Exploring and Implementing Non – Negative Matrix Factorization using Particle Swarm Optimization

Report submitted in partial fulfillment of the requirement for the degree of B.Tech

In

Information Technology

Under the Supervision of

Prof. O.P Vyas

By

Vijender Singh Aswal

IIT2011151

To

Indian Institute of Information Technology,

Allahabad

May 2015

Declaration

This is to certify that Report entitled "Exploring and Implementing Non – Negative Matrix Factorization using MapReduce and Hadoop" which is submitted by me in partial fulfillment of the requirement for the award of degree B.Tech in Information Technology to Indian Institute of Information Technology, Allahabad comprises only my original work and due acknowledgement has been made in the text to all other material used.

Date:

Vijender Singh Aswal

IIT2011151

Certificate

I do hereby recommend that the project work entitled "Exploring and Implementing Non – Negative Matrix Factorization using MapReduce and Hadoop" is being submitted in fulfilment of Mini Project for the academic session Jan 2015 - Jun 2015. The declaration made by the candidates is true to the best of my knowledge.

Place:
IIIT Allahabad
Jhalwa Campus
Date: 6nd May, 2015

Project Mentor
Prof. O.P. Vyas

Acknowledgements

I would like to convey our deepest gratitude to Prof. O.P. Vyas, who guided us through this project. His keen awe-inspiring personality, superb guidance, and constant encouragement is the motivating force behind this project work. Also, I would like to thank Mr. Bharat Singh for her enormous help and support throughout the project duration

Table of Contents

1. Introduction 7
2. Literature Survey.....26
3. Problem definition and Objectives..... 28
4. Proposed Approach 29
5. Hardware & Software requirements.....31
6. Activity Time Chart 31
7. Results.....32
8. Conclusion and future scope.....35
9. References 36
10. Appendix 37

1. Introduction

In every single second in this modern era, tons of data are being generated. Think of the number of online people writing their blogs, designing their homepages and sharing their experiences through many other digital supports: videos, photos, etc. Think also of the data generated when decoding genes of living creatures and the data acquired from the outer space or even from our own planet, etc. Data only become useful when having been processed. In front of this fast-growing amount of data, there are several approaches for data processing: applying classical methods, designing more powerful computing structures such as distributed computing, multicore processors, supercomputers, etc. But the growing amount and complexity of accumulated data seems to outweigh the growth of computing power which is, at the present time, roughly doubling every year. One very popular approach is called model reduction which tries to reduce the complexity while keeping the essentials of the problem (or data).

Besides, different types of data require different models to capture the insight of the data. Using the right model saves a lot of time. Of course, a model believed to be right will stand until a better model is found. An ideal model may not exist.

1.1 Motivation

Reducing the number of features (dimensionality) is important in statistical learning. For many data sets with a large number of features and a limited number of observations, such as bioinformatics data, usually many features are not useful for producing a desired learning result and only increase computational time. Reducing features can also save storage and computation time and increase comprehensibility. By feature selection, we are able to find out the subset of features which can most accurately represent our data. Feature selection is also useful as part of the data analysis process, as it shows which features are important for prediction, and how these features are related. This will lead to better understanding and explanation of our data set. The ultimate objective of any machine learning algorithm is to correctly group/classify objects (documents, pictures, emails etc.) into correct categories.

Feature selection techniques provide three main benefits when constructing predictive models:

- improved model interpretability,
- shorter training times,
- enhanced generalisation by reducing overfitting.

1.2 Non-negativity of data

Non-negative data are pervasive. Consider the following three important applications, each of which give rise to nonnegative data matrices.

• In document collections, documents are stored as vectors. Each element of a document vector is a count (possibly weighted) of the number of times a corresponding term appears in that document. Stacking document vectors one after the other creates a nonnegative term-by-document matrix that represents the entire document collection numerically.

• Similarly, in image collections, each image is represented by a vector, and each element of the vector corresponds to a pixel. The intensity and color of the pixel is given by a nonnegative number, thereby creating a nonnegative pixel-by-image matrix.

• For item sets or recommendation systems, the information for a purchase history of customers or ratings on a subset of items is stored in a non-negative sparse matrix.

Three common goals in mining information from such matrices are:

(1) to automatically cluster similar items into groups,

(2) to retrieve items most similar to a user's query, and

(3) identify interpretable critical dimensions within the collection.

1.2 Low Rank Approximations

Applications, such as text processing, data mining, and image processing, store pertinent information in a huge matrix. This matrix A is large, sparse, and often

times nonnegative. In the last few decades, researchers realized that the data matrix could be replaced with a related matrix, of much lower rank.

The low rank approximation to the data matrix A brought several advantages. The rank-k approximation, denoted $\mathbf{A_k}$, sometimes required less storage than **A**. But most importantly, the low rank matrix seemed to give a much cleaner, more efficient representation of the relationship between data elements. The low rank approximation identified the most essential components of the data by ignoring inessential components attributed to noise, pollution, or inconsistencies.

1.3 Non-Negative Matrix Factorization[1]

Non-negative matrix factorization (NMF) is a low-rank approximation technique for multivariate data decomposition. Given an n × m real nonnegative matrix **V** and a positive integer r < min(n, m), it aims to find a factorization of **V** into an n × r real matrix **W** and an r × m real matrix H such that:

$$\mathbf{V} \approx \mathbf{WH} \qquad \text{Eq. 1}$$

The multivariate data to decompose is stacked into **V**, whose columns represent the different observations, and whose rows represent the different variables.

Each column vj of **V** can be expressed as:

$$\mathbf{v}_j \approx \mathbf{W}\mathbf{h}_j = \sum_i h_{ij}\mathbf{w}_i \qquad \text{Eq. 2}$$

Where h_j and w_i are respectively the $j-$ th column of **H** and the $i-$ th column of **W**. The columns of **W** then form a basis and each column of **H** is the decomposition or encoding of the corresponding column of **V** into this basis. The rank r of the factorization is generally chosen such that (n + m)r << nm, so **WH** can be thought of as a compression or reduction of **V**.

Given a data matrix **V**, the optimal choice of matrices **W** and **H** are defined to be those nonnegative matrices that minimize the reconstruction error between **V** and **WH**. Various error functions have been proposed (Paatero and Tapper, 1994; Lee and Seung, 2001), perhaps the most widely used is the squared error (euclidean distance) function.

1.4 Why Non-Negative Matrix Factorization?

Latent factor models are an alternative approach to collaborative filtering that tries to explain the ratings by characterizing both items and users on, say, 20 to 100 factors inferred from the ratings patterns. In a sense, such factors comprise a computerized alternative to the aforementioned human created song genes. For movies, the discovered factors might measure obvious dimensions such as comedy versus drama, amount of action, or orientation to children; less well-defined dimensions such as depth of character development or quirkiness; or completely uninterpretable dimensions. For users, each factor measures how much the user likes movies that score high on the corresponding movie factor.

$$E(\mathbf{W},\mathbf{H}) = \|\mathbf{V}-\mathbf{WH}\|^2 = \sum_{i,j}(V_{ij}-(\mathbf{WH})_{ij})^2 \qquad \text{Eq. 3}$$

Figure 2 illustrates this idea for a simplified example in two dimensions. Consider two hypothetical dimensions characterized as female- versus male-oriented and serious versus escapist. The figure shows where several well-known movies and a few fictitious users might fall on these two dimensions. For this model, a user's predicted rating for a movie, relative to the movie's average rating, would equal the dot product of the movie's and user's locations on the graph. For example, we would expect Gus to love Dumb and Dumber, to hate The Color Purple, and to rate Braveheart about average. Note that some movies—for example, Ocean's 11—and users—for example, Dave—would be characterized as fairly neutral on these two dimensions.

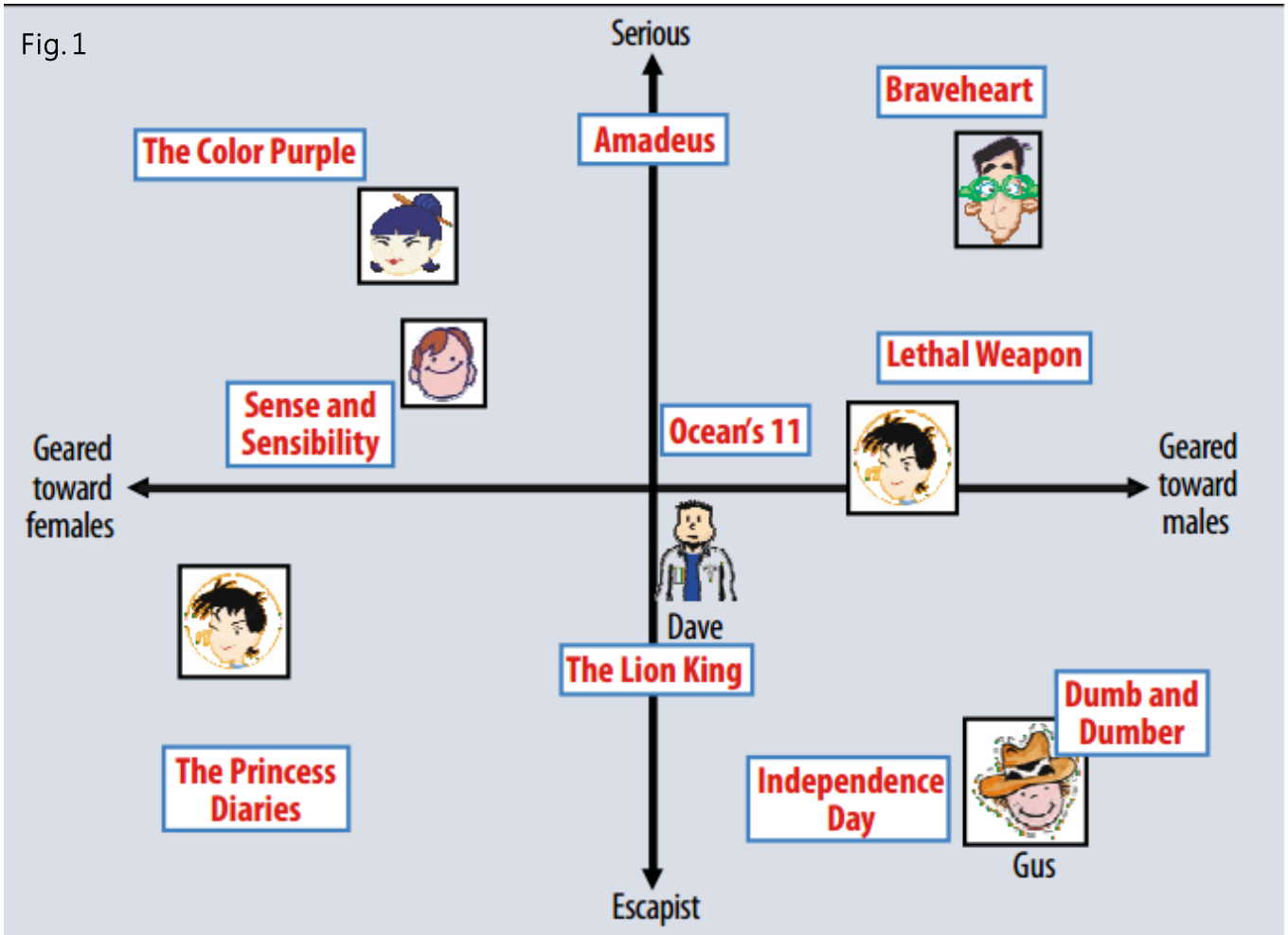


Figure 2. A simplified illustration of the latent factor approach, which characterizes both users and movies using two axes—male versus female and serious versus escapist.

Some of the most successful realizations of latent factor models are based on matrix factorization. In its basic form, matrix factorization characterizes both items and users by vectors of factors inferred from item rating patterns. High correspondence between item and user factors leads to a recommendation. These methods have become popular in recent years by combining good scalability with predictive accuracy. In addition, they offer much flexibility for modeling various real-life situations.

Recommender systems rely on different types of input data, which are often placed in a matrix with one dimension representing users and the other dimension representing items of interest. The most convenient data is high-quality explicit feedback, which includes explicit input by users regarding their interest in products. For example, Netflix collects star ratings for movies, and TiVo users indicate their preferences for TV shows by pressing thumbs-up and thumbs-down buttons. We refer to explicit user feedback as ratings. Usually, explicit feedback comprises a sparse matrix, since any single user is likely to have rated only a small percentage of possible items.

One strength of matrix factorization is that it allows incorporation of additional information. When explicit feedback is not available, recommender systems can infer user preferences using implicit feedback, which indirectly reflects opinion by observing user behavior including purchase history, browsing history, search patterns, or even mouse movements. Implicit feedback usually denotes the presence or absence of an event, so it is typically represented by a densely filled matrix.

1.5 A basic matrix factorization model[2]

Matrix factorization models map both users and items to a joint latent factor space of dimensionality f, such that user-item interactions are modeled as inner products in that space. Accordingly, each item i is associated with a vector $q_i \in f$, and each user u is associated with a vector $p_u \in f$.

For a given item i, the elements of q_i measure the extent to which the item possesses those factors, positive or negative.

For a given user u, the elements of p_u measure the extent of interest the user has in items that are high on the corresponding factors, again, positive or negative.

The resulting dot product, $q_i^T p_u$, captures the interaction between user u and item i—the user's overall interest in the item's characteristics. This approximates user u's rating of item i, which is denoted by r_{ui}, leading to the estimate:

$$\hat{r}_{ui} = q_i^T p_u$$ Eq. 4

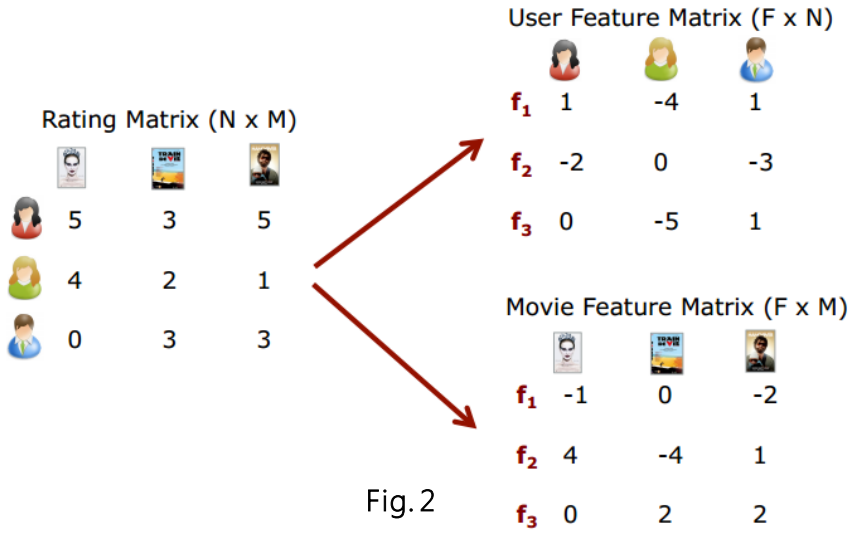


Fig. 2

The major challenge is computing the mapping of each item and user to factor vectors qi, pu ∈ □ f. After the recommender system completes this mapping, it can easily estimate the rating a user will give to any item by using Equation 1.

Such a model is closely related to singular value decomposition (SVD), a well-established technique for identifying latent semantic factors in information retrieval. Applying SVD in the collaborative filtering domain requires factoring the user-item rating matrix. This often raises difficulties due to the high portion of missing values caused by sparseness in the user-item ratings matrix. Conventional SVD is undefined

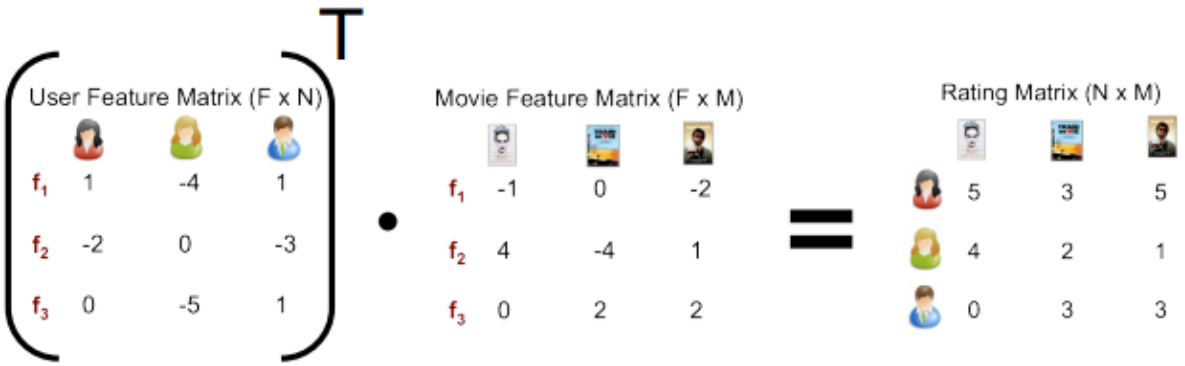


Fig. 3

when knowledge about the matrix is incomplete. Moreover, carelessly addressing only the relatively few known entries is highly prone to overfitting.

Earlier systems relied on imputation to fill in missing ratings and make the rating matrix dense. 2 However, imputation can be very expensive as it significantly increases the amount of data. In addition, inaccurate imputation might distort the data considerably. Hence, more recent works3-6 suggested modeling directly the observed ratings only, while avoiding over fitting through a regularized model.

1.6 Fundamental Algorithms [3]

The prototypical multiplicative algorithm originated with Lee and Seung (Lee and Seung, 2001). Their multiplicative update algorithm with the mean squared error objective function (using MATLAB array operator notation) is provided below:

MULTIPLICATIVE UPDATE ALGORITHM FOR NMF

$$\begin{aligned}
&\mathbf{W} = \text{rand(m,k)}; \quad \% \text{ initialize } \mathbf{W} \text{ as random dense matrix} \\
&\mathbf{H} = \text{rand(k,n)}; \quad \% \text{ initialize } \mathbf{H} \text{ as random dense matrix} \\
&\text{for i} = 1 : \text{maxiter} \\
&\quad (\text{MU}) \qquad \mathbf{H} = \mathbf{H} \,.*\, (\mathbf{W}^T\mathbf{A}) \,./\, (\mathbf{W}^T\mathbf{W}\mathbf{H} + 10^{-9}); \\
&\quad (\text{MU}) \qquad \mathbf{W} = \mathbf{W} \,.*\, (\mathbf{A}\mathbf{H}^T) \,./\, (\mathbf{W}\mathbf{H}\mathbf{H}^T + 10^{-9}); \\
&\text{end}
\end{aligned}$$

Eq. 4

The 10^{-9} in each update rule is added to avoid division by zero.

1.7 Alternating least squares

Because both w_i and h_u are unknowns, Equation 2 is not convex. However, if we fix one of the unknowns, the optimization problem becomes quadratic and can be solved optimally. Thus, ALS techniques rotate between fixing the q_i's and fixing the w_u's. When all h_u's are fixed, the system recomputes the h_i's by solving a least-squares

problem, and vice versa. This ensures that each step decreases Equation 2 until convergence.

input λ_W, λ_H
$\mathbf{W}$ = rand(m,k); % initialize $\mathbf{W}$ as random dense matrix or use another initialization from Section 4
for i = 1 : maxiter
(CLS) Solve for $\mathbf{H}$ in matrix equation $(\mathbf{W}^T\mathbf{W} + \lambda_H\mathbf{I})\ \mathbf{H} = \mathbf{W}^T\mathbf{A}$. % for $\mathbf{W}$ fixed, find $\mathbf{H}$
(NONNEG) Set all negative elements in $\mathbf{H}$ to 0.
(CLS) Solve for $\mathbf{W}$ in matrix equation $(\mathbf{H}\mathbf{H}^T + \lambda_W\mathbf{I})\ \mathbf{W}^T = \mathbf{H}\mathbf{A}^T$. % for $\mathbf{H}$ fixed, find $\mathbf{W}$
(NONNEG) Set all negative elements in $\mathbf{W}$ to 0.
end

Eq. 5

While in general stochastic gradient descent is easier and faster than ALS, ALS is favorable in at least two cases. The first is when the system can use parallelization. In ALS, the system computes each qi independently of the other item factors and computes each wu independently of the other user factors. This gives rise to potentially massive parallelization of the algorithm. The second case is for systems centered on implicit data. Because the training set cannot be considered sparse, looping over each single training case—as gradient descent does—would not be practical. ALS can efficiently handle such cases.

1.8 Particle Swarm Optimization[4]

1.8.1 Introduction

Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995, inspired by social behavior of bird flocking or fish schooling.

PSO shares many similarities with evolutionary computation techniques such as Genetic Algorithms (GA). The system is initialized with a population of random solutions and searches for optima by updating generations. However, unlike GA, PSO has no evolution operators such as crossover and mutation. In PSO, the potential

solutions, called particles, fly through the problem space by following the current optimum particles.

Each particle keeps track of its coordinates in the problem space which are associated with the best solution (fitness) it has achieved so far. (The fitness value is also stored.) This value is called pbest. Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the neighbors of the particle. This location is called lbest. when a particle takes all the population as its topological neighbors, the best value is a global best and is called gbest.

The particle swarm optimization concept consists of, at each time step, changing the velocity of (accelerating) each particle toward its pbest and lbest locations (local version of PSO). Acceleration is weighted by a random term, with separate random numbers being generated for acceleration toward pbest and lbest locations.

In past several years, PSO has been successfully applied in many research and application areas. It is demonstrated that PSO gets better results in a faster, cheaper way compared with other methods.

Another reason that PSO is attractive is that there are few parameters to adjust. One version, with slight variations, works well in a wide variety of applications. Particle swarm optimization has been used for approaches that can be used across a wide range of applications, as well as for specific applications focused on a specific requirement.

1.8.2 Algorithm

As stated before, PSO simulates the behaviors of bird flocking. Suppose the following scenario: a group of birds are randomly searching food in an area. There is only one piece of food in the area being searched. All the birds do not know where the food is. But they know how far the food is in each iteration. So what's the best strategy to find the food? The effective one is to follow the bird which is nearest to the food.

PSO learned from the scenario and used it to solve the optimization problems. In PSO, each single solution is a "bird" in the search space. We call it "particle". All of particles have fitness values which are evaluated by the fitness function to be

optimized, and have velocities which direct the flying of the particles. The particles fly through the problem space by following the current optimum particles.

PSO is initialized with a group of random particles (solutions) and then searches for optima by updating generations. In every iteration, each particle is updated by following two "best" values. The first one is the best solution (fitness) it has achieved so far. (The fitness value is also stored.) This value is called pbest. Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the population. This best value is a global best and called gbest. When a particle takes part of the population as its topological neighbors, the best value is a local best and is called lbest.

Particle Swarm Optimization might sound complicated, but it's really a very simple algorithm. Over a number of iterations, a group of variables have their values adjusted closer to the member whose value is closest to the target at any given moment. Imagine a flock of birds circling over an area where they can smell a hidden source of food. The one who is closest to the food chirps the loudest and the other birds swing around in his direction. If any of the other circling birds comes closer to the target than the first, it chirps louder and the others veer over toward him. This tightening pattern continues until one of the birds happens upon the food. It's an algorithm that's simple and easy to implement.

The algorithm keeps track of three global variables:

- Target value or condition
- Global best (gBest) value indicating which particle's data is currently closest to the Target
- Stopping value indicating when the algorithm should stop if the Target isn't found

Each particle consists of:

- Data representing a possible solution
- A Velocity value indicating how much the Data can be changed
- A personal best (pBest) value indicating the closest the particle's Data has ever come to the Target

The particles' data could be anything. In the flocking birds example above, the data would be the X, Y, Z coordinates of each bird. The individual coordinates of each bird would try to move closer to the coordinates of the bird which is closer to the food's coordinates (gBest). If the data is a pattern or sequence, then individual pieces of the data would be manipulated until the pattern matches the target pattern.

The velocity value is calculated according to how far an individual's data is from the target. The further it is, the larger the velocity value. In the birds example, the individuals furthest from the food would make an effort to keep up with the others by flying faster toward the gBest bird.

If the data is a pattern or sequence, the velocity would describe how different the pattern is from the target, and thus, how much it needs to be changed to match the target.

Each particle's pBest value only indicates the closest the data has ever come to the target since the algorithm started.

The gBest value only changes when any particle's pBest value comes closer to the target than gBest. Through each iteration of the algorithm, gBest gradually moves closer and closer to the target until one of the particles reaches the target.

After finding the two best values, the particle updates its velocity and positions with following equation (a) and (b).

v[] = v[] + c1 * rand() * (pbest[] - present[]) + c2 * rand() * (gbest[] - present[]) (a)

present[] = persent[] + v[] (b)

v[] is the particle velocity, persent[] is the current particle (solution). pbest[] and gbest[] are defined as stated before. rand () is a random number between (0,1). c1, c2 are learning factors. usually c1 = c2 = 2.

Pseudo Code

```
For each particle {
    Initialize particle
}
Do until maximum iterations or minimum error criteria {
    For each particle {
```

```
    Calculate Data fitness value
    If the fitness value is better than pBest {
        Set pBest = current fitness value
    }
    If pBest is better than gBest {
        Set gBest = pBest
    }
}
For each particle {
    Calculate particle Velocity
    Use gBest and Velocity to update particle Data
}
```

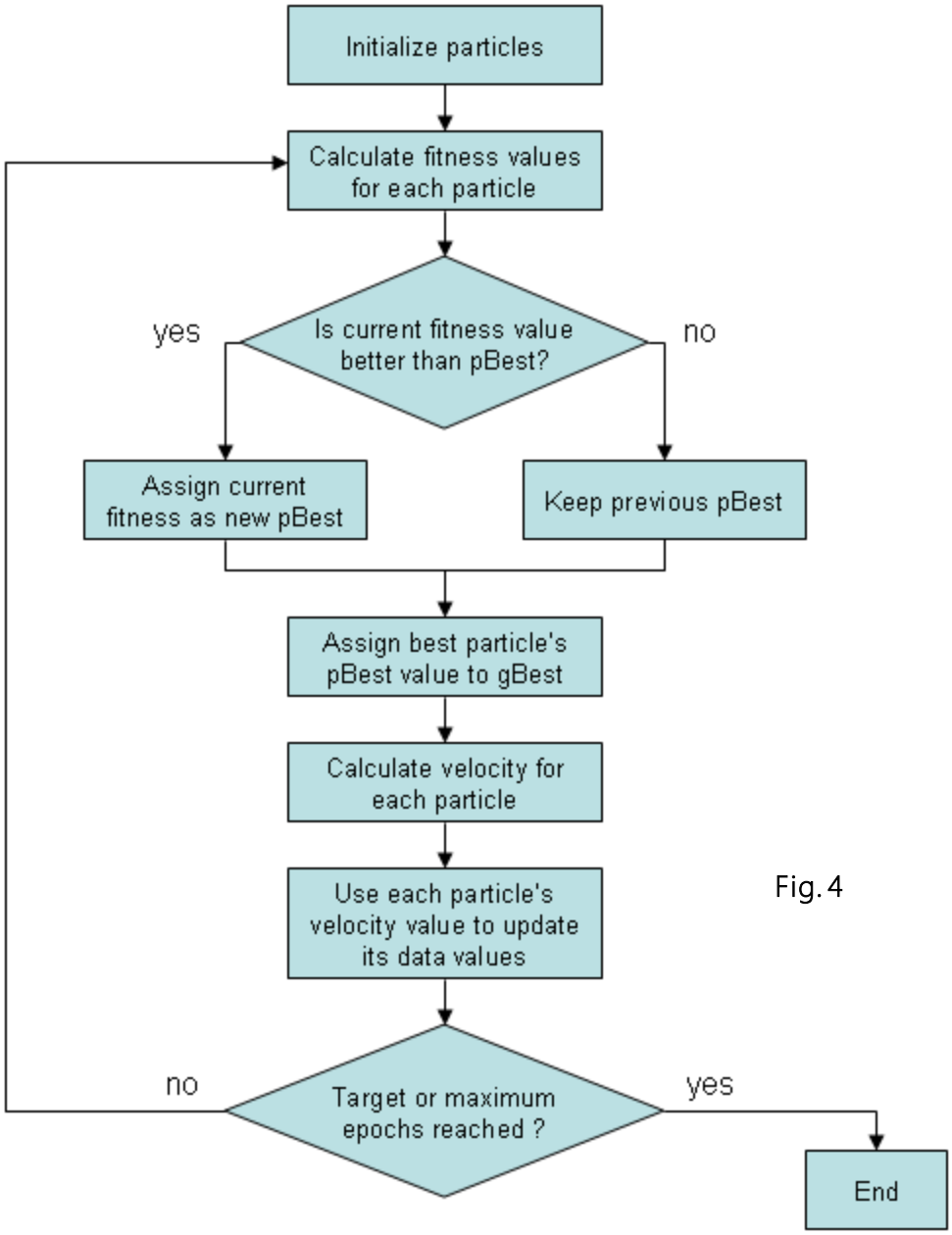


Fig. 4

1.8.3 PSO parameter control

From the above case, we can learn that there are two key steps when applying PSO to optimization problems: the representation of the solution and the fitness function. One of the advantages of PSO is that PSO take real numbers as particles. It is not like GA, which needs to change to binary encoding, or special genetic operators have to be used. For example, we try to find the solution for f(x) = x1^2 + x2^2+x3^2, the particle can be set as (x1, x2, x3), and fitness function is f(x). Then we can use the standard procedure to find the optimum.

The searching is a repeat process, and the stop criteria are that the maximum iteration number is reached or the minimum error condition is satisfied.

There are not many parameter need to be tuned in PSO. Here is a list of the parameters and their typical values.

The number of particles: the typical range is 20 - 40. Actually for most of the problems 10 particles is large enough to get good results. For some difficult or special problems, one can try 100 or 200 particles as well.

- Dimension of particles: It is determined by the problem to be optimized,
- Range of particles: It is also determined by the problem to be optimized, you can specify different ranges for different dimension of particles.
- Vmax: it determines the maximum change one particle can take during one iteration. Usually we set the range of the particle as the Vmax for example, the particle (x1, x2, x3) X1 belongs [-10, 10], then Vmax = 20
- Learning factors: c1 and c2 usually equal to 2. However, other settings were also used in different papers. But usually c1 equals to c2 and ranges from [0, 4]
- The stop condition: the maximum number of iterations the PSO execute and the minimum error requirement. for example, for ANN training in previous section, we can set the minimum error requirement is one mis-classified pattern. the maximum number of iterations is set to 2000. this stop condition depends on the problem to be optimized.

Global version vs. local version: we introduced two versions of PSO. global and local version. global version is faster but might converge to local optimum for some

problems. Local version is a little bit slower but not easy to be trapped into local optimum. One can use global version to get quick result and use local version to refine the search.

Another factor is inertia weight, which is introduced by Shi and Eberhart. If you are interested in it, please refer to their paper in 1998. (Title:).

1.9 A modified particle swarm optimizer[5]

$$v_{id} = v_{id} + c_1 * \text{rand}( ) * (p_{id} - x_{id}) + c_2 * \text{Rand}( ) * (p_{gd} - x_{id}) \quad \text{Eq. 6a}$$

$$x_{id} = x_{id} + v_{id} \quad \text{Eq. 6b}$$

Refer to equation (6a). the right side of which consists of three parts: the first part is the previous velocity of the particle; the second and third parts are the ones contributing to the change of the velocity of a particle. Without these two parts, the particles will keep on "flying" at the current speed in the same direction until they hit the boundary. PSO will not find an acceptable solution unless there are acceptable solutions on their "flying" trajectories. But that is a rare case.

On the other hand refer to equation (6a) without the first part. Then the "flying" particles' velocities are only determined by their current positions and their best positions in history. The velocity itself is memoryless.

Assume at the beginning, the particle i has the best global position, then the particle z will be "Flying" at the velocity 0, that is, it will keep still until another particle takes over the global best position. At the same time, each other particle will be "flying" toward its weighted centroid of its own best position and the global best position of the population. A recommended choice for constant c1 and c2 is integer 2 since it on average makes the weights for "social" and "cognition'' parts to be 1. Under this condition, the particles statistically contract swarm to the current global best position until another particle takes over from which time all the particles statistically contract to the new global best position.

Therefore, it can be imagined that the search process for PSO without the first part is a process where the search space statistically shrinks through the generations. It

resembles a local search algorithm. This can be illuminated more clearly by displaying the "flying" process on a screen. From the screen, it can be easily seen that without the first part of equation (6a), all the particles will tend to move toward the same position, that is, the search area is contracting through the generations. Only when the global optimum is within the initial search space, then there is a chance for PSO to find the solution. The final solution is heavily dependent on the initial seeds (population). So it is more likely to exhibit a local search ability without the first part.

On the other hand, by adding the first part, the particles have a tendency to expand the search space, that is, they have the ability to explore the new area. So they more likely have a global search ability by adding the first part. Both the local search and global search will benefit solving some kinds of problems. There is a tradeoff between the global and local search for different problems, there should be different balances between the local searchability and global search ability.

Considering of this, an inertia weight w is brought into the equation (6a) as shown. This w plays the role of balancing the global search and local search. It can be a positive constant or even a positive linear or nonlinear function of time.

```
v[] = w * v[] + c1 * rand() * (pbest[] - present[]) + c2 * rand() * (gbest[] - present[])
present[] = persent[] + v[]                                                    Eq. 7
```

When w is small (<O.8), if the PSO finds the global optimum. then it finds it fast and all the particles tend to move together quickly. This confirms our discussion in the previous section that when w is small, the PSO is more like a local search algorithm. If there is acceptable solution within the initial search space, then the PSO will find the global optimum quickly, otherwise it will not find the global optimum.

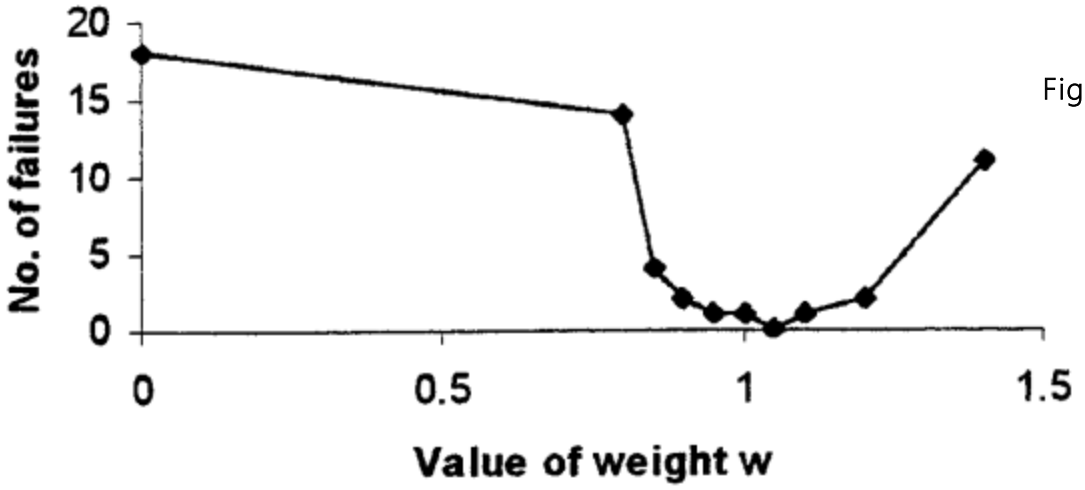


Fig. 5

When w is large (>1.2), the PSO is more like a global search method and even more it always tries to exploit the new areas. It is natural that the PSO will take more iterations to find the global optimum and have more chances of failing to find the global optimum. When w is medium (0.8<w<1.2). the PSO will have the best chance to find the global optimum but also takes a moderate number of iterations.

Here we used Chaotic Inertia Weight strategy, which turn out be the best in comparison to 15 other strategies [6].

$$z = 4 \times z \times (1 - z)$$

$$w = (w_1 - w_{2)} \times \frac{MAXiter - iter}{MAXiter} + w_2 \times z$$

Eq. 8

2.0 NMF Initialization Using Population Based Algorithms [7]

Before describing new initialization methods using population based algorithms, we discuss some properties of the Frobenius norm, which is used as objective function to measure the quality of the NMF approximation. The Frobenius norm of a matrix $D \in R_{m \times n}$ is defined as:

$$||D||_F = \left(\sum_{i=1}^{min(m,n)} \sigma_i \right)^{1/2} = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |d_{ij}|^2 \right)^{1/2} \quad \text{Eq. 9}$$

where σ_i are the singular values of D, and d_{ij} is the element in the i^{th} row and j^{th} column of D. The Frobenius norm can also be computed row wise or column wise. The *row wise* calculation is

$$||D||_F^{RW} = \left(\sum_{i=1}^{m} |\mathbf{d}_i^r|^2 \right)^{1/2} \quad \text{Eq. 10}$$

Obviously, a reduction of the Frobenius norm of any row or any column of D leads to a reduction of the total Frobenius norm $//D//$. In the following, D refers to the distance matrix of the original data and the approximation, $D = A - WH$.

2.1 Initialization procedure.

We exploit these properties of the Frobenius norm to initialize the basis vectors in W row wise and the coefficient matrix H column wise. The goal is to find heuristically optimal starting points for single rows of W and single columns of H, which can be computed with any PBAs like PSO etc.

The below algorithm shows the pseudo code for the initialization procedure. In the beginning, H_0 needs to be initialized randomly using a non-negative lower bound for the initialization. In the first loop, W is initialized row wise, i.e., row w^r_i is optimized in order to minimize the Frobenius norm of the i^{th} row d^r_i of D, which is defined as $d^r_i = a^r_i - w^r_i H_0$.

In the second loop, the columns of H are initialized using on the previously computed rows of W. H is initialized column wise i.e., column h^c_j is optimized in order to minimize the Frobenius norm of the j^{th} column d^c_j of D, which is defined as

$d^c_j = a^c_j - Wh^c_j$.

Given matrix $A \in \mathbb{R}^{m \times n}$ and $k \ll min\{m, n\}$;
$H0 = \text{rand}(k, n)$;
for $i = 1$ *to* m **do**
    Use PBAs to find $\mathbf{w}_i^r$ that minimizes $||\mathbf{a}_i^r - \mathbf{w}_i^r H0||_F$
    $W(i, :) = \mathbf{w}_i^r$;
end
for $j = 1$ *to* n **do**
    Use PBAs to find $\mathbf{h}_j^c$ that minimizes $||\mathbf{a}_j^c - W\mathbf{h}_j^c||_F$,
    $H(:, j) = \mathbf{h}_j^c$;
end

Fig. 9

In line 4, input parameters for the PBAs are a^r_i (the i^{th} row of A) and H_0, the output is the initialized row vector w^r_i, the i^{th} row of W. In line 8, input parameters are a^c_j (the j^{th} column of A) and the *already optimized* factor W, the output is the initialized column vector h^c_j, the j^{th} column of H. Global parameters used for all PBAs are upper/lower bound of the search space and the initialization (the starting values of the PBAs), number of particles (chromosomes, fish, ...), and maximum number of fitness evaluations. The dimension of the optimization problem is identical to the rank k of the NMF.

2. Literature Survey

S. No.	Publisher Name & Year	Paper Title	Objective	Technique/ Algorithm
1	Advances in Neural Information Processing Systems 13 (NIPS 2000)	Algorithms for Non-negative Matrix Factorization	To run and compare two different multiplicative algorithms for NMF are analyzed.	Multiplicative update rules and Additive update rules are explained with proofs.
2	IEEE Computer Volume 42 Issue 8, 2009	Matrix factorization techniques for recommender systems	Explains various Recommender system strategies and leverages Matrix Factorization to achieve superior recommendations	Simply introduces a Matrix factorization model and puts forward many Learning algorithms like ALS, Multiplicative updates.
3	Computational Statistics and Data Analysis 2006.	Algorithms and applications for approximate nonnegative matrix factorization	This paper discusses the development and use of low-rank approximate non-negative matrix factorization (NMF) algorithms along with sparsity and smoothness constraints.	Explain already existing multiplicative update and alternating least squares algorithms as well introduces their own CNMF.
4	Neural Networks, 1995. Proceedings., IEEE International Conference	Particle swarm optimization	A concept for the optimization of nonlinear functions using particle swarm methodology is introduced. The evolution of several paradigms is outlined, and an implementation of one of the paradigms is discussed.	Algorithm uses a number of agents (particles) that constitute a swarm moving around in the search space looking for the best solution. Each particle in search space adjusts its "flying" according to its own flying experience as well as the flying experience of other particles.
5	Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence	A modified particle swarm optimizer	To improve existing PSO, they have introduced a new parameter, called inertia weight.	The inertia weight w plays the role of balancing the global search and local search. It is introduced in the velocity update formula of PSO.

6	Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress	Inertia Weight strategies in Particle Swarm Optimization	In order to propose one or more than one Inertia Weight strategies which are efficient than others, this paper studies 15 relatively recent and popular Inertia Weight strategies and compares their performance on 05 optimization test problems.	Different inertia weight formula were applied in the original PSO algorithm and the results were compared.
7	Advances in Swarm Intelligence Lecture Notes in Computer Science Volume 6729, 2011	Using Population Based Algorithms for Initializing Nonnegative Matrix Factorization	This paper investigates the application of five population based algorithms (genetic algorithms, particle swarm optimization, fish school search, differential evolution, and fireworks algorithm) as new initialization variants for NMF.	Using existing PBA's, we initialize our seed matrices W and H for best factorization.

3. Problem definition

Given an n × m real nonnegative matrix V and a positive integer k < min(n, m), it aims to find a factorization of V into an n × k real matrix W and an k × m real matrix H such that:

$$\mathbf{V} \approx \mathbf{WH}$$

Our main aim would be to reduce the error while reconstruction V by multiplying W and H. This would give us a matrix W which will have be a reduced set of features as compared to W but still provide us with good classification, clustering etc.

4. Proposed approach

1. We first run a simple NMF algorithm by initialising W and H as random matrices and modifying them with Multiplicative update rules.

MULTIPLICATIVE UPDATE ALGORITHM FOR NMF

$\mathbf{W}$ = rand(m,k); % initialize $\mathbf{W}$ as random dense matrix
$\mathbf{H}$ = rand(k,n); % initialize $\mathbf{H}$ as random dense matrix
for i = 1 : maxiter
(MU) $\mathbf{H} = \mathbf{H} .* (\mathbf{W}^T\mathbf{A}) ./ (\mathbf{W}^T\mathbf{W}\mathbf{H} + 10^{-9})$;
(MU) $\mathbf{W} = \mathbf{W} .* (\mathbf{A}\mathbf{H}^T) ./ (\mathbf{W}\mathbf{H}\mathbf{H}^T + 10^{-9})$;
end

2. We then tried ACLS as the update rule.

input λ_W, λ_H
$\mathbf{W}$ = rand(m,k); % initialize $\mathbf{W}$ as random dense matrix or use another initialization from Section 4
for i = 1 : maxiter
(CLS) Solve for $\mathbf{H}$ in matrix equation $(\mathbf{W}^T\mathbf{W} + \lambda_H\mathbf{I})\,\mathbf{H} = \mathbf{W}^T\mathbf{A}$. % for $\mathbf{W}$ fixed, find $\mathbf{H}$
(NONNEG) Set all negative elements in $\mathbf{H}$ to 0.
(CLS) Solve for $\mathbf{W}$ in matrix equation $(\mathbf{H}\mathbf{H}^T + \lambda_W\mathbf{I})\,\mathbf{W}^T = \mathbf{H}\mathbf{A}^T$. % for $\mathbf{H}$ fixed, find $\mathbf{W}$
(NONNEG) Set all negative elements in $\mathbf{W}$ to 0.
end

3. To be able to initialise W and H (not randomly) so that they may act as better seed values, we use a PBA known as PSO.

Given matrix $A \in \mathbb{R}^{m\times n}$ and $k \ll min\{m,n\}$;
$H0 = \text{rand}(k,n)$;
for $i = 1$ *to* m **do**
Use PBAs to find $\mathbf{w}_i^r$ that minimizes $||\mathbf{a}_i^r - \mathbf{w}_i^r H0||_F$
$W(i,:) = \mathbf{w}_i^r$;
end
for $j = 1$ *to* n **do**
Use PBAs to find $\mathbf{h}_j^c$ that minimizes $||\mathbf{a}_j^c - W\mathbf{h}_j^c||_F$,
$H(:,j) = \mathbf{h}_j^c$;
end

4. PSO too can be optimized further by introducing an inertia weight strategy which would help us maintain a good balance between global and local search. Here we use Chaotic Inertia Weight strategy.

$$z = 4 \times z \times (1 - z)$$

$$w = (w_1 - w_{2)} \times \frac{MAXiter - iter}{MAXiter} + w_2 \times z$$

5. We run various algorithms by setting up the parameters and note down the experimental results including convergence rate, time taken etc..

5. Tools, Software and Hardware used

1. NetBeans IDE 8.0.2
2. Java
3. ejml
4. Weka 3.6.2

6. Activity Time chart

Before Mid-Sem	
25th Jan.	Literature Survey which included learning about various approaches to our problem.
5th Feb.	Familiarizing with various matrix libraries for JAVA.
25th Feb	Coding and executing a NNMF with Multiplicative update rule.
After Midsem	
5th March	Apply ACLS as the update rule.
10th April	Apply PSO to initialize W and H.
30th April	Run experiments and tabulate the results.

7. Results

We used 3 datasets to evaluate our 3 different NMF strategies. To perform the evaluation were found the RMSE after each iteration by comparing V with (W.H). This we did for all 3 strategies and tabulated the results.

We obtained the following results:

(I) Wall-Following Robot Navigation Dataset

The data were collected as the SCITOS G5 robot navigates through the room following the wall in a clockwise direction using 24 ultrasound sensors arranged circularly around its 'waist'.

Number of Instances: 5456
Number of Attributes: 24
Number of classes: 4 {Move-Forward, Slight-Right-Turn, Sharp-Right-Turn, Slight-Left-Turn}
Hence, m = 5456, n = 24 and we selected k = 20.

Table 1

Wall-Following Robot Navigation Dataset

No. of interation	NMF-Multiplicative update RMSE	Error difference = RMSE(i) - RMSE(i + 1)	NMF-ACLS RMSE	Error difference = RMSE(i) - RMSE(i + 1)	NMF-Multiplicative update-PSO RMSE	Error difference = RMSE(i) - RMSE(i + 1)
0	6.059455		12.32521		6.059513	
5	6.037156	0.022299	3.429139	8.896071	6.035455	0.024058
15	5.361941	0.675215	2.353954	1.075185	5.269997	0.765458
30	3.786018	1.575923	1.886802	0.467152	3.505313	1.764684
100	2.325166	1.460852	1.807037	0.079765	1.94646	1.558853
200	1.964569	0.360597	1.776049	0.030988	1.859392	0.087068
500	1.796323	0.168246	1.762646	0.013403	1.814599	0.044793
1500	1.733934	0.062389	1.762556	9E-05	1.729706	0.084893
3000	1.721252	0.012682	1.762556	0	1.70945	0.020256
6000	0.711584	1.009668	1.762556	0	0.50542	1.20403

(II) Hill-Valley Data Set

Each record represents 100 points on a two-dimensional graph. When plotted in order (from 1 through 100) as the Y co-ordinate, the points will create either a Hill (a "bump" in the terrain) or a Valley (a "dip" in the terrain).

Number of Instances: 606
Number of Attributes: 100
Hence, m =606, n = 100 and we selected k = 90
Number of classes: 2 {0 – Dip, 1 - Hill}

Table 2

Hill-Valley Data Set

No. of interation	NMF-Multiplicative update RMSE	Error difference = RMSE(i) - RMSE(i + 1)	NMF-ACLS RMSE	Error difference = RMSE(i) - RMSE(i + 1)	NMF-Multiplicative update-PSO RMSE	Error difference = RMSE(i) - RMSE(i + 1)
0	12.9240982		151.5741929		12.9275183	
5	12.5778483	0.3462499	27.7422149	123.831978	12.5543239	0.3731944
15	11.9053897	0.6724586	21.7816664	5.9605485	11.8355348	0.7187891
30	10.53624004	1.369149665	8.2134291	13.5682373	10.3800315	1.4555033
100	4.61151329	5.924726745	8.11342915	0.09999995	4.5858257	5.7942058
200	2.35968439	2.2518289	8.11342915	0	2.3490185	2.2368072
500	1.05979564	1.29988875	8.11342915	0	1.0509191	1.2980994
1500	0.424059373	0.635736267	8.11342915	0	0.4293838	0.6215353
3000	0.272138481	0.151920892	8.11342915	0	0.2709546	0.1584292
6000	0.13298025	0.139158231	8.11342915	0	0.1089434	0.1620112

(III) Breast Cancer Wisconsin (Diagnostic) Data Set

32 features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass.

Number of Instances: 569
Number of Attributes: 32
Hence, m = 569, n = 32 and we selected k = 25.

Number of classes: 2 {Malignant, Benign}

Table 3

Breast Cancer Wisconsin (Diagnostic) Data Set

No. of interation	NMF-Multiplicative update RMSE	Error difference = RMSE(i) - RMSE(i + 1)	NMF-ACLS RMSE	Error difference = RMSE(i) - RMSE(i + 1)	NMF-Multiplicative update-PSO RMSE	Error difference = RMSE(i) - RMSE(i + 1)
0	117.2449364		382.0679107		116.3006716	
5	108.3448003	8.90013615	115.8986652	266.1692455	107.8506583	8.45001329
15	101.5873926	6.757407682	41.4982087	74.4004565	100.2362706	7.61438775
30	70.3152527	31.27213987	41.4982087	0	67.08163274	33.15463782
100	24.2059054	46.1093473	41.4982087	0	26.17936433	40.90226841
200	10.83363865	13.37226675	41.4982087	0	11.80992584	14.36943849
500	6.6176607	4.21597795	39.2705171	2.2276916	6.46314259	5.34678325
1500	4.31826574	2.29939496	25.4612028	13.8093143	4.00630465	2.45683794
3000	2.53084496	1.78742078	25.4612028	0	2.60556953	1.40073512
6000	1.953408974	0.577435986	25.4612028	0	1.1353657	1.47020383

From the above results, we can conclude that NMF-Multiplicative update-PSO out performs both NMF-Multiplicative update and NMF-ACLS. Most unsatisfactory results are given by NMF-ACLS which clearly suffers from getting stuck at local minima.

8. Conclusion and future scope

In this project, we introduced new initialization variants for nonnegative matrix factorization (NMF) using a popular population based algorithm (PBAs) - particle swarm optimization (PSO. These algorithms were used to initialize the rows of the NMF factor W, and the columns of the other factor H, in order to achieve a smaller approximation error for a given number of iterations. Overall, the new initialization variants achieve better approximation results than random initialization and state-of-the-art methods.

A future goal is to improve NMF algorithms by utilizing heuristic search methods and various other strategies like simulated annealing [8] to avoid NMF getting stuck in local minima.

References

[1] Daniel D. Lee, H. Sebastian Seung. "Algorithms for Non-negative Matrix Factorization". *Advances in Neural Information Processing Systems 13 (NIPS 2000).*

[2] Yehuda Koren, Robert Bell and Chris Volinsky. "Matrix factorization techniques for recommender systems". *IEEE Computer Volume 42 Issue 8, August 2009 Pages 30-37.*

[3] Michael W. Berry , Murray Browne , Amy N. Langville , V. Paul Pauca , Robert J. Plemmons. "Algorithms and applications for approximate nonnegative matrix factorization". *Computational Statistics and Data Analysis 2006.*

[4] Kennedy, J.,Eberhart, R. "Particle swarm optimization". *Neural Networks, 1995. Proceedings., IEEE International Conference (Volume:4) Pages 1942 - 1948.*

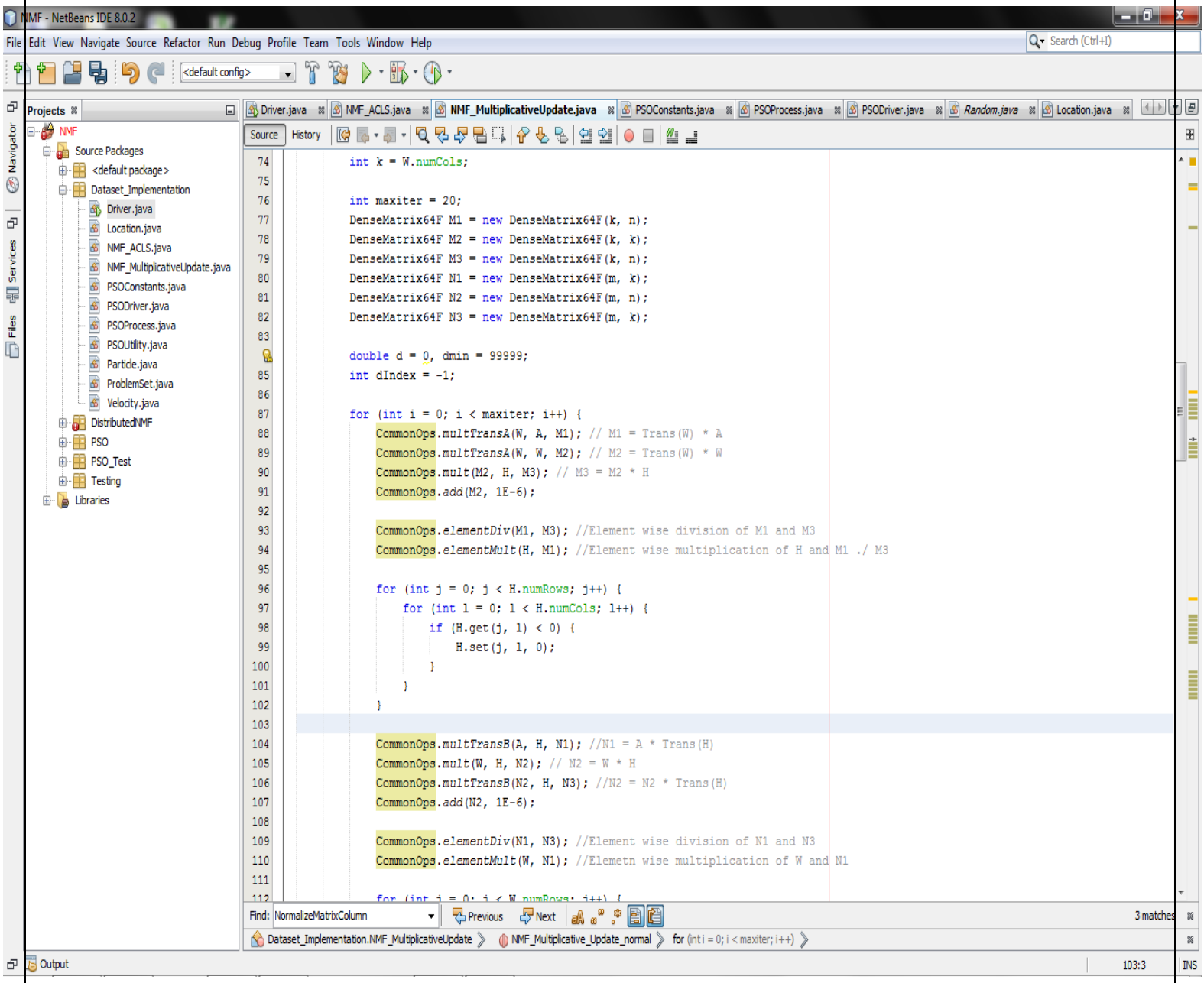
[5] Yuhui Shi; Eberhart, R. "A modified particle swarm optimizer". *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., Page 69-73.*

[6] Bansal, J.C. "Inertia Weight strategies in Particle Swarm Optimization". *Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on 21 Oct. 2011, Page(s) 633 - 640.*

[7] Andreas Janecek; Ying Tan. "Using Population Based Algorithms for Initializing Nonnegative Matrix Factorization". *Advances in Swarm Intelligence Lecture Notes in Computer Science Volume 6729, 2011, pp 307-31.*

[8] Honghong Peng; Rao, R.; Dianat, S.A. "Nonnegative matrix factorization with deterministic annealing for unsupervised unmixing of hyperspectral imagery". *Image Processing (ICIP), 2012 19th IEEE International Conference Page(s): 2145-2148.*

Appendix


NMF - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Projects
NMF
Source Packages
<default package>
Dataset_Implementation
Driver.java
Location.java
NMF_ACLS.java
NMF_MultiplicativeUpdate.java
PSOConstants.java
PSODriver.java
PSOProcess.java
PSOUtility.java
Particle.java
ProblemSet.java
Velocity.java
DistributedNMF
PSO
PSO_Test
Testing
Libraries
Driver.java
NMF_ACLS.java
NMF_MultiplicativeUpdate.java
PSOConstants.java
PSOProcess.java
PSODriver.java
Random.java
Location.java
Source
History
int k = W.numCols;
int maxiter = 20;
DenseMatrix64F M1 = new DenseMatrix64F(k, n);
DenseMatrix64F M2 = new DenseMatrix64F(k, k);
DenseMatrix64F M3 = new DenseMatrix64F(k, n);
DenseMatrix64F N1 = new DenseMatrix64F(m, k);
DenseMatrix64F N2 = new DenseMatrix64F(m, n);
DenseMatrix64F N3 = new DenseMatrix64F(m, k);
double d = 0, dmin = 99999;
int dIndex = -1;
for (int i = 0; i < maxiter; i++) {
CommonOps.multTransA(W, A, M1); // M1 = Trans(W) * A
CommonOps.multTransA(W, W, M2); // M2 = Trans(W) * W
CommonOps.mult(M2, H, M3); // M3 = M2 * H
CommonOps.add(M2, 1E-6);
CommonOps.elementDiv(M1, M3); //Element wise division of M1 and M3
CommonOps.elementMult(H, M1); //Element wise multiplication of H and M1 ./ M3
for (int j = 0; j < H.numRows; j++) {
for (int l = 0; l < H.numCols; l++) {
if (H.get(j, l) < 0) {
H.set(j, l, 0);
}
}
}
CommonOps.multTransB(A, H, N1); //N1 = A * Trans(H)
CommonOps.mult(W, H, N2); // N2 = W * H
CommonOps.multTransB(N2, H, N3); //N2 = N2 * Trans(H)
CommonOps.add(N2, 1E-6);
CommonOps.elementDiv(N1, N3); //Element wise division of N1 and N3
CommonOps.elementMult(W, N1); //Elemetn wise multiplication of W and N1
Find: NormalizeMatrixColumn
Previous
Next
3 matches
Dataset_Implementation.NMF_MultiplicativeUpdate
NMF_Multiplicative_Update_normal
for (int i = 0; i < maxiter; i++)
Output
103:3
INS