

Existing and New Algorithms for Non-negative Matrix Factorization

By Wenguo Liu & Jianliang Yi

[Abstract]

Three non-negative matrix factorization algorithms proposed by Lee & Seung^[1] and Paatero^[2], and the implementation of these algorithms are discussed in this report. The advantages and disadvantages of these three algorithms are compared based on initial matrix selection, convergent speed, result quality, and easiness to implement.

A new non-negative matrix factorization algorithm using rank-deficient QR decomposition is presented in this report. Computation samples manifest the efficiency of this new algorithm. The comparison of this new algorithm with the other three algorithms is also given in the report.

0. Introduction

Non-negative matrix factorization (NMF)^[1]: given a non-negative $m \times n$ matrix \mathbf{X} , find non-negative matrix factors \mathbf{G} ($m \times p$) and \mathbf{F} ($p \times n$) such that $\mathbf{X} \approx \mathbf{GF}$, where p is a smaller number compared to m and n . The reasonable value of p depends on the property of the input matrix \mathbf{X} . For some matrices, if p is too small, no reasonable solution exists.

Non-negative matrix factorization is a very important matrix factorization in Linear Algebra. It is widely used in environmental science, spectroscopy and chemometrics^[2, 3, 6, 7] where the matrix elements have clear physical meanings.

Two easy to implement algorithms are presented in [1]. These two algorithms are based on two different multiplicative update rules. Convergence proofs are also provided in the paper^[1], and at least locally optimal solutions are guaranteed to be found.

Positive Matrix Factorization (PMF) algorithm^[2, 7] was first proposed by Dr. Paatero. It is time-efficient compared with commonly used Alternative Least Squares algorithm (ALS)^[8] though with some similarities. PMF can be considered as a generalization of the ALS algorithm^[2].

In this paper, NMF algorithms^[1] and PMF algorithm^[2] are implemented in MATLAB. Their basic implementation details are illustrated in the discussion. The comparisons are carried out based on the convergent rate, initial matrix selection, result

quality, and easiness to implement. Also, a new PMF algorithm using rank-deficient QR decomposition with column pivoting is discussed and implemented.

1. Non-negative Matrix Factorization with Multiplicative Update ^[1]

(1) Cost Function of NMF Algorithms

Since the non-negative factorization is an approximation factorization $\mathbf{X} \approx \mathbf{GF}$, we need to define the cost function to qualify this approximation.

One natural way is to use the Euclidean distance between two matrices to evaluate the approximation. Assume matrix \mathbf{A} and \mathbf{B} , the Euclidean distance between them is defined as:

$$\|\mathbf{A} - \mathbf{B}\|^2 = \sum_{i,j} (\mathbf{A}_{ij} - \mathbf{B}_{ij})^2. \quad (1-1)$$

It's lower bounded by 0 and equals to 0 if and only if $\mathbf{A} = \mathbf{B}$ ^[1].

Another useful cost function is the divergence of \mathbf{A} and \mathbf{B} . It's defined as:

$$D(\mathbf{A} \parallel \mathbf{B}) = \sum_{i,j} (\mathbf{A}_{ij} \log \frac{\mathbf{A}_{ij}}{\mathbf{B}_{ij}} - \mathbf{A}_{ij} + \mathbf{B}_{ij}). \quad (1-2)$$

We call it "divergence" instead of "distance" because it's not symmetric between \mathbf{A} and \mathbf{B} . It's also lower bounded by 0 and equals to 0 if and only if $\mathbf{A} = \mathbf{B}$ ^[1].

(2) NMF Algorithm 1 by Lee & Seung (Euclidean Update) ^[1]: Minimize $\|\mathbf{X} - \mathbf{GF}\|^2$ with respect to \mathbf{G} and \mathbf{F} , subject to the constraints $\mathbf{G}, \mathbf{F} \geq 0$.

1. Initialize \mathbf{G} and \mathbf{F} to be two random non-negative matrices;
2. Keep updating \mathbf{G} and \mathbf{F} until $\|\mathbf{X} - \mathbf{GF}\|^2$ converges. The multiplicative update rules are as the following:

$$\mathbf{F}_{am} = \mathbf{F}_{am} \frac{(\mathbf{G}^T \mathbf{X})_{am}}{(\mathbf{G}^T \mathbf{GF})_{am}}, \quad \mathbf{G}_{ia} = \mathbf{G}_{ia} \frac{(\mathbf{XF}^T)_{ia}}{(\mathbf{G}^T \mathbf{FF}^T)_{ia}}. \quad (1-3)$$

During the above updates, we should update \mathbf{G} and \mathbf{F} "simultaneously". We shouldn't update the whole matrix \mathbf{F} first followed by updating the matrix \mathbf{G} . Instead, after updating one row of \mathbf{F} , we need to update the corresponding column of \mathbf{G} . So actually we update \mathbf{F} and \mathbf{G} alternatively. When update a row of \mathbf{F} or a column of \mathbf{G} , we don't need to calculate out the whole matrices $\mathbf{G}^T \mathbf{X}$, $\mathbf{G}^T \mathbf{GF}$, \mathbf{XF}^T and $\mathbf{G}^T \mathbf{FF}^T$ as suggested by the appearance of the rules, since we only need one row (or column) of these matrices during one update.

(3) Computation Sample of Euclidean Update

Computation samples verify that our implementation is successful. For example, for a matrix

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix},$$

the **output** of our program when running at a tolerance of 1.0e-3 (Initial value $\mathbf{G} = 0.5 * \text{ones}(4, 2)$; $\mathbf{F} = 0.5 * \text{ones}(2, 3)$;) is given below:

Converge at 126 step, tolerance = 0.001000, euc_val = 0.000976

$\mathbf{G} =$

$$\begin{bmatrix} 0.1475 & 1.5118 \\ 0.6416 & 1.1179 \\ 1.1391 & 0.6933 \\ 1.6271 & 0.3548 \end{bmatrix}$$

$\mathbf{F} =$

$$\begin{bmatrix} 6.1231 & 6.6129 & 7.1000 \\ 0.0644 & 0.6761 & 1.2929 \end{bmatrix}$$

$\mathbf{X} - \mathbf{G} * \mathbf{F} =$

$$\begin{bmatrix} -0.0003 & 0.0026 & -0.0017 \\ -0.0007 & 0.0012 & -0.0009 \\ -0.0194 & -0.0014 & 0.0161 \\ 0.0139 & -0.0001 & -0.0116 \end{bmatrix}$$

(4) NMF Algorithm 2 by Lee & Seung (Divergence Update)^[1]: Minimize $D(\mathbf{X} \parallel \mathbf{GF})$ with respect to \mathbf{G} and \mathbf{F} , subject to the constraints $\mathbf{G}, \mathbf{F} \geq 0$.

1. Initialize \mathbf{G} and \mathbf{F} to be two random non-negative matrices;
2. Keep updating \mathbf{G} and \mathbf{F} until $D(\mathbf{X} \parallel \mathbf{GF})$ converges. The multiplicative update rules are as the following:

$$\mathbf{F}_{am} = \mathbf{F}_{am} \frac{\sum_i \mathbf{G}_{ia} \mathbf{X}_{im} / (\mathbf{GF})_{im}}{\sum_k \mathbf{G}_{ka}}, \quad \mathbf{G}_{ia} = \mathbf{G}_{ia} \frac{\sum_m \mathbf{F}_{am} \mathbf{X}_{im} / (\mathbf{GF})_{im}}{\sum_v \mathbf{F}_{av}}. \quad (1-4)$$

During the above updates, we should update \mathbf{G} and \mathbf{F} “simultaneously”. We shouldn’t update the whole matrix \mathbf{F} first followed by updating the matrix \mathbf{G} . Instead, after updating one row of \mathbf{F} , we need to update the corresponding column of \mathbf{G} . So actually we update \mathbf{F} and \mathbf{G} alternatively.

(5) Computation Sample of Divergence Update

Computation samples verify that our implementation is successful. For example, for a matrix

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix},$$

the **output** of our program when running at a tolerance of $1.0e-3$ (Initial value $\mathbf{G} = 0.5 * \text{ones}(4, 2)$; $\mathbf{F} = 0.5 * \text{ones}(2, 3)$;) is given below:

Converge at 80 step, tolerance = 0.001000, div_val = 0.000965

$\mathbf{G} =$

0.1588 1.2308

0.6708 0.9005

1.1861 0.5435

1.6867 0.3038

$\mathbf{F} =$

5.9011 6.3785 6.8525

0.0509 0.8006 1.5546

$\mathbf{X-G*F} =$

0.0000 0.0015 -0.0018

-0.0044 0.0003 0.0034

-0.0270 -0.0008 0.0273

0.0315 -0.0016 -0.0300

2. The Iterative Positive Matrix Factorization (PMF) Algorithm ^[2]

(1) Basic Equation

The basic equations for this algorithm are:

$$(\mathbf{G} + \mathbf{g})(\mathbf{F} + \mathbf{f}) = \mathbf{X}, \quad (2-1)$$

$\mathbf{R} = \mathbf{X} - \mathbf{GF}$ is the residual of factorization, \mathbf{f} is a matrix with the same size as \mathbf{F} matrix, and \mathbf{g} is matrix with the same size as \mathbf{G} matrix. We need to compute the unknowns \mathbf{g} and \mathbf{f} at each computation step, and then we can move on to the next step with updated matrices $\mathbf{G} = \mathbf{G} + \mathbf{g}$ and $\mathbf{F} = \mathbf{F} + \mathbf{f}$. Initial values of \mathbf{G} and \mathbf{F} matrices are provided, currently we tried different initial matrices such as *ones* and random matrices.

For the above nonlinear equation, Newton-Raphson method is a good choice, provided we are lucky enough that we do not encounter non-positive-definite matrix ^[2]. Basically, we cannot ensure that non-positive-definite matrix will not occur, although the probability is small, therefore, using Newton_Raphson method to solve this nonlinear equation system will sometimes fail. Our current MATLAB code actually implemented using the second alternative (Named Gauss-Newton procedure), which is explained in the following text.

Equation (2-1) can be simplified by omitting the 2nd order term \mathbf{gf} ^[2], which results in a linear equation system

$$\mathbf{Gf} + \mathbf{Fg} = \mathbf{R}. \quad (2-2)$$

If matrix \mathbf{X} is m by n , \mathbf{G} is m by p , and \mathbf{F} is p by n , reiterating the above equation results in a linear equation system with $(m+n)p$ unknowns in \mathbf{f} and \mathbf{g} ,

$$\sum_{h=1}^p \mathbf{G}_{ih} \mathbf{f}_{hj} + \sum_{h=1}^p \mathbf{g}_{ih} \mathbf{F}_{hj} = \mathbf{R}_{ij}, \quad (2-3)$$

where \mathbf{G}_{ih} represents the i^{th} row, h^{th} column element of matrix \mathbf{G} . The above linear equation system can be written in matrix form,

$$\mathbf{A}\Delta\mathbf{x} = \mathit{vect}(\mathbf{R}) \quad (2-4)$$

where $\Delta\mathbf{x} = [\mathbf{f}_{11} \ \mathbf{f}_{12} \ \dots \ \mathbf{f}_{pn} \ \mathbf{g}_{11} \ \mathbf{g}_{12} \ \dots \ \mathbf{g}_{mp}]^T$, matrix \mathbf{A} consists of many zeros and the remaining part are elements of \mathbf{G} and \mathbf{F} matrices, and $\mathit{vect}(\mathbf{R})$ is a vector for the matrix \mathbf{R} expressed in vector form with row dominant order. For example, if

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}, \quad (2-5)$$

then

$$\mathit{vect}(\mathbf{R}) = \begin{bmatrix} r_{11} \\ r_{12} \\ r_{21} \\ r_{22} \end{bmatrix}. \quad (2-6)$$

To check whether the algorithm has got convergent results, the error function is evaluated at each step ^[2],

$$Q(\mathbf{G}, \mathbf{F}) = \sum_{i=1}^m \sum_{j=1}^n (\mathbf{R}_{ij} / \hat{\sigma}_{ij})^2, \quad (2-7)$$

where $\hat{\sigma}$ is the standard deviation matrix of the input matrix \mathbf{X} .

The standard deviation matrix $\hat{\sigma}$ is related to the physical meaning of the problem, and makes our current method a weighted least squares problem (Define weights $w_{ij} = 1/\hat{\sigma}_{ij}^2$). In practice, you may put less weight on some part of the data in the \mathbf{X} matrix if you know in advance that the data might have some errors because of experimental errors or other reasons. Currently in our code it is evaluated to be all-one matrix (equal weights for all elements of \mathbf{X}) simply because we are only concerned with the algorithm itself, not its physical meaning at this time.

If the above function Q gets a small quantity below our prefixed error tolerance, then this algorithm has converged.

(2) Regularization Procedure

Several factors determine that we cannot have a unique solution to the above equation system.

(a) Rotation

If we have a positive matrix factorization of the form $\mathbf{X} = \mathbf{GF}$ ($\mathbf{X} \geq 0, \mathbf{G} \geq 0, \mathbf{F} \geq 0$) (where $\mathbf{G} \geq 0$ means every element of matrix \mathbf{G} is greater than or equal to 0), then if we can find a square matrix \mathbf{T} , s.t.,

$$\begin{aligned} \mathbf{X} &= \mathbf{GTT}^{-1}\mathbf{F}, \\ \mathbf{GT} &\geq 0, \mathbf{T}^{-1}\mathbf{F} \geq 0. \end{aligned} \quad (2-8)$$

Then we have found another solution to the positive matrix factorization problem. Therefore, rotation makes the solution not unique.

(b) Scaling

If we have a positive matrix factorization of the form $\mathbf{X} = \mathbf{GF}$ ($\mathbf{X} \geq 0, \mathbf{G} \geq 0, \mathbf{F} \geq 0$), then

$$\mathbf{X} = (\mathbf{Gc})(\frac{1}{c}\mathbf{F}), (\text{scalerc} > 0) \quad (2-9)$$

is also a solution to the problem. Therefore, scaling also makes the solution not unique.

No unique solution exists implies that the matrix \mathbf{A} in Eq. (2-4) is singular. This is obvious from the following example:

Based on Eq. (2-3), for a 2 by 2 matrix \mathbf{X} , the equations used to solve unknowns $\mathbf{f}_{11}, \mathbf{f}_{12}, \mathbf{g}_{11}, \mathbf{g}_{21}$ are

$$\begin{aligned}
\mathbf{G}_{11}\mathbf{f}_{11} + \mathbf{g}_{11}\mathbf{F}_{11} &= \mathbf{R}_{11} \\
\mathbf{G}_{11}\mathbf{f}_{12} + \mathbf{g}_{11}\mathbf{F}_{12} &= \mathbf{R}_{12} \\
\mathbf{G}_{21}\mathbf{f}_{11} + \mathbf{g}_{21}\mathbf{F}_{11} &= \mathbf{R}_{21} \\
\mathbf{G}_{21}\mathbf{f}_{12} + \mathbf{g}_{21}\mathbf{F}_{12} &= \mathbf{R}_{22}.
\end{aligned} \tag{2-10}$$

If expressed in matrix form, it is

$$\begin{bmatrix} \mathbf{G}_{11} & 0 & \mathbf{F}_{11} & 0 \\ 0 & \mathbf{G}_{11} & \mathbf{F}_{12} & 0 \\ \mathbf{G}_{21} & 0 & 0 & \mathbf{F}_{11} \\ 0 & \mathbf{G}_{21} & 0 & \mathbf{F}_{12} \end{bmatrix} \begin{bmatrix} \mathbf{f}_{11} \\ \mathbf{f}_{12} \\ \mathbf{f}_{21} \\ \mathbf{f}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{11} \\ \mathbf{R}_{12} \\ \mathbf{R}_{21} \\ \mathbf{R}_{22} \end{bmatrix}. \tag{2-11}$$

The matrix on the left hand side is always singular, no matter what are the values \mathbf{G} and \mathbf{F} . This can be shown via calculating the determinant of the left hand side matrix.

Therefore, a regularization procedure should be carried out to eliminate the singularity from the equation system [2, 3]. Regularization terms are added to the Q function to be minimized at each iteration step,

$$Q(\mathbf{G}, \mathbf{F}) = \sum_{i=1}^m \sum_{j=1}^n (\mathbf{R}_{ij} / \delta_{ij})^2 + \mathbf{g} \sum_{i=1}^m \sum_{h=1}^p \mathbf{G}_{ih}^2 + \mathbf{d} \sum_{h=1}^p \sum_{j=1}^n \mathbf{F}_{hj}^2, \tag{2-12}$$

where \mathbf{g} and \mathbf{d} are small quantities for enforcing regularization. Eq. (2-4) used to compute the unknowns are expressed in normal equation format, and also revised for the new target function,

$$(\mathbf{A}^T \mathbf{W} \mathbf{A} + \ddot{\mathbf{E}}) \Delta \mathbf{x} = \mathbf{A}^T \mathbf{W} \mathbf{R} - \ddot{\mathbf{E}} \mathbf{x}_i, \tag{2-13}$$

where \mathbf{W} matrix is a diagonal matrix of the weights $w_{ij} = 1 / \delta_{ij}^2$, . For example, for 2 by 2 matrix case,

$$\mathbf{W} = \begin{bmatrix} w_{11} & 0 & 0 & 0 \\ 0 & w_{12} & 0 & 0 \\ 0 & 0 & w_{21} & 0 \\ 0 & 0 & 0 & w_{22} \end{bmatrix}. \tag{2-14}$$

$\ddot{\mathbf{E}}$ is a diagonal matrix with \mathbf{g} or \mathbf{d} on the diagonal positions. $\mathbf{x}_i = [\mathbf{F}_{11} \ \mathbf{F}_{12} \ \dots \ \mathbf{F}_{pn} \ \mathbf{G}_{11} \ \mathbf{G}_{12} \ \dots \ \mathbf{G}_{mp}]^T$ represents the solution of matrices \mathbf{G} and \mathbf{F} at current iteration step. $\Delta \mathbf{x} = [\mathbf{f}_{11} \ \mathbf{f}_{12} \ \dots \ \mathbf{f}_{pn} \ \mathbf{g}_{11} \ \mathbf{g}_{12} \ \dots \ \mathbf{g}_{mp}]^T$ represents the increment matrix for current step \mathbf{G} and \mathbf{F} matrices. After $\Delta \mathbf{x}$ is solved, \mathbf{G} and \mathbf{F} matrices can be formed via the increment relationship $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta \mathbf{x}$.

The detailed development of Eq. (2-13) is shown explicitly in [3]. The development proceeds in a way that each term of the target function Q is linearized around current iteration step solution of \mathbf{G} and \mathbf{F} matrices, and then minimizing Q (through the method of matrix differentiation) leads to the normal equation desired.

(3) Positive Constraints

Positive constraints can be enforced by adding penalty functions to the Q function^[2, 3]. In our case, logarithmic terms are added. The revised target function Q is

$$Q(\mathbf{G}, \mathbf{F}) = \sum_{i=1}^m \sum_{j=1}^n (\mathbf{R}_{ij} / \acute{o}_{ij})^2 - \mathbf{a} \sum_{i=1}^m \sum_{h=1}^p \log \mathbf{G}_{ih} - \mathbf{b} \sum_{h=1}^p \sum_{j=1}^n \log \mathbf{F}_{hj} + \mathbf{g} \sum_{i=1}^m \sum_{h=1}^p \mathbf{G}_{ih}^2 + \mathbf{d} \sum_{h=1}^p \sum_{j=1}^n \mathbf{F}_{hj}^2, \quad (2-15)$$

where \mathbf{a} and \mathbf{b} are small quantities used to control the strength of the penalty terms. It is also called logarithmic barrier method. It's obvious from Eq. (2-15) that when the elements of \mathbf{G} or \mathbf{F} get close to zero, large penalty of Q will come out. Therefore, this method can keep the computed \mathbf{G} and \mathbf{F} from getting close to zero.

To use the logarithmic barrier method in our model, logarithmic terms in Q must be approximated by quadratic functions^[3], for example, for the logarithmic function

$$f(y) = \log_{10}(y), \quad (2-16)$$

the corresponding quadratic function is

$$q(y) = c_1(y - c_2)^2, \quad (2-17)$$

where

$$c_1 = \frac{1}{4 \ln(10) y_0^2 \ln(y_0)}, c_2 = y_0 - 2y_0 \ln(y_0) \quad (2-18)$$

can be derived by letting the value of the 1st derivative of $f(y)$ and $q(y)$ be equal at current solution point y_0 .

Eq. (2-13) can be modified to be consistent with the new target function Q ,

$$(\mathbf{A}^T \mathbf{W} \mathbf{A} + \ddot{\mathbf{E}} + \ddot{\mathbf{E}}_p) \Delta \mathbf{x} = \mathbf{A}^T \mathbf{W} \mathbf{R} - \ddot{\mathbf{E}} \mathbf{x}_0 + \ddot{\mathbf{E}}_p (\mathbf{x}^* - \mathbf{x}_i), \quad (2-19)$$

where $\ddot{\mathbf{E}}_p$ is a diagonal matrix with $c_1 \mathbf{a}$ or $c_1 \mathbf{b}$ on the diagonal positions, here the coefficient c_1 is defined in Eq. (2-18). $\mathbf{x}^* = [c_2^1 \quad c_2^2 \quad \dots \quad c_2^{(m+n)p}]^T$ is a vector with $c_2^i (1 \leq i \leq (m+n)p)$ defined in Eq. (2-18), the superscript i represents that c_2 is the coefficient corresponding to the i^{th} variable in \mathbf{x} (\mathbf{x} is a $(m+n)p \times 1$ matrix). $\Delta \mathbf{x} = [\mathbf{f}_{11} \quad \mathbf{f}_{12} \quad \dots \quad \mathbf{f}_{pn} \quad \mathbf{g}_{11} \quad \mathbf{g}_{12} \quad \dots \quad \mathbf{g}_{mp}]^T$ represents the increment matrix for current step \mathbf{G} and \mathbf{F} matrices. After $\Delta \mathbf{x}$ is solved, \mathbf{G} and \mathbf{F} matrices can be formed via the increment relationship $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta \mathbf{x}$.

However, this approximation of logarithmic penalty functions as quadratic functions cannot prevent occasionally computed negative values appear in the solution^[3]. In such cases, MATLAB *lsqnonneg* function is used to change the solution back to positive. MATLAB command $\mathbf{X} = \text{lsqnonneg}(\mathbf{A}, \mathbf{b})$ function return the vector \mathbf{X} that minimizes $\text{norm}(\mathbf{A} * \mathbf{X} - \mathbf{b}, 2)$, subject to $\mathbf{X} \geq 0$.

Thus, we have derived an equation system that can be used to compute the increment vector \mathbf{x} with the knowledge of current iteration solution \mathbf{x}_0 .

(4) Implementation Pseudocode

- a. Set the initial value of \mathbf{G} and \mathbf{F} matrices;
- b. Form matrix \mathbf{A} in Eq. (2-4);
- c. Form matrix $\mathbf{B} = \mathbf{A}^T \mathbf{W} \mathbf{A}$, where \mathbf{W} is the weight matrix, it is current hard-wired as identity matrix in our code because we are not concerned with the physical meaning of the matrix \mathbf{A} ;
- d. Form matrix $\mathbf{R} = \mathbf{X} - \mathbf{G} \mathbf{F}$;
- e. Form matrix $\mathbf{C} = \mathbf{A}^T \mathbf{W} \mathbf{R}$;
- f. Form matrix $\ddot{\mathbf{E}}$;
- g. Form matrix $-\ddot{\mathbf{E}} \mathbf{x}_i$, where \mathbf{x}_i is a vector consists of the current iteration step \mathbf{G} and \mathbf{F} matrices element values;
- h. Form matrix $\ddot{\mathbf{E}}_p$;
- i. Approximate logarithmic barrier functions as quadratic functions using Eq. (2-18);
- j. Form vector $\ddot{\mathbf{E}}_p(\mathbf{x}^* - \mathbf{x}_i)$;
- k. Form the equations system $\mathbf{LHS} \Delta \mathbf{x} = \mathbf{RHS}$ of Eq. (2-19) by summing the matrices computed above together;
- l. Solve the equation in j^{th} step to get unknowns \mathbf{x} , i.e., \mathbf{g} and \mathbf{f} ;
- m. Form matrix \mathbf{F} and \mathbf{G} for next step, with $\mathbf{F} = \mathbf{F} + \mathbf{f}$ and $\mathbf{G} = \mathbf{G} + \mathbf{g}$;
- n. If \mathbf{F} or \mathbf{G} are not positive (this situation occasionally happens because approximation of logarithmic functions with quadratic functions cannot enforce non-negative condition strictly), use MATLAB *lsqnonneg* function to adjust it back to positive;
- o. Evaluate error Q based on \mathbf{F} and \mathbf{G} matrices, if this value is smaller than our prefixed error tolerance, then the algorithm converged and program stops. Otherwise, goto step b to begin another iteration step.

(5) Computation Samples

Computation samples verify that our implementation is successful. For example, for a matrix

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix},$$

the **output** of our program when running at a tolerance of 1.0e-3 (Initial value $\mathbf{G} = 0.5 * \text{ones}(4, 2)$; $\mathbf{F} = 0.5 * \text{ones}(2, 3)$;) is given below:

Converge at 21 step, tolerance = 0.001000, v_Qbar = 0.000126

$\mathbf{G} =$

1.1943 0.1309
1.3492 1.2415

	1.5042	2.3521	
	1.6591	3.4627	
$F =$			
	0.5520	1.3983	2.2447
	2.6237	2.5063	2.3890
$X-G*F=$			
	-0.0026	0.0019	0.0065
	-0.0020	0.0017	0.0054
	-0.0014	0.0015	0.0044
	-0.0008	0.0013	0.0033

3. A New PMF Algorithm with Rank-Deficient QR Decomposition

Notice that we do not have to add the regularization terms and logarithmic penalty terms into the target function in Eq. (2-7), because:

- Rank-Deficient least squares problems can be solved via QR decomposition with column pivoting ^[5];
- MATLAB *lsqnonneg* function can enforce the non-negativity of the matrices.

Therefore, in this section, we use MATLAB *lsqnonneg* function and rank-deficient QR decomposition to solve positive matrix factorization problem.

(1) Solving rank-deficient least squares problems using QR with pivoting

The QR decomposition of a rank-deficient matrix \mathbf{A} can be written as:

$$\mathbf{A}\mathbf{P} = \mathbf{Q}\mathbf{R} = [\mathbf{Q}_1 \quad \mathbf{Q}_2] \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \quad (3-1)$$

where \mathbf{A} is a m by n matrix with rank r , \mathbf{R}_1 is an invertible r by r matrix, \mathbf{R}_2 is a r by $(n-r)$ matrix, \mathbf{Q}_1 is a m by r matrix, \mathbf{Q}_2 is a m by $(m-r)$ matrix, \mathbf{P} is a permutation matrix introduced by pivoting.

The least square problem $\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2$ has a solution ^[5] of

$$\mathbf{x} = \mathbf{P} \begin{bmatrix} \mathbf{R}_1^{-1}(\mathbf{Q}_1^T \mathbf{b}) \\ \mathbf{0} \end{bmatrix} \quad (3-2)$$

The detailed development of the above equation is in [5]. The permutation matrix \mathbf{P} is the only difference between Eq. (3-2) and the expression of \mathbf{x} in [5].

(2) Solution Space Partition ^[2]

Realized that $\mathbf{x} = [\mathbf{f}_{11} \quad \mathbf{f}_{12} \quad \dots \quad \mathbf{f}_{pn} \quad \mathbf{g}_{11} \quad \mathbf{g}_{12} \quad \dots \quad \mathbf{g}_{mp}]^T$ in Eq. (2-4) might consist of too many variables and cause the size of matrix \mathbf{A} to be too large, in this algorithm, we actually break the solution procedure into two steps.

Let $\mathbf{x}_1 = [\mathbf{f}_{11} \quad \mathbf{f}_{12} \quad \dots \quad \mathbf{f}_{pn} \quad \mathbf{a}]^T$, and solve the equations

$$\mathbf{G}\mathbf{f} + \mathbf{a}\mathbf{G}\mathbf{F} = \mathbf{R} \quad (3-3)$$

where \mathbf{g} and \mathbf{F} are known matrices from the previous iteration step. We have only $(np+1)$ unknowns in this equation system, which will make the matrix size smaller.

Eq. (3-3) can be written as

$$\mathbf{M}_1 \mathbf{x}_1 = \mathit{vect}(\mathbf{R}) \quad (3-4)$$

where \mathbf{M}_1 matrix consists of many zeros and the remaining part is determined by elements of \mathbf{F} , \mathbf{G} , and \mathbf{g} matrices, and $\mathit{vect}(\mathbf{R})$ is a vector for the matrix \mathbf{R} expressed in vector form with row dominant order.

The next part is to let $\mathbf{x}_2 = [\mathbf{g}_{11} \quad \mathbf{g}_{12} \quad \dots \quad \mathbf{g}_{pn} \quad \mathbf{a}]^T$, and solve the equations

$$\mathbf{aGf} + \mathbf{gF} = \mathbf{R} \quad (3-5)$$

where \mathbf{G} and \mathbf{f} are known matrices from the previous iteration step. We have $(mp+1)$ unknowns in this equation system.

Eq. (3-5) can be written as

$$\mathbf{M}_2 \mathbf{x}_2 = \mathit{vect}(\mathbf{R}) \quad (3-6)$$

where \mathbf{M}_2 matrix consists of many zeros and the remaining part is determined by elements of \mathbf{G} , \mathbf{F} , and \mathbf{f} matrices, and $\mathit{vect}(\mathbf{R})$ is a vector for the matrix \mathbf{R} expressed in vector form with row dominant order.

(3) Algorithm Implementation Pseudocode

- a. Set initial value of \mathbf{G} , \mathbf{F} , \mathbf{f} matrices;
- b. Form matrix \mathbf{M}_1 in Eq. (3-4);
- c. Form matrix \mathbf{R} as $\mathbf{R} = \mathbf{X} - \mathbf{GF}$;
- d. Solve Eq. (3-4) using QR with pivoting as described above;
- e. Form new matrices $\mathbf{G} = \mathbf{G} + \mathbf{a g}$, $\mathbf{F} = \mathbf{F} + \mathbf{f}$;
- f. Form matrix \mathbf{M}_2 in Eq. (3-6);
- g. Form matrix \mathbf{R} as $\mathbf{R} = \mathbf{X} - \mathbf{GF}$;
- h. Solve Eq. (3-6) using QR with pivoting as described above;
- i. Form new matrices $\mathbf{G} = \mathbf{G} + \mathbf{g}$, $\mathbf{F} = \mathbf{F} + \mathbf{a f}$;
- p. If \mathbf{F} or \mathbf{G} are not positive (because we did not enforce non-negativity using penalty function), use MATLAB *lsqnonneg* function to adjust it back to positive;
- j. Evaluate error Q based on \mathbf{F} and \mathbf{G} matrices, if this value is smaller than our prefixed error tolerance, then the algorithm has converged and program stops. Otherwise, goto step b to begin another iteration step.

(4) Computation Samples

Computation samples verify that our implementation is successful. For example, for a matrix

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix},$$

the **output** of our program when running at a tolerance of $1.0e-3$ (Initial value $\mathbf{G} = 0.5 * \mathit{ones}(4, 2)$; $\mathbf{F} = 0.5 * \mathit{ones}(2, 3)$;) is given below:

Converge at 4 step, tolerance = 0.001000, v_Qbar = 0.000856
 $G =$
 2.3098 0.6767
 9.0492 0.5085
 15.7886 0.3404

$$\begin{array}{l} 22.5280 \quad 0.1722 \\ F = \\ 0.4435 \quad 0.4782 \quad 0.5121 \\ 0 \quad 1.3233 \quad 2.6850 \\ X-G*F = \\ -0.0244 \quad -0.0000 \quad -0.0000 \\ -0.0132 \quad -0.0000 \quad -0.0000 \\ -0.0020 \quad 0 \quad -0.0000 \\ 0.0092 \quad 0 \quad -0.0000 \end{array}$$

4. Computation, Comparison and Conclusion

We described the details of the four algorithms above and implemented them in MATLAB. These four algorithms are: Euclidean Update, Divergence Update, PMF, and PMF_DefQR.

Here are some aspects tested for persuasive results:

(1) Convergent Speed

All these algorithms start with some initial matrices \mathbf{G} and \mathbf{F} , and keep updating them until convergence. Here convergence means that the cost functions (or the evaluation functions) are satisfied. It's obvious that the number of iterations is a very important value that can indicate the speed of an algorithm.

(2) Initial Value Selection

All these algorithms have the same problem: local minima problem. With some initial values, \mathbf{G} and \mathbf{F} may reach some local minima, which cannot satisfy the cost functions (or the evaluation functions). So the algorithm cannot converge with these initial values.

Although all these algorithms have this problem, some algorithms are more sensitive to initial values, which means with some random initial values, the possibility that they cannot converge is large. While other algorithms are much better since for most of initial values, these algorithms can converge.

If an algorithm is not sensitive to the initial values, this means that this algorithm is easy to use in a wide area. So this is also a very important criterion we need to consider.

(3) Result Quality

The result quality is absolutely what we concern. Here result quality refers to the closeness of \mathbf{X} and \mathbf{GF} . Since different algorithms use different cost functions (or evaluation functions), it's difficult to compare the result quality directly based on these functions. As an alternative we use the 2-norm of the residual matrix ($\text{norm}(\mathbf{X} - \mathbf{GF}, 2)$) to evaluate the result quality of all these algorithms.

(4) Easiness of Implementation

It is obvious from our implementation efforts that NMF algorithms ^[1] (Euclidean Update and Divergence Update) are the easiest to implement. PMF algorithm ^[2] is the most difficult to implement, actually, most of our time is spent in this part because of many hidden materials in the paper and the complexity of the algorithm. Our new PMF_DefQR algorithm is relatively easier to implement compared with PMF algorithm ^[2], and with better convergent speed.

The data in the table is the statistical information in the following conditions: each algorithm is run on a matrix \mathbf{X} for 100 times. Each time we use two random matrices \mathbf{G} and \mathbf{F} as the initial matrices. Then we record the convergent rate, average convergence steps and average norm of the residual matrix $\mathbf{X} - \mathbf{GF}$.

	Euclidean	Divergence	PMF	PMF_DefQR
Convergent rate	58%	93%	100%	94%
Average convergence steps	223.7241	211.6774	18.8400	3.8723
Average norm of $\mathbf{X} - \mathbf{GF}$	0.0308	0.0598	0.0205	0.0095

One thing that we need to mention here before we further discuss the advantages and disadvantages of all four algorithms is the convergent speed. From the above table we can see that generally Euclidean Update and Divergence Update use hundreds of steps while PMF and PMF_DefQR algorithms only use several steps. But the difference of running time between them is not so large as suggested by the convergent steps. That's because in the Euclidean and Divergence algorithm, every step is just a multiplicative update and consumes less time than PMF and PMF_DefQR algorithms, which need to solve linear equations or do QR decomposition.

Advantages and Disadvantages of Euclidean Update

Euclidean Update algorithm converges very slowly. From the above table we can clearly see that in our test case, it averagely takes more than 200 steps if it can converge.

Euclidean Update algorithm also has the disadvantage that local minima may occur, in which we cannot get a global optimal solution. The convergent rate of this algorithm is quite low in our test, only 58%, which means in many cases, it converges to local minima instead of the result that satisfies our requirement. So if we want to use this algorithm, we need to be very careful to select the initial value in order to get a successful non-negative factorization.

But if it can converge, the result quality is not bad comparing with other algorithms. Also the algorithm is very easy to implement.

Advantages and Disadvantages of Divergence Update

Divergence Update algorithm converges very slowly. From the above table we can see that in our test case, it averagely takes more then 200 steps if it can converge.

Divergence Update algorithm also has the disadvantage that local minima may occur, in which we cannot get a global optimal solution. But from our test we can see that the convergent rate (93%) is much better than the Euclidean Update algorithm. This means that this algorithm is easier to use than the Euclidean algorithm.

And this algorithm is also very easy to implement comparing with PMF and PMF_DefQR algorithm.

But from the above table we can see that the result quality of this algorithm is not as good as other algorithms. In order to get better results, we need to make the cost function stricter, which generally means to reduce the convergent rate or increase the convergence steps.

Advantages and Disadvantages of PMF algorithm

PMF algorithm ^[2, 3] usually can converge faster compared with NMF algorithms ^[1]. Also, PMF algorithm has an excellent merit in that it can deal with the weighted positive matrix factorization problem, which is very important in environmental science, spectroscopy and chemometrics ^[2, 3, 6, 7] where the matrix elements have clear physical meanings.

Just like NMF algorithms, PMF algorithm also has the disadvantage that local minima may occur, in which we cannot get a global optimal solution. But from the above table we can see that all the cases we test converge, which means that generally in PMF algorithm, the selection of initial value is not that strict comparing with NMF algorithms. This is a great advantage of this algorithm.

Also from the above table we can see that the result quality (which is indicated by the average norm) of the PMF is a little bit better than the NMF algorithms.

But PMF algorithm needs much more time to implement comparing with NMF algorithms, which is a disadvantage.

Advantages and Disadvantages of PMF algorithm

It is awesome that our new PMF_DefQR algorithm converges so fast, even compared with the PMF algorithm. Also we can see that the result quality of our algorithm is very good, also better than the PMF algorithm.

Our new algorithm has another merit that it can be implemented very easily compared with Paatero's PMF algorithm.

However, just like other algorithms, this new algorithm also has the disadvantage that local minima may occur, in which we cannot get a global optimal solution. But from the above table we can see that the convergent rate of our algorithm is 94%, only slightly less than the PMF algorithm.

5. Future Work

One of the most important problems in the use of PMF algorithms to solve practical problems is rotation. As illustrated earlier in this paper, the existence of rotation results in non-unique solution to the PMF problem. Through the use of regularization terms in the

target function Q , singularity of the equation system can be removed. Most of the positive matrix factorizations have rotational domains (“rotational domain” means the set of all possible rotations) ^[4]. However, our PMF algorithm actually attempts to compute a solution in the middle of the rotational domain ^[4]. Computing the rotational domain of a positive matrix factorization is an interesting research area.

It is well known that Singular Value Decomposition (SVD) method can provide a better and more accurate way to solve rank-deficient least squares problems compared with QR with pivoting method. Therefore, it might be a good choice to use SVD instead of QR with pivoting in the future version of this new PMF algorithm.

References

- [1] D. D. Lee, H. S. Seung. Algorithms for non-negative matrix factorization.
- [2] P. Paatero, U. Tapper. Least squares formulation of robust non-negative factor analysis. *Chemometr. Intell. Lab.* 37 (1997), 23-35.
- [3] P. Paatero. A weighted non-negative least squares algorithm for three-way 'PARAFAC' factor analysis. *Chemometr. Intell. Lab.* 38 (1997), 223-242.
- [4] P. Paatero, P. K. Hopke, etc. Understanding and controlling rotations in factor analytic models. *Chemometr. Intell. Lab.* 60 (2002), 253-264.
- [5] J. W. Demmel. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia. 1997.
- [6] S. Junnto, P. Paatero. Analysis of daily precipitation data by positive matrix factorization. *Environmetrics*, 5 (1994), 127-144.
- [7] P. Paatero, U. Tapper. Positive matrix factorization: a non-negative factor model with optimal utilization of error estimates of data values. *Environmetrics*, 5 (1994), 111-126.
- [8] C. L. Lawson, R. J. Hanson. *Solving least squares problems*. Prentice-Hall, Englewood Cliffs, NJ, 1974.