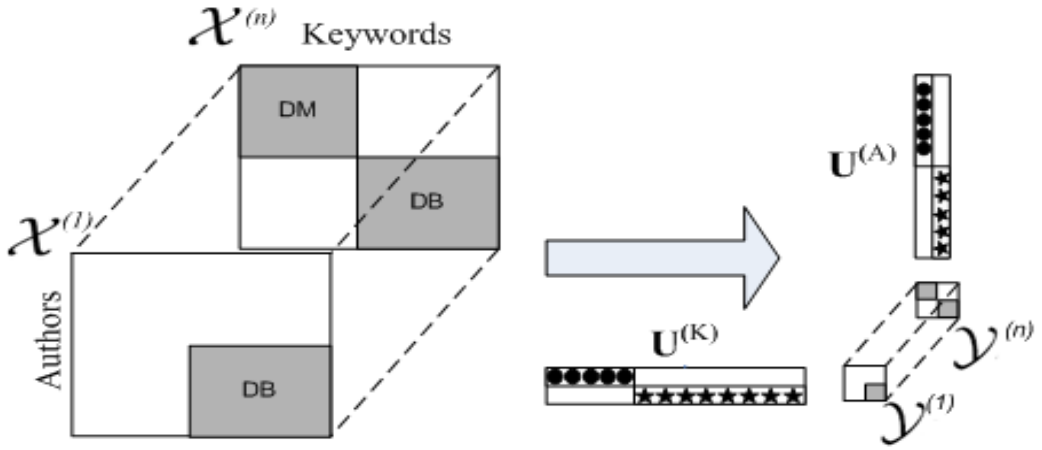


Figure 4.21: $\mathbf{U}^{(A)}$ and $\mathbf{U}^{(K)}$ capture the DB (stars) and DM (circles) concepts in authors and keywords, respectively; initially, only DB is activated in $\mathcal{Y}_1$; later on both DB and DM are in $\mathcal{Y}_n$.

4.4.2 Applications of WTA

In this section, we introduce a mining application of window-based tensor analysis on the SENSOR dataset. The goal is to find high correlated dimensions within the same mode (aspect) and across different modes (aspects) of a tensor window. Consider the SENSOR example for environmental monitoring. It can not only identify the cluster behaviors (correlation) on the *time*, *location* and *type* modes, but also detect the cross-mode association, i.e., answering the question of "how do the three aspects interact with each other?"

Correlation within a mode/aspect: A projection matrix gives the correlation information among dimensions of a given mode. More specifically, the dimensions of the i-th mode can be grouped based on their values in the columns of $\mathbf{U}^{(i)}$. The entries with high absolute values in a column of $\mathbf{U}^{(i)}$ correspond to the important dimensions of the same "concept."

In the SENSOR data, Figure 4.22 shows the first and second columns of $\mathbf{U}^{(i)}$ for all three modes. Along the time mode, Figure 4.22(a1) gives the daily periodic pattern where high values correspond to daytime and low values to nighttime. Figure 4.22(a2) captures the residual "noise" in the data. Along the sensor mode, Figure 4.22(b1) indicates the main variation of different sensors, i.e., the relative magnitude of each sensor's "participation" in this pattern. Figure 4.22(b2) shows just three participating abnormal sensors 18, 20, and 47. Along the type mode, Figure 4.22(c1) shows that Light, Temperature and Voltage are positively correlated with each other and negatively correlated with Humidity[6]. Figure 4.22(c2) shows another pattern focusing on Voltage.

[6]Voltage is indeed correlated with temperature due to the physical design of MICA2 sensors.

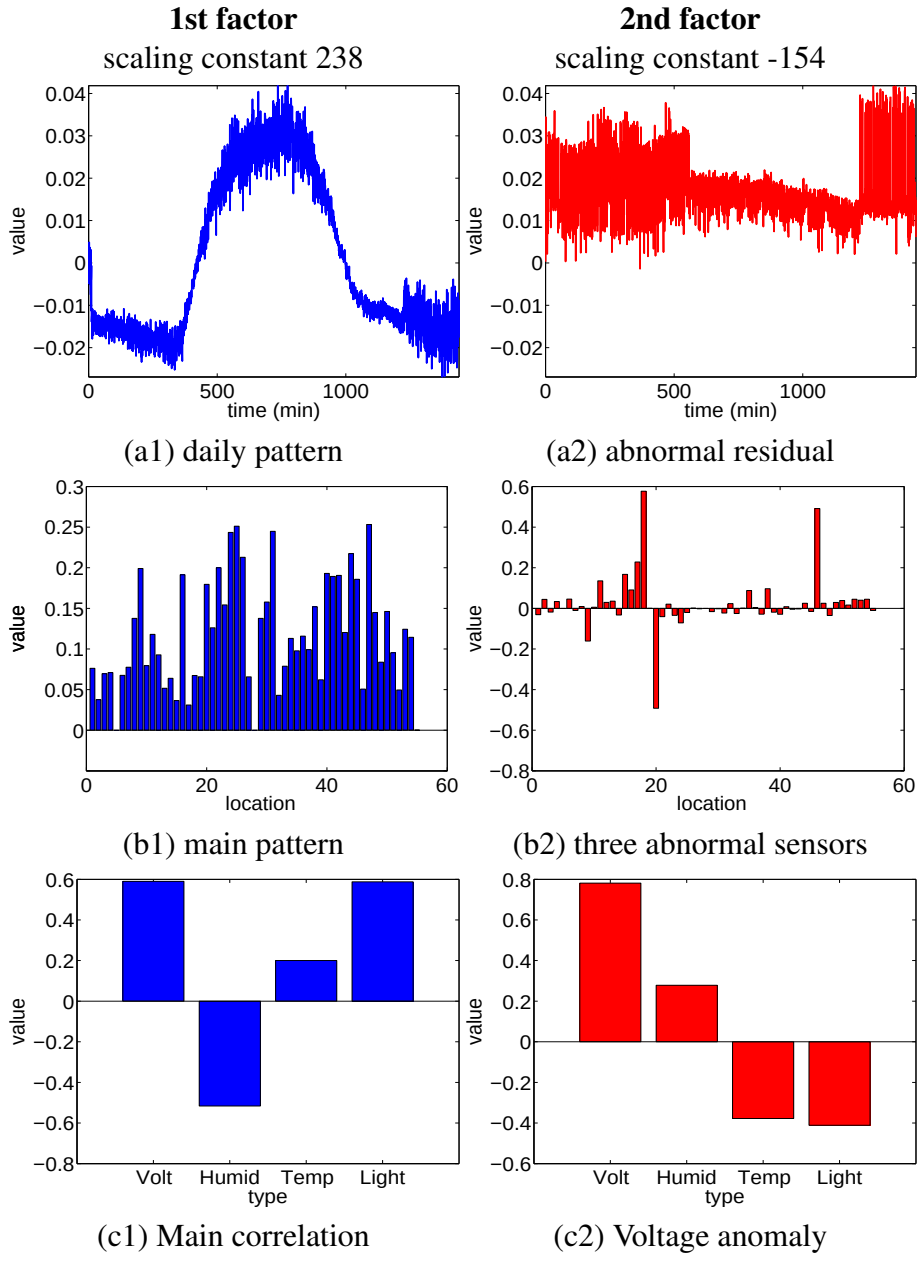


Figure 4.22: WTA on environmental data, daily window

Correlations across modes/aspects: The core tensor $\mathcal{Y}$ can give indications on the correlations across different modes. More specifically, a large entry in $\mathcal{Y}$ in absolute value means a high correlation between the corresponding dimensions of different modes.

For the SENSOR example, $\mathcal{Y}(1,1,1)$ has the highest value (238), which means that the first columns of $\mathbf{U}^{(i)}|_{i=0}^{2}$ (first factor in Figure 4.22) act together closely to characterize the original data. More specifically, the tensor product of those first columns should be scaled by 238 in order to approximate the tensor window. Intuitively, the daily periodicity, sensor variation, and type correlation are all positively associated together.

The core tensor weight $\mathcal{Y}(2,2,2)$ for the second factor is -154, which means that the tensor product of all second factors should be scaled by -154. Essentially, it means that sensor 18 and 47 have lower voltage level while sensor 20 has higher voltage level compared to the rest, because after negative scaling sensor 18 and 47 have negative values while sensor 20 has a big positive value (see Figure 4.22(b2)), whereas the type concentration is mainly on Voltage (see Figure 4.22(c2)). Combining the information in (a2,b2,c2), we can conclude that sensor 18,20 and 47 have extreme values on voltage across all time.

In summary, the first factor describe the main patterns in the data, while the second factor capture the abnormal behaviors.

Correlations across time: Over time, as the window moves forward, the correlations are also drifting. In particular, the projection matrices along different modes are adjusting over time. For SENSOR data, the time aspect are periodic with variation, while the location and type aspect are more or less stable over time, which is indeed identified by running WTA over time as shown in Figure 4.23.

4.5 Chapter summary: tensor mining

Numerous mining applications can be handled by matrices, the powerful SVD/PCA, and its variants and extensions. However, they all fall short when we want to study multiple modes, for example, time-evolving traffic matrices, time-evolving dataCubes, social networks with labeled edges, to name a few.

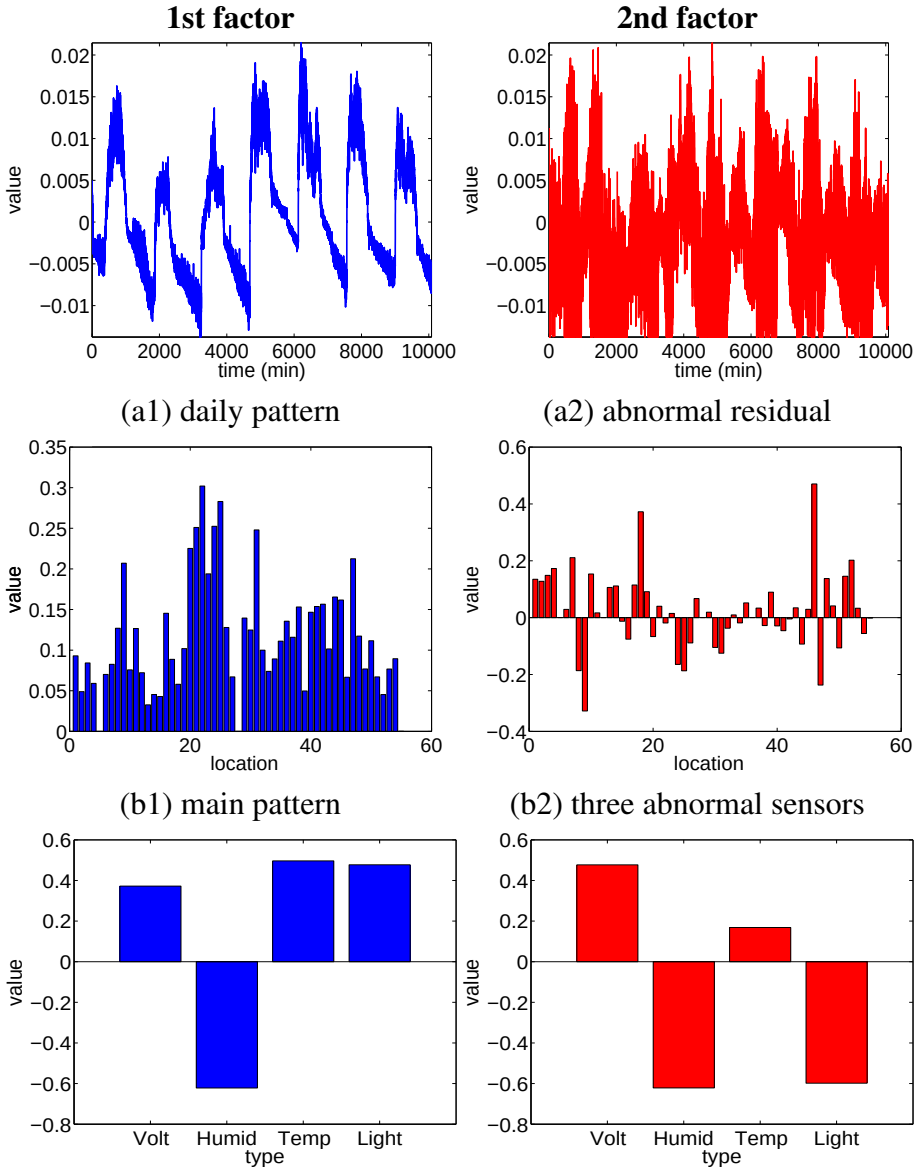
We show how to solve these higher order problems, by introducing the concept and vast machinery of *tensors*. Our contributions are the following:

We introduce *tensors* and specifically *tensor streams* to solve even more general streaming problems than those in the past. Our approach is applicable to all time-evolving settings, including co-evolving time series, data streams and sensor networks, time-evolving graphs (even labeled ones), time-evolving social networks.

We propose three new tools, the *dynamic*, the *streaming* and the *window-based tensor analysis* (DTA, STA and WTA) which incrementally mine and summarize large tensors, saving space and detecting patterns. DTA, STA and WTA are *fast*, *nimble* (since they avoid

storing old tensors), and *fully automatic*, without requiring any user-defined parameters.

We provide experiments on several real, large datasets. With respect to efficiency, all methods give significant performance gain compared to offline tensor analysis. With respect to effectiveness, we applied our methods to anomaly detection and "multi-way LSI"; in both settings our tools found interesting and explainable patterns.



(a1) daily pattern (a2) abnormal residual

(b1) main pattern (b2) three abnormal sensors

(c1) Main correlation (c2) Voltage anomaly

Figure 4.23: WTA on environmental data, weekly window

Chapter 5

Conclusions

Modern applications often face huge amounts of data that arrive in real-time. A big challenge in the IT industry is to manage, utilize and understand this data in real-time. The data is of *high dimensionality*, contains *multiple aspects* and arrives in high speed. The data patterns *change over time*, e.g., the workload is different depending on the time of the week. To deal with such dynamic and highly complex data, we propose the Incremental Tensor Analysis (ITA) framework to find patterns and anomalies incrementally.

For example, in data centers, nearly every measurement needs to be monitored such as power consumption, temperature and humidity (in the room), and workload and network flow (in every machine). It is physically and economically impossible to have human administrators monitoring everything. In order to monitor all the data streams in the data center, smart monitoring programs need to be developed. They should be able to summarize the data on the fly without using too many resources and require few tuning parameters or even none. These kinds of applications require efficient and effective data mining algorithms. The challenges for incremental pattern discovery concerns (a) high dimensionality and multiple orders of the data, (b) space and computational requirements, (c) how to minimize or even eliminate the need for parameter tuning.

To address these challenges, we first propose the *tensor stream* as a general dynamic data model for diverse applications. Under this model, data streams and time-evolving graphs become the 1st- and 2nd-order special cases, respectively.

Second, within the tensor stream perspective, we envision Incremental Pattern Discovery as an online dimensionality reduction process. We developed an array of mining tools including the incremental PCA (SPIRIT); and its high-order generalizations (DTA/STA/WTA); sparsity preserving decompositions, CMD; and incremental commu-

nity detection based on the MDL principle, GraphScope.

Third, we demonstrate the effectiveness and efficiency of our methods on large, real datasets. Specifically. for 1st order tensor streams, we develop the monitoring system InteMon using SPIRIT, which is deployed to monitor 100 machines with petabyte storage. For 2nd order tensor stream, we apply CMD on 500Gb of network traffic data, achieving 10X less space and time requirements compared to the previous state of the arts; we also apply GraphScope on Enron email graphs to identify important patterns. For 3rd order tensor streams, we apply DTA/STA/WTA on environmental sensors and discover cross-mode correlations, which confirmed important patterns in the data.

This thesis focuses on mining different orders of tensor streams. The mining results are presented as core tensors along with projection matrices. To realize these results, the design choices include 1) the order of the data, 2) the computational and storage requirement, 3) the parameter setting and 4) the properties of the projection matrices.

For stream mining (or first-order tensor streams), we developed the SPIRIT algorithm which has the following characteristics: 1) It works with first-order tensor streams. 2) It performs in an incremental fashion without the need to buffer any historical data. Its computational cost per timestamp increases linearly with the dimensionality (independent of the duration). 3) Only two parameters are needed, i.e., the min and max of the approximation accuracy thresholds. 4) It maintains a single orthogonal projection matrix.

SPIRIT has been applied to a variety of datasets, including the environmental sensor data such as temperature, humidity, light intensity and chlorine levels in water distribution systems. Also, the InteMon system [73] utilizes SPIRIT for compression and anomaly detection on a data center. More specifically, it successfully identifies the abnormal behavior in the A/C unit and is able to achieve 10-to-1 compression ratio on the monitoring data. We also extend SPIRIT to work in a distributed environment and apply it as a underlying computational kernel for two applications: local correlation measures and privacy preservation. All of them provide streaming capability for various application domains.

For graph mining (second-order tensor streams), we introduced two techniques: *Compact Matrix Decomposition (CMD)* and *GraphScope*. They concern two different characteristics of real time-evolving graphs, i.e., the sparsity and time-dependency.

CMD deals with sparse graphs and has the following characteristics. 1) It works with graphs (second tensors); 2) It requires only scanning a single graph/matrix twice, and buffering a small number of columns and rows. Its computational cost increases linearly on the dimensionality (the number of columns and rows of the original matrix). 3) Only two parameters need to be set, namely the number of sampled columns and rows. 4) The projection matrices are actual columns and rows from original matrix.

Furthermore, CMD provides a concise and intuitive summary of the original graphs. In particular, CMD achieves orders of magnitude improvements on space and speed compared to the previous state of the arts (on several real large datasets). Furthermore, the results of CMD can also help with anomaly detection. e.g., CMD has successfully been applied to network traffic flows. As a result, a worm-like hierarchical scanning behavior is identified using CMD.

GraphScope is another proposed technique with the following characteristics: 1) It clusters time-evolving graphs (second-order tensor streams) and identifies change points in time. 2) It only requires buffering the current graph. Its computational cost increases linearly on the number of edges. 3) No parameter is required thanks to the compression objective. 4) The clustering assignments can be viewed as orthogonal matrices in which the (i,j)-entry is an indicator variable showing whether the i-th node belongs to the j-th cluster.

GraphScope has been successfully applied to a number of time-evolving graphs including Enron email graphs, network traffic graphs, and financial transaction graphs. In Enron email graphs, for example, the change points identified by GraphScope match well with the key events associated with Enron such as the FBI investigation and CEO resignation.

For tensor mining (high-order tensor streams), we present the Incremental Tensor Analysis (ITA) framework. Under the ITA framework, three variants are proposed with different updating algorithms, specifically, Dynamic Tensor Analysis (DTA), Streaming Tensor Analysis (STA), and Window-based Tensor Analysis (WTA). 1) They also work with general high-order tensor streams. 2) The computational and storage costs of all three variants are independent of the duration. 3) The only parameters are the sizes of the core tensors. 4) They all maintain multiple orthogonal projection matrices (one for each mode).

DTA, STA and WTA differ from each other in how they exploit the fundamental design trade-offs for speed, space and accuracy. They have been successfully applied to many different applications such as community tracking on social networks, and anomaly detection on network traffic flows. A summary of all the methods is shown in Table 5.1.

	Orthogonal	**Non-orthogonal**
1st order	**SPIRIT(2.2)**, PCA	[76]
2nd order	SVD	**CMD(3.2), GraphScope(3.3)**, [54, 47, 29]
M**th order**	**DTA,STA(4.2.4)**, [125]	**WTA(4.2.6)**, [70, 24]

Table 5.1: Algorithm classification

In conclusion, Incremental Pattern Discovery provides an unified view of several fundamental problems from the data mining perspective. A set of related tools are presented all embodying the same approach, i.e., the pursuit of efficient and effective algorithms for analyzing data streams automatically. Its success on diverse applications has confirmed the significance of all the algorithms. This incremental data comprehension and decision making is beyond the field of data mining. Incremental Pattern Discovery has far reaching impact on many different disciplines: business management, natural science and engineering. Ultimately, we need an interdisciplinary solution to tackle these problems. In this thesis, we addressed the problem from a data mining perspective, which emphasizes an algorithmic perspective and also provides a few system prototypes. In the future, all these techniques can be leveraged with a more system-oriented approach, such as deployment to de-centralized clusters and integration with existing monitoring systems.

Bibliography

[1] http://reality.media.mit.edu/download.php. 3.3, 3.3.4

[2] http://www.cs.cmu.edu/ enron/. 3.2.4, 3.3, 3.3.4

[3] http://www.informatik.uni-trier.de/ ley/db/. 3.2.4, 4.3.1

[4] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003. 2.1.4

[5] Dimitris Achlioptas and Frank McSherry. Fast computation of low rank matrix approximations. In *STOC*, 2001. 3.1.1, 3.2.3

[6] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for projected clustering of high dimensional data streams. In *VLDB*, pages 852–863, 2004. 3.1.3

[7] Charu C. Aggarwal, Jiawei Han, and Philip S. Yu. A framework for clustering evolving data streams. In *VLDB*, 2003. 2.1.4

[8] Charu C. Aggarwal and Philip S. Yu. A condensation approach to privacy preserving data mining. In *EDBT*, 2004. 2.1.5

[9] D. Agrawal and C. C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *PODS*, 2001. 2.1.5

[10] R. Agrawal and R. Srikant. Privacy preserving data mining. In *SIGMOD*, 2000. 2.1.5

[11] R. Agrawal, R. Srikant, and D. Thomas. Privacy preserving olap. In *SIGMOD*, 2005. 2.1.5

[12] Rakesh Agrawal, Roberto Bayardo, Christos Faloutsos, Jerry Kiernan, Ralf Rantzau, and Ramakrishnan Srikant. Auditing compliance with a hippocratic database. *VLDB*, 2004. 3, 3.3

[13] S. Agrawal and J. R. Haritsa. A framework for high-accuracy privacy-preserving mining. In *ICDE*, 2005. 2.1.5

[14] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. In *PODS*, 2002. 2.1.4

[15] Sitaram Asur, Srinivasan Parthasarathy, and Duygu Ucar. An event-based framework for characterizing the evolutionary behavior of interaction graphs. In *KDD*, 2007. 3.1.4

[16] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003. 2.1.4

[17] Brian Babcock and Chris Olston. Distributed Top-K Monitoring. In *SIGMOD*, 2003. 2.1.4

[18] Stephen Bay, Krishna Kumaraswamy, Markus G. Anderle, Rohit Kumar, and David M. Steier. Large scale detection of irregularities in accounting data. In *ICDM*, pages 75–86, 2006. 1, 2.4, 3.3

[19] Michael W. Berry, Shakhina A. Pulatova, and G. W. Stewart. Computing sparse reduced-rank approximations to sparse matrices. *ACM Transactions on Mathematical Software*, 31:252–269, 2005. 3.1.1

[20] J. Bethencourt, D. Song, and B. Waters. Constructions and practical applications for private stream searching. In *IEEE Symposium on Security and Privacy*, 2006. 2.1.5

[21] Rajeev Motwani Brian Babcock, Mayur Datar. Sampling from a moving window over streaming data. In *SODA*, 2002. 2.1.4

[22] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasing*. Springer, 2nd edition, 2003. 2.5.3

[23] Donald Carney, Ugur Cetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, 2003. 2.1.4

[24] J. D. Carroll and J. J. Chang. Analysis of individual differences in multidimensional scaling via an N-way generalization of 'Eckart-Young' decomposition. *Psychometrika*, 35:283–319, 1970. 5

[25] Patrick Celka and Paul Colditz. A computer-aided detection of eeg seizures in infants: A singular-spectrum approach and performance comparison. *IEEE Trans. Biomed. Eng*, 49(5), 2002. 2.4

[26] Deepayan Chakrabarti. Autopart: Parameter-free graph partitioning and outlier detection. In *PKDD*, pages 112–124, 2004. 3.3

[27] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1):2, 2006. 3.1

[28] Deepayan Chakrabarti, Ravi Kumar, and Andrew Tomkins. Evolutionary clustering. In *KDD*, 2006. 3.1.4

[29] Deepayan Chakrabarti, Spiros Papadimitriou, Dharmendra S. Modha, and Christos Faloutsos. Fully automatic cross-associations. In *KDD*, pages 79–88, 2004. 3.1.2, 3.1.3, 5

[30] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003. 2.1.4

[31] K. Chen and L. Liu. Privacy preserving data classification with rotation perturbation. In *ICDM*, 2005. 2.1.5, 2.5.1

[32] R. Chen, S. Krishnamoorthy, and H. Kargupta. Distributed Web Mining using Bayesian Networks from Multiple Data Streams. In *ICDM*, pages 281–288, 2001. 2.1.4

[33] Yizong Cheng and George M. Church. Biclustering of expression data. In *ISMB*, pages 93–10, 2000. 3.1.3

[34] D. W. Cheung, V. T. Ng, A. W. Fu, and Y. Fu. Efficient Mining of Association Rules in Distributed Databases. *TKDE*, 8:911–922, 1996. 2.1.4

[35] Yun Chi, Xiaodan Song, Dengyong Zhou, Koji Hino, and Belle L. Tseng. Evolutionary spectral clustering by incorporating temporal smoothness. In *KDD*, 2007. 3.1.4

[36] Yun Chi, Shenghuo Zhu, Xiaodan Song, Junichi Tatemura, and Belle L. Tseng. Structural and temporal analysis of the blogosphere through community factorization. In *KDD*, 2007. 3.1.4

[37] Hyuk Cho, Inderjit S. Dhillon, Yuqiang Guan, and Suvrit Sra. Minimum sum squared residue co-clustering of gene expression data. In *SDM*, 2004. 3.1.3

[38] Graham Cormode, Mayur Datar, Piotr Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). In *VLDB*, 2002. 2.1.4

[39] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991. 3.3.2

[40] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, 2003. 2.1.4

[41] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *SIGMOD*, 2003. 2.1.4

[42] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. In *SODA*, 2002. 2.1.4

[43] L. de Lathauwer. *Signal Processing Based on Multilinear Algebra*. PhD thesis, Katholieke, University of Leuven, Belgium, 1997. 1.1

[44] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990. 1.2, 2.1.1, 3.1.1

[45] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2005. 2.5.4

[46] Amol Deshpande, Carlos Guestrin, Samuel Madden, and Wei Hong. Exploiting correlated attributes in acqusitional query processing. In *ICDE*, 2005. 2.1.4

[47] Inderjit S. Dhillon, Subramanyam Mallela, and Dharmendra S. Modha. Information-theoretic co-clustering. In *KDD*, pages 89–98, 2003. 1.2, 3.1.3, 5

[48] Kostas I. Diamantaras and Sun-Yuan Kung. *Principal Component Neural Networks: Theory and Applications*. John Wiley, 1996. 2.2.1

[49] Chris Ding and Jieping Ye. Two-dimensional singular value decomposition (2dsvd) for 2d maps and images. In *SDM*, 2005. 4.1.4, 4.2.2

[50] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *SIGMOD*, 2002. 2.1.4

[51] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *KDD*, 2000. 2.1.4

[52] P. Drineas, R. Kannan, and M.W. Mahoney. Fast monte carlo algorithms for matrices i: Approximating matrix multiplication. *SIAM Journal of Computing*, 2005. 3.1.1, 3

[53] P. Drineas, R. Kannan, and M.W. Mahoney. Fast monte carlo algorithms for matrices ii: Computing a low rank approximation to a matrix. *SIAM Journal of Computing*, 2005. 3.1.1

[54] P. Drineas, R. Kannan, and M.W. Mahoney. Fast monte carlo algorithms for matrices iii: Computing a compressed approximate matrix decomposition. *SIAM Journal of Computing*, 2005. 3.1.1, 3.2, 3.2.2, 3.2.2, 6, 5

[55] Petros Drineas, Iordanis Kerenidis, and Prabhakar Raghavan. Competitive recommendation systems. In *STOC*, pages 82–90, 2002. 3.1.1

[56] Petros Drineas and Michael W. Mahoney. A randomized algorithm for a tensor-based generalization of the svd. Technical Report YALEU/DCS/TR-1327, Yale Univ., 2005. 4.1.4

[57] EPA. Epanet 2.0. `http://www.epa.gov/ORD/NRMRL/wswrd/epanet.html`. 2.5.4

[58] A. Evfimevski, J. Gehrke, and R. Srikant. Limiting privacy breaches in privacy preserving data mining. In *PODS*, 2003. 2.1.5, 2.5

[59] A. Evfimievski, R. Srikant, R. Agarwal, and J. Gehrke. Privacy preserving mining of association rules. In *SIGKDD*, 2002. 2.1.5

[60] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999. 3.1.4, 3.2

[61] Sumit Ganguly, Minos Garofalakis, and Rajeev Rastogi. Processing set expressions over continuous update streams. In *SIGMOD*, 2003. 2.1.4

[62] Venkatesh Ganti, Johannes Gehrke, and Raghu Ramakrishnan. Mining data streams under block evolution. *SIGKDD Explorations*, 3(2):1–10, 2002. 2.1.4

[63] M. Ghil, M.R. Allen, M.D. Dettinger, K. Ide, D. Kondrashov, M.E. Mann, A.W. Robertson, A. Saunders, Y. Tian, F. Varadi, and P. Yiou. Advanced spectral methods for climatic time series. *Rev. Geophys.*, 40(1), 2002. 2.4.2, 2.5.3

[64] Gene H. Golub and Charles F. Van Loan. *Matrix Computation*. Johns Hopkins, 3rd edition, 1996. 2.1.1, 2.1.3, 3.1.1, 3.2

[65] Nina Golyandina, Vladimir Nekrutkin, and Anatoly Zhigljavsky. *Analysis of Time Series Structure: SSA and Related Techniques*. CRC Press, 2001. 2.4.2

[66] S. A. Goreinov, E. E. Tyrtyshnikov, and N. L. Zamarashkin. A theory of pseudoskeleton approximations. *Journal of Linear Algebra Application*, 261, August 1997. 3.1.1

[67] Sudipto Guha, Dimitrios Gunopulos, and Nick Koudas. Correlating synchronous and asynchronous data streams. In *KDD*, 2003. 2.1.4, 2.1.5, 3.1.1

[68] Sudipto Guha, Chulyun Kim, and Kyuseok Shim. XWAVE: Optimal and approximate extended wavelets for streaming data. In *VLDB*, 2004. 2.1.4

[69] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE TKDE*, 15(3):515–528, 2003. 2.1.4

[70] Richard A. Harshman. Foundations of the PARAFAC procedure: models and conditions for an "explanatory" multi-modal factor analysis. *UCLA working papers in phonetics*, 16:1–84, 1970. 5

[71] Simon Haykin. *Adaptive Filter Theory*. Prentice Hall, 2nd ed. edition, 1992. 2.2.1, 2.2.1, 2.2.4, 4.2.4

[72] Xiaofei He, Deng Cai, and Partha Niyogi. Tensor subspace analysis. In *NIPS*, 2005. 4.1.4, 1

[73] Evan Hoke, Jimeng Sun, John D. Strunk, Gregory R. Ganger, and Christos Faloutsos. Intemon: Continuous mining of sensor data in large-scale self-* infrastructures. *ACM SIGOPS Operating Systems Review*, 40(3), 2003. 1, 1.3, 2, 2.2.4, 2.2.5, 2.4, 5

[74] Z. Huang, W. Du, and B. Chen. Deriving private information from randomized data. In *SIGMOD*, 2005. 2.1.5, 2.5.1, 2.5.2

[75] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *KDD*, 2001. 2.1.4

[76] Aapo Hyvärinen, Juha Karhunen, and Erkki Oja. *Independent Component Analysis*. Wiley-Interscience, 2001. 5

[77] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *FOCS*, 2000. 3.2

[78] INET ATS, Inc. http://www.inetats.com/. 2.5.4

[79] David S. Johnson, Shankar Krishnan, Jatin Chhugani, Subodh Kumar, and Suresh Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In *VLDB*, pages 13–23, 2004. 9

[80] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 2002. 2.1.2, 2.2.1, 2.2.1, 2.4.2, 2.5.1, 3.1.1, 3.2, 4.2.4

[81] H. Kargupta, S. Datta, Q. Wang, and K. Sivakumar. On the privacy preserving properties of random data perturbation techniques. In *ICDM*, 2003. 2.1.5, 2.5.1

[82] Eammon Keogh and T. Folias. Ucr time series data mining archive. `http://www.cs.ucr.edu/~eamonn/TSDMA/`. 2.4.4

[83] Eamonn Keogh, Stefano Lonardi, and Chotirat Ann Ratanamahatana. Towards parameter-free data mining. In *KDD*, 2004. 2.1.4, 3.1.2

[84] D. Kifer and J. Gehrke. Injecting utility into anonymized datasets. In *SIGMOD*, 2006. 2.1.5, 2.5

[85] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999. 2.1.1

[86] Tamara G. Kolda. Multilinear operators for higher-order decompositions. Technical Report SAND2006-2081, Sandia National Lab, April 2006. 4.1.2

[87] Tamara G. Kolda, Brett W. Bader, and Joseph P. Kenny. Higher-order web link analysis using multilinear algebra. In *ICDM*, 2005. 4.1.4

[88] Flip Korn, H. V. Jagadish, and Christos Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. In *SIGMOD*, pages 289–300, 1997. 3.1.1

[89] P. Kroonenberg and J. D. Leeuw. Principal component analysis of three-mode data by means of alternating least square algorithms. *Psychometrika*, 45, 1980. 1

[90] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Extracting large-scale knowledge bases from the web. In *VLDB*, 1999. 3.1.4, 3.2

[91] Jurij Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *SIGKDD*, 2005. 3, 3.1.4, 3.3

[92] Feifei Li, Jimeng Sun, Spiros Papadimitriou, George A. Mihaila, and Ioana Stanoi. Hiding in the crowd: Privacy preservation on evolving streams through correlation tracking. In *ICDE*, 2007. 1.3, 2

[93] Jessica Lin, Michail Vlachos, Eamonn Keogh, and Dimitrios Gunopulos. Iterative incremental clustering of time series. In *EDBT*, 2004. 2.1.4

[94] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *CRYTO*, 2000. 2.1.5

[95] Jinze Liu, Wei Wang, and Jiong Yang. A framework for ontology-driven subspace clustering. In *KDD*, 2004. 3.1.3

[96] K. Liu, H. Kargupta, and J. Ryan. Random Projection-Based Multiplicative Data Perturbation for Privacy Preserving Distributed Data Mining. *IEEE TKDE*, 18(1), 2006. 2.1.5, 2.5.1

[97] Charles F. Van Loan. The ubiquitous kronecker product. *Journal of Computational and Applied Mathematics*, 123:85–100, 2000. 4.1.1

[98] K. K. Loo, I. Tong, B. Kao, and D. Cheung. Online Algorithms for Mining Inter-Stream Associations From Large Sensor Networks. In *PAKDD*, 2005. 2.1.4

[99] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *ICDE*, 2006. 2.1.5

[100] Sara C. Madeira and Arlindo L. Oliveira. Biclustering algorithms for biological data analysis: a survey. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1:24–45, 2004. 3.1.3

[101] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003. 2.1.4

[102] Erkki Oja. Neural networks, principal components, and subspaces. *Intl. J. Neural Syst.*, 1:61–68, 1989. 2.2.1

[103] R. Ostrovsky and W. Skeith. Private searching on streaming data. In *CRYPTO*, 2005. 2.1.5

[104] Themistoklis Palpanas, Michail Vlachos, Eamonn Keogh, Dimitrios Gunopulos, and Wagner Truppel. Online amnesic approximation of streaming time series. In *ICDE*, 2004. 2.1.4

[105] Christos H. Papadimitriou, Hisao Tamaki, Prabhakar Raghavan, and Santosh Vempala. Latent semantic indexing: A probabilistic analysis. In *PODS*, pages 159–168, 1998. 3.1.1

[106] Spiros Papadimitriou, Anthony Brockwell, and Christos Faloutsos. Adaptive, hands-off stream mining. In *VLDB*, 2003. 2.1.4

[107] Spiros Papadimitriou, Aristides Gionis, Panayiotis Tsaparas, Risto A. Väisänen, Christos Faloutsos, and Heikki Mannila. Parameter-free spatial data mining using mdl. In *ICDM*, 2005. 3.1.2

[108] Spiros Papadimitriou, Jimeng Sun, and Christos Faloutsos. Streaming pattern discovery in multiple time-series. In *VLDB*, pages 697–708, 2005. 1, 1.3, 2, 2.1.5, 3.1.1

[109] Spiros Papadimitriou, Jimeng Sun, and Philip Yu. Local correlation tracking in time series. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2006. 1.3, 2

[110] Spiros Papadimitriou and Philip S. Yu. Optimal multi-scale patterns in time series streams. In *SIGMOD*, 2006. 2.4.2

[111] Jorma Rissanen. A universal prior for integers and estimation by minimum description length. *Annals of Statistics*, 11(2):416–431, 1983. 8

[112] Yasushi Sakurai, Spiros Papadimitriou, and Christos Faloutsos. Braid: Stream mining through group lag correlations. In *SIGMOD*, pages 599–610, 2005. 2.1.4

[113] Ralph O. Schmidt. Multiple emitter location and signal parameter estimation. *IEEE Trans. Ant. Prop.*, 34(3), 1986. 2.4.2

[114] Amnon Shashua and Anat Levin. Linear image coding for regression and classification using the tensor-rank principle. In *CVPR*, 2001. 4.1.4

[115] Jimeng Sun, Spiros Papadimitriou, and Christos Faloutsos. Online latent variable detection in sensor networks. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2005. 1, 2.2.3, 2.2.4, 2.2.5

[116] Jimeng Sun, Spiros Papadimitriou, and Christos Faloutsos. Distributed pattern discovery in multiple streams. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2006. 1, 1.3, 2

[117] Jimeng Sun, Spiros Papadimitriou, and Philip Yu. Window-based tensor analysis on high-dimensional and multi-aspect streams. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2006. 1.3

[118] Jimeng Sun, Spiros Papadimitriou, Philip S. Yu, and Christos Faloutsos. Graphscope: Parameter-free mining of large time-evolving graphs. In *KDD 2007*, 2007. 3, 1.3, 3.1.2, 3.1.3

[119] Jimeng Sun, Dacheng Tao, and Christos Faloutsos. Beyond streams and graphs: Dynamic tensor analysis. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2006. 1, 1, 1.3, 3, 3.3

[120] Jimeng Sun, Yinglian Xie, Hui Zhang, and Christos Faloutsos. Less is more: Compact matrix decomposition for large sparse graphs. In *Proceedings of the 2007 SIAM International Conference on Data Mining (SDM)*, 2007. 2, 1.3

[121] L. Sweeney. k-anonymity: a model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5), 2002. 2.1.5

[122] Dacheng Tao, Xuelong Li, Xindong Wu, and Stevephen J. Maybank. Elapsed time in human gait recognition: A new approach. In *ICASSP*, 2006. 4.1.4

[123] Dacheng Tao, Xuelong Li, Xindong Wu, and Stevephen J. Maybank. Human carrying status in visual surveillance. In *CVPR*, 2006. 4.1.4

[124] Nesime Tatbul, Ugur Cetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003. 2.1.4

[125] Ledyard R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31:279–311, 1966. 4.1.3, 5

[126] J. Vaidya and C. W. Clifton. Privacy prserving association rule mining in vertically partitionaed data. In *SIGKDD*, 2002. 2.1.5

[127] M. A. O. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *ECCV*, 2002. 4.1.4

[128] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Trans. Math. Software*, 11(1):37–57, 1985. 3.2.3

[129] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *Proc.ACM SIGKDD*, 2003. 2.1.4

[130] Yang Wang, Deepayan Chakrabarti, Chenxi Wang, and Christos Faloutsos. Epidemic spreading in real networks: An eigenvalue viewpoint. In *Symposium on Reliable Distributed Systems*, 2003. 3.1.4

[131] K. Xiao and Y. Tao. Personalized privacy preservation. In *SIGMOD*, 2006. 2.1.5

[132] Dong Xu, Shuicheng Yan, Lei Zhang, Hong-Jiang Zhang, Zhengkai Liu, and Heung-Yeung Shum. Concurrent subspaces analysis. In *CVPR*, 2005. 4.1.4, 1

[133] Bin Yang. Projection approximation subspace tracking. *IEEE Trans. Sig. Proc.*, 43(1):95–107, 1995. 2.2.1, 2.2.1, 2.4.2

[134] Jieping Ye. Generalized low rank approximations of matrices. *Machine Learning*, 61, 2004. 4.1.4

[135] Jieping Ye, Ravi Janardan, and Qi Li. Two-dimensional linear discriminant analysis. In *NIPS*, 2004. 4.1.4

[136] Byoung-Kee Yi, N.D. Sidiropoulos, Theodore Johnson, H.V. Jagadish, Christos Faloutsos, and Alexandros Biliris. Online data mining for co-evolving time sequences. In *ICDE*, 2000. 2.1.4, 2.2.2

[137] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: An efficient data clustering method for very large databases. In *SIGMOD*, 1996. 2.1.4

[138] Zhenyue Zhang, Hongyuan Zha, and Horst Simon. Low-rank approximations with sparse factors i: Basic algorithms and error analysis. *Journal of Matrix Analysis and Applications*, 23:706–727, 2002. 3.1.1

[139] Lizhuang Zhao and Mohammed J. Zaki. Tricluster: An effective algorithm for mining coherent clusters in 3d microarray data. In *SIGMOD*, 2005. 4

[140] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB*, 2002. 2.1.4