On Factorizing Million Scale Non-Negative Matrices using Compressed Structures

Sudhindra Gopal Krishna, Aditya Narasimhan, Sridhar Radhakrishnan
School of Computer Science
University of Oklahoma
Norman, USA
email:{sudhi, adinaras, sridhar}@ou.edu

Chandra N Sekharan
Department of Computing Sciences
Texas A&M University, Corpus Christi
Corpus Christi, TX, USA
email: csekharan@tamucc.edu

***Abstract*—Non-negative Matrix Factorization (NMF) is one of the algorithms with a wide range of applications, from dimensionality reduction and computer vision to text mining. The dimensions of these matrices can be of the order of several hundreds of thousands to millions, which is a raw format that would not fit in the main memory. Additionally, while performing matrix factorization on these extremely large matrices, the algorithms involving matrix operations such as transpose, multiplication, and subtraction; demand more storage for intermediate resultant matrices. In this paper, we store the matrices in compressed structures (Compressed Binary Tree CBT and Compressed Sparse Row CSR) that allow factorization without decompression. We also perform factorization CBT without using any intermediate structures by performing a virtual transpose and streaming the intermediate resultant matrices of a sequence of matrix multiplications directly into the compressed structure for every iteration. As an example, for an input matrix A of dimension $65,536 \times 65,536$ with $1.46M$ number of non-zero elements, the peak storage in any iteration of the multiplicative update factorization algorithm is $32.98GB$ when using a 2D array, $200MB$ when using CSR and $14.8MB$ for CBT. The ability to stream (add and delete) into the CBT structure without reallocation is why CBT performs the best. Furthermore, we provide a heuristic to reduce memory usage that also aids in faster convergence.**

***Keywords*—Compression; Matrix Operations; Matrix Factorization**

I. Introduction

Non-negative Matrix Factorization (NMF) can be formally defined as follows: Given a non-negative matrix $A \in \mathbb{R}_+$ of dimension $m \times n$ and an inner dimension $k > 0$, find the factor matrices if any, $W \in \mathbb{R}_+$ of dimension $m \times k$ and $H \in \mathbb{R}_+$ of dimension $k \times n$ such that:

$$A = WH$$

The factor matrices W and H are also non-negative in nature. The rank of the input matrix A gives a lower bound for the inner dimension k. This inner dimension k is referred to as the Non-negative rank of a matrix. This problem of finding the factors that satisfy condition $A = WH$ with $rank(A) = k$ has proved to be an NP-hard problem [1] [2]. The short proof of [2] tries to reduce the graph coloring problem and equates the NP-hardness of the graph chromatic number with the non-negative ranks of the input matrix, which is the smallest inner dimension for NMF.

There are various applications [3] that use NMF from computer vision, text mining/information retrieval, email, and pattern recognition to clustering in machine learning [4], face recognition [5] and data mining [6], [7]. Another application of NMF is that it can be used as a lossy compression algorithm to compress a large matrix. If the inner dimension k is small enough, then the input matrix A can be factored in $W \times H$, resulting in a lower number of elements in total. The number of elements in A to be stored will be $m \times n$, but if factorized, the number of elements to be stored will be $m \times k + k \times n$. The latter is assumed to be smaller when k is small.

The correctness of the factorization is calculated using the Frobenius norm suggested by [8] [9]. Now, the problem can be rewritten as:

$$\min_{W \geq 0, H \geq 0} \|A - WH\|_F$$

Some of the well-known sequential algorithms to solve the non-negative factorization are, *Multiplicative Update Algorithms [8] [10], Gradient Descent Algorithms and Alternating Least Squares Algorithms [11] [12]*. There are several approaches as defined in [13] that can be taken to solve this problem. In this paper, we will evaluate the Multiplicative Update Algorithm defined by Lee & Seung [8].

To solve any of the sequential algorithms mentioned above for large matrices, the algorithms require a system

 ISBN: 978-1-68558-041-4

configuration that can handle a huge number of gigabytes of data at a time. We present two state-of-the-art compressed structures (CBT [14] and CSR [15]) that are used to store these matrices and used for operations and algorithms. The input matrices and the factor matrices are all stored in either of these compressed structures. The matrices used for the analyses are both real-world and synthetically generated. We have also shown that there is a space-time trade-off between the two structures CBT and CSR. CBT taking lesser space and CSR having a shorter query time [15].

Our contribution is as follows:

- We provide a method for factorizing matrices with the least memory footprint per iteration using compressed structures.
- Sections III-A and III-B, explain various value-based matrix−matrix operations that are performed without decompression.
- We provide a matrix-transpose multiplication algorithm (Section III-C) that provides results without transposing, by streaming the result directly into compressed structures.
- In Section III-D, we explain how we sequence 3 or more matrix multiplication operations, without storing any intermediate matrices.
- Proposed a heuristic (Section III-E) that eliminates unnecessary rows/columns that leads to lower memory usage and faster convergence.

II. Related Work

The foundation for the Non-negative matrix factorization was laid by Lee and Seung [8] in 1999, opening the opportunity to hundreds of research journals. Before Lee and Seung, few other notable contributions were made in the area of NMF, but none came close to the fame of Lee and Seung. Paatero and Tapper, 1994 [16], produced the work on positive matrix factorization. Lee and Seung cite the work of Paatero and Tapper in their work. Articles have shown the significance of Paatero's work prior to Lee and Seung but have gone unnoticed.

Since Lee and Seung's NMF was one of the first ones to be popular, it became a baseline for many research. Several researchers have proven that the multiplicative update algorithm proposed by Lee and Seung [8], is slower to converge, which means that it takes many more iterations to complete compared to the gradient descent method and the alternating least squares. Each implementation required a total of 12 matrix operations, of which six require $O(n^3)$ matrix-matrix multiplication, and the rest require $O(n^2)$ matrix-matrix element-wise operations.

To overcome this issue, other researchers, such as Gonzalez and Zhang in 2005 [10], proposed an alteration to the multiplicative update, but it ended up having the same convergence issue. Another researcher named Lin [17] in 2007 proposed a modification that ended with earlier convergence but at the cost of more operations per iteration.

Theoretically performing 12 matrix operations on a matrix is time- and space-consuming; performing the same operation on larger matrices would require a great deal of memory. For example, a $65,536 \times 65,536$ requires about 32 GB of storage in its raw format. To overcome this, in this paper, we use our novel CBT [18], which works well with binary matrices and the bit-packing algorithm proposed in [15] to store integer values. We also propose to store the matrix in CSR [19], a common data structure for storing matrices.

To perform factorization or any operation on large sparse matrices, one must efficiently store the matrices so that the entire data can be loaded onto the main memory in one go. Given a matrix of size n rows and m column, the total number of possible elements in the matrix is the size of the matrix itself, which $m \times n$, therefore, the cost of storing a matrix in raw format would require $(m \times n) \times 64$ number of bits, where 64 is the number of bits required to store a number. But in a sparse matrix, this number tends to be very small, where the number of non-zero elements is extremely less compared to the number of zeros.

Therefore, the sparsity of a matrix is defined as the ratio of the number of non-zero elements to the number of all possible elements that can be in the matrix.

$$Sparsity = \frac{nnz}{m \times n},$$

where nnz is the number of non-zero elements in the matrices, n is the number of rows and m is the number of columns.

This type of behavior in the matrices are found in the real-world, such as social networks, biological network, topological network, and so on. The cost of storing zeros in such cases becomes expensive and redundant to an extent, as they do not contribute to the analysis.

Therefore, to store large sparse matrices, in this paper, we are using existing structures such as CSR [15] [19], and CBT [20].

III. Matrix Factorization

There are several approaches that can be taken to factorize a given matrix. To mention a few of the popular ones, multiplicative update, gradient descent,

 ISBN: 978-1-68558-041-4

and alternating least squares [8] [11]. Here, we take the updated rules provided by Lee and Seung [8].

$$H \leftarrow H \frac{(W^T V)}{(W^T W H)}, \quad W \leftarrow W \frac{(V H^T)}{(W H H^T)}$$

```
begin
    W = rand(m, k)
    H = rand(k, n)
    for i : maxiter do
        H ← H .* (W^T A) ./ (W^T W H + 10^-9)
        W ← W .* (A H^T) ./ (W H H^T + 10^-9)
```

Figure 1. Multiplicative Update algorithm for NMF using the Frobenius norm as a cost function

Algorithm 1, shows the workings of how to factorize the given large matrix using the multiplicative update algorithm. The algorithm involves a series of operations to obtain the desired result of W and H. To clarify the various matrix element-wise operations, the $.*$ operation represents an element-wise multiplication, and a $./$ represents an element-wise division and matrix-based operations such as matrix-matrix multiplication. So, we continue this section by providing the algorithms for the various operations that are the building blocks of 1.

A. Matrix-Matrix Multiplication

One of the first and most important operations to be performed during the factorization process is matrix-matrix multiplication. The work on matrix-matrix multiplication has been published in [21], which explains the working of how two matrices stored in either of the data structures CSR and CBT are multiplied without the need for an intermediate data structure.

B. Element-Wise Matrix Operation

The multiplicative update algorithm consists of several element-wise matrix operations. The operations involved in the algorithm are element-wise multiplication $.*$, element-wise division $./$, and element-wise subtraction $-$ to find the Frobenius norm. Apart from these three, we can also extend the algorithm for element-wise addition $+$.

Algorithm 2, explains the working of the element-wise matrix operation. The operation to be performed, "Op," is specified as input. The algorithm first checks if the dimensions of the two matrices are equal and, if not, throws an error. It then loops through each row of the matrices, and for each row, it checks if the size of the row is zero in either matrix. If it is, it appends a zero to the corresponding row of the resultant matrix C. If both matrices have a row of size zero, it also appends a zero to the corresponding row of C. If only one matrix has a row of size zero, it copies the elements from the non-zero row and appends them to the corresponding row of C. If neither matrix has a row of size zero, the algorithm performs the specified operation on each element of the corresponding rows of A and B and appends the result to the corresponding row of C. Finally, the algorithm returns the resultant matrix C.

```
Input: Matrix A, Matrix B, Operation Op
Output: resultan_matrix C
if A.rowSize != B.rowSize or A.colSize != B.colSize then
    Error: Matrix dimensions should be the same for both the matrices
for i in numberofRows do
    if A[i].rows == 0 and B[i].rows == 0 then
        C[i] = 0
        continue to the next row
    else if A[i] == 0 then
        C[i] = B[i]
        continue to the next row
    else if B[i] == 0 then
        C[i] = A[i]
        continue to the next row
    for aIndex in A[i] do
        for bIndex in B[i] do
            C[i][j] = A[i][j] "Op" B[i][j]
            Where "Op" = "+ or - or .* or ./"
return C
```

Figure 2. Element-wise matrix Addition, Subtraction, Multiplication, and Division

C. Matrix Transpose

Another important operation required to perform matrix factorization is to transpose a given matrix. There are two ways we have handled this situation in this paper, one way is to transpose the given matrix and store it as another matrix that occupies extra space, and another way to do it is to incorporate transpose during the required operation.

 ISBN: 978-1-68558-041-4

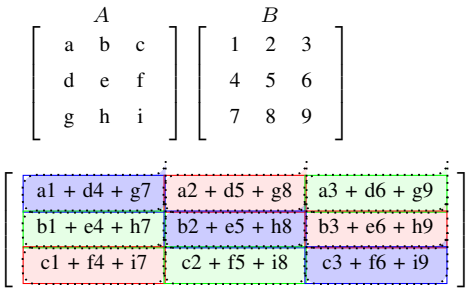

Figure 3. The working of $A^T \times B$, by storing the result in a pattern to eliminate the need to transpose the actual matrix.

The multiplicative update algorithm contains matrix-matrix multiplication where either one of the matrices needs to be transposed. A way to achieve this operation would be to transpose the required matrix and use the algorithm mentioned in [21], but this requires additional memory; here the additional memory is the transposed matrix. To avoid this issue, we perform an in-place transpose multiplication. This can be achieved by accessing the matrices with a different access pattern.

$$A \times B^T = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 5 & 0 & 2 & 3 \\ 3 & 0 & 0 & 5 \\ 0 & 0 & 2 & 4 \\ 0 & 1 & 2 & 0 \end{bmatrix} \end{array} \times \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 2 & 4 & 3 & 1 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix} \end{array} \tag{1}$$

$$\Rightarrow \begin{array}{l} r_0(A) \rightarrow \\ \\ c_0(B) \rightarrow \end{array} \begin{pmatrix} 5 \\ \times \\ 2 \end{pmatrix} + \begin{pmatrix} 5 \\ \times \\ 4 \end{pmatrix} + \begin{pmatrix} 5 \\ \times \\ 3 \end{pmatrix} + \begin{pmatrix} 5 \\ \times \\ 1 \end{pmatrix}$$

$$\Rightarrow c_0[C] = \{10\ 20\ 15\ 5\} \tag{2}$$

Equation 2 shows an example of $A \times B^T$, where the partial resultant of column $c_0[C]$, is obtained after multiplying the first row $r_0[A]$ of A, and virtually transposed the first column of B, in this case, it is still $r_0[B]$.

Figure 3 shows the multiplication of $A^T \times B$ by virtually transposing A. Here, the colors along the diagonal show the order in which the resultant is obtained. Multiplying $r_0[A]$ with all rows of B, we obtain the main diagonal; continuing the process to the farther rows of A, we move the resultant to the upper triangle and wrap it around to the lower triangle, as shown in **red**, and **green**.

D. Sequence of matrix multiplications

Revisiting algorithm in [21], where algorithms take two matrices as input and multiply them to produce the resultant matrix. However, data structures such as our novel versions of CBT and CSR are amenable to multiplying multiple matrices without storing the intermediate resultant matrix.

Algorithm 5 shows multiple matrix-matrix multiplication. Line 5 takes the output of line 3, the intermediate resultant row, and computes the resultant row on the third matrix. This process can be repeated through any number of input matrices. Therefore, this can be scaled to k as the number of matrices.

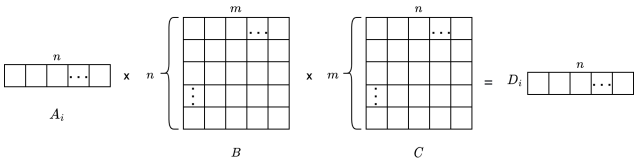

Figure 4. The working of sequential matrix multiplication.

Figure 4 shows the pictorial representation of sequential multiplications of multiple matrices. A row of matrix A, A_i is multiplied by matrix B using the partial sum algorithm to obtain the intermediate resultant row Z_i, then Z_i is multiplied with the next matrix C to obtain the final resultant row D_i.

E. Heuristic for faster convergence

One of the drawbacks of the multiplicative update approach is the convergence time and the iterations it

Input: Matrix A, Matrix B, Matrix C
Output: Resultant_Matrix D
1 **for** *row = 0 to numberOfRows* **do**
2 aRow = getRow(A, row)
3 tempArray = copmutePartialSum(aRow,B)
4 /* Call Algorithm in Sec III-A */
5 tempArray = computePartialSum(tempArray, C)
6 /* Call Algorithm in Sec III-A */
7 **for** *nnzElements in tempArray* **do**
8 D[row].streamEdge(nnzIndex)
9 $D.values$.bitPack(nnzValues)
10 /* If we are performing the matrix multiplication in CSR, then the number of non-zero elements in the resultant data for each row should be stored in C */
11 return D

Figure 5. Matrix-Matrix Multiplication in Sequence

takes to find an optimal solution. One of the ways to make the algorithm faster would be to reduce the number of non-zero values in the input matrix. If we are given a threshold number of index positions per row that can be made zero, we can come up with a heuristic approach to make specific values zero so that our compression is more efficient. One way to approach this is to remove the noise in the data; that is, we remove the data that do not contribute to the overall solution. This may lead to more loss, but the threshold will dictate the metric of the percentage of loss added to this already lossy factorization approach if we had not taken the heuristic approach. This will be a heuristic approach and will not be optimal. But it will lead to reduced resource utilization. Space is reduced in the already compressed structure and time to query the smaller CBT structure.

IV. Experimental Results

This section evaluates matrix factorization on various matrices. For this experiment, we considered the variety of matrices with variable sparsity.

To factorize the matrices, we must first choose low-rank dense random W and H matrices. Choosing a low-ranking matrix leads to the formation of a smaller resultant matrix, which in turn consumes less space. Finding an optimal rank for factorization is a hard problem, as the algorithm has to go through the process of finding the number of orthogonal rows in the matrix. It is also more likely that the larger the inner dimension of the factors that we compute (W and H), the sparser these matrices will be, in which case CBT outperforms CSR even in terms of the storage of dense matrices. Therefore, in this paper, we perform a brute-force analysis to obtain a minimal rank that would satisfy the criteria to reproduce the almost original matrix when W and H are multiplied.

Table I shows the overall result of the computation performed in this paper. The first set of columns in the table explains the basic details of the input, matrix dimensions in the first column, the number of non-zero elements in the second, matrix size when represented by using the $2-D$ matrix in the third, and the compressed sizes in the fourth and fifth respects. In the next part of the table, we present the inner rank of the factored matrices, followed by the result of $W \times H$ for both CBT and CSR, and the amount of memory required to process factorization at each iteration by CBT, CSR, and $2-D$ representation of the matrix.

In the results, one can notice that the memory required by the $2-D$ matrix is the highest. Still, the majority of the size is just the A matrix. Since the resultants can be streamed into a matrix in O(1) (constant), the extra memory used is very minimal. Still, as the inner rank increases, memory usage will increase accordingly.

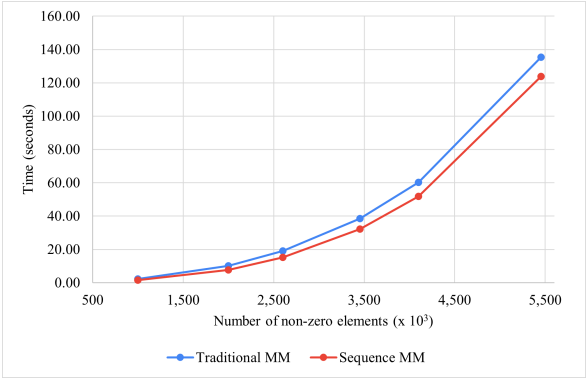

Figure 6. Comparison between the time taken to multiply three matrices in traditional two steps and uses our novel sequence multiplication in a single step for a Million-by-Million matrix.

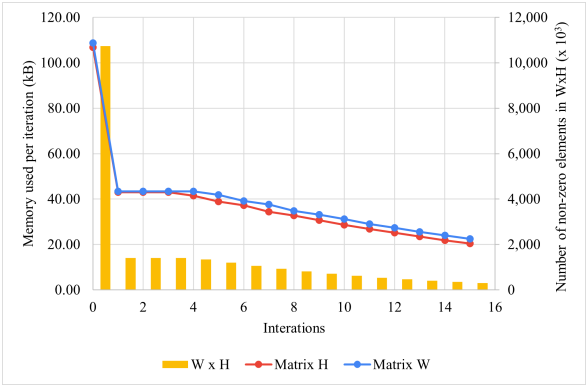

Figure 7. The evolution of W and H during the factorization for Matrix of size (21,504× 21,504, 1.36M nnz elements)

However, when considering the two compressed structures, the proportion of memory consumed by CSR is much greater compared to the memory consumed by CBT [15]. This is due to the inability of CSR's to stream (add/delete), as the arrays need to resize, whereas CBT's ability to perform in-line operations, the advantage of one such operation is shown in the Figure 6, the figure compares the time taken to multiply three matrices in traditional two steps and uses our novel sequence multiplication in a single step for a Million-by-Million matrix of various levels of sparsity ranging from 1M to 5.4M elements. This memory usage will have a significant impact for a very large matrix, as shown in [21].

Figure 7 shows the decrease in the memory required to store W and H as the iteration progresses, with the number of non-zero elements represented in the bars.

 ISBN: 978-1-68558-041-4

TABLE I. THE FACTORIZATION RESULT USING CBT AND CSR AND THE MEMORY REQUIRED TO PROCESS THE FACTORS.

						W × H		Avg Mem/Iter		
Matrix A	**NNZ**	**Matrix Size**	**CBT**	**CSR**	**Inner Rank**	***CBT***	***CSR***	***Matrix***	***CBT***	***CSR***
2688×2688	23,089	55.12 MB	217.36 KB	216.23 KB	448	216.58 KB	216.51 KB	73.5 MB	0.54 KB	0.67 MB
5376×5376	57,752	220.5 MB	547.53 KB	546.68 KB	255	513.87 KB	526.46 KB	241.41 MB	0.29 KB	30 MB
21504×21504	1,385,198	3.44 GB	12.7 MB	12.98 MB	512	12.65 MB	12.95 MB	3.6 GB	13.1 MB	150 MB
43008×43008	998,531	13.78 GB	9.45 MB	9.53 MB	670	9.1 MB	9.98 MB	14.21 GB	9.92 MB	87 MB
65536×65536	1,460,048	32 GB	14.23 MB	14.05 MB	665	13.45 MB	14.12 MB	32.64 GB	14.80 MB	200 MB

All experiments were run on an Intel(R) Xeon(R) W-2295 CPU @ 3.00GHz (16 Cores) with 64 GB of RAM, and the programs were written in GNU C/C++.

V. CONCLUSION AND FUTURE WORK

This paper shows that the given million-scale matrix can be factorized directly on the compressed structure. We also show that the intermediate result obtained in the matrix factorization process can be eliminated using sequential matrix operations. In this paper, we also introduced element-wise matrix multiplication, division, subtraction, addition, and sequential multiple matrix multiplications on top of the existing work of matrix multiplication. We have also shown that traversing through the matrix in the pattern can avoid an explicit transpose operation during the matrix factorization. We also provide the heuristic relationship between inner rank and the sparsity of the factor matrices, and we have also shown in the results that the lower the rank, the smaller the factors W and H. In the future, we would expand the computation to the Alternating Least Squares and Gradient Descent approach to factorize matrices. Our compression algorithms mentioned in this paper natively support binary matrices. Hence, we would also expand our work toward Binary Matrix Factorization.

REFERENCES

[1] S. A. Vavasis, "On the complexity of nonnegative matrix factorization," *SIAM Journal on Optimization*, vol. 20, no. 3, pp. 1364–1377, 2010.

[2] Y. Shitov, "The nonnegative rank of a matrix: Hard problems, easy solutions," *SIAM Review*, vol. 59, no. 4, pp. 794–800, 2017.

[3] N. Gillis, "The why and how of nonnegative matrix factorization," *Connections*, vol. 12, no. 2, pp. 257–291, 2014.

[4] W. Xu, X. Liu, and Y. Gong, "Document clustering based on non-negative matrix factorization," in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pp. 267–273, 2003.

[5] D. Guillamet and J. Vitria, "Non-negative matrix factorization for face recognition," in *Catalonian Conference on Artificial Intelligence*, pp. 336–344, Springer, 2002.

[6] M. W. Berry and M. Browne, "Email surveillance using non-negative matrix factorization," *Computational & Mathematical Organization Theory*, vol. 11, no. 3, pp. 249–264, 2005.

[7] S. Zhang, W. Wang, J. Ford, and F. Makedon, "Learning from incomplete ratings using non-negative matrix factorization," in *Proceedings of the 2006 SIAM international conference on data mining*, pp. 549–553, SIAM, 2006.

[8] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Advances in neural information processing systems*, pp. 556–562, 2001.

[9] N. Guan, D. Tao, Z. Luo, and J. Shawe-Taylor, "Mahnmf: Manhattan non-negative matrix factorization," *arXiv preprint arXiv:1207.3438*, 2012.

[10] E. F. Gonzalez and Y. Zhang, "Accelerating the lee-seung algorithm for nonnegative matrix factorization." http://www.caam.rice.edu/tech_reports/2005/TR05-02.ps, 2005.

[11] M. W. Berry, M. Browne, A. N. Langville, V. P. Pauca, and R. J. Plemmons, "Algorithms and applications for approximate nonnegative matrix factorization," *Computational statistics & data analysis*, vol. 52, no. 1, pp. 155–173, 2007.

[12] H. Kim and H. Park, "Nonnegative matrix factorization based on alternating nonnegativity constrained least squares and active set method," *SIAM journal on matrix analysis and applications*, vol. 30, no. 2, pp. 713–730, 2008.

[13] Y.-X. Wang and Y.-J. Zhang, "Nonnegative matrix factorization: A comprehensive review," *IEEE Transactions on knowledge and data engineering*, vol. 25, no. 6, pp. 1336–1353, 2012.

[14] M. Nelson, S. Radhakrishnan, and C. N. Sekharan, "Billion-scale matrix compression and multiplication with implications in data mining," in *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pp. 395–402, IEEE, 2019.

[15] S. Gopal Krishna, M. Nelson, S. Radhakrishnan, A. Chatterjee, and C. Sekharan, "On Compressing Time-Evolving Networks," in *ALLDATA 2021, The Seventh International Conference on Big Data, Small Data, Linked Data and Open Data*, pp. 43–48, 2021.

[16] P. Paatero and U. Tapper, "Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values," *Environmetrics*, vol. 5, no. 2, pp. 111–126, 1994.

[17] C.-J. Lin, "On the convergence of multiplicative update algorithms for nonnegative matrix factorization," *IEEE Transactions on Neural Networks*, vol. 18, no. 6, pp. 1589–1596, 2007.

[18] M. Nelson, S. Radhakrishnan, and C. Sekharan, "Queryable Compression on Time-Evolving Social Networks with Streaming," in *Big Data (Big Data), 2018 IEEE International Conference on*, IEEE BigData '18, pp. 146–151, IEEE Computer Society, 2018.

[19] R. A. Snay, "Reducing the profile of sparse symmetric matrices," *Bulletin Géodésique*, vol. 50, no. 4, pp. 341–352, 1976.

[20] M. Nelson, S. Radhakrishnan, A. Chatterjee, and C. Sekharan, "Queryable Compression on Streaming Social Networks," in *Big Data (Big Data), 2017 IEEE International Conference on*, IEEE BigData '17, pp. 988–993, IEEE Computer Society, 2017.

[21] S. G. Krishna, A. Narasimhan, S. Radhakrishnan, and R. Veras, "On large-scale matrix-matrix multiplication on compressed structures," in *2021 IEEE International Conference on Big Data (Big Data)*, pp. 2976–2985, 2021.

 ISBN: 978-1-68558-041-4