MATHEMATICS FOR MACHINE LEARNING

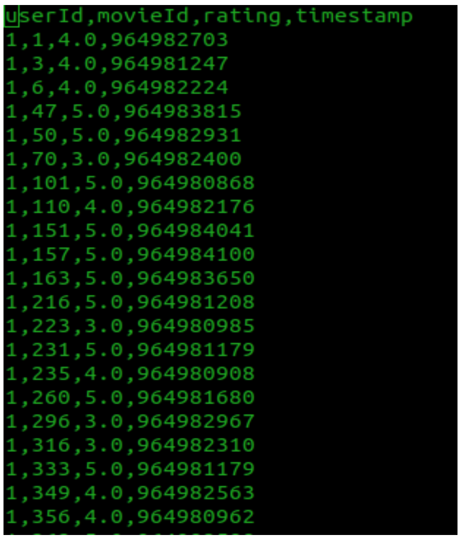
PROJECT 1
MATRIX FACTORIZATION & COLLABORATIVE FILTERING

Student:
Deepak Laxman | IST1104341
Bologna Masters degree in Biotechnology,
Instituto Superior Tecnico

1 TASK 1: To construct a sparse ratings matrix where the ij^{th} element represents the ratings given by user j to movie i.

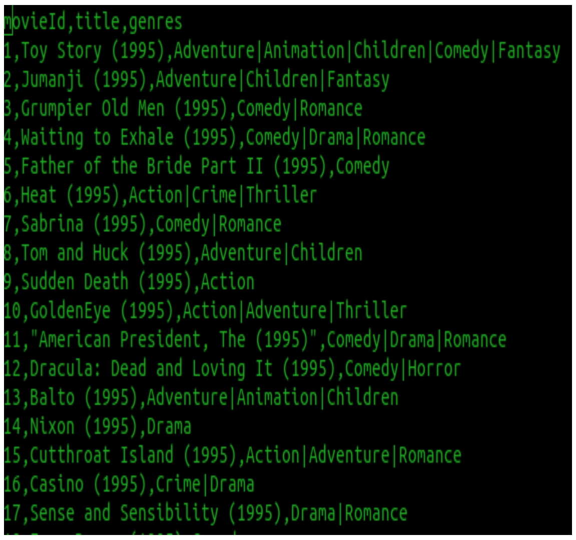
This is the basic foundation for the entire project. We are given a data set (ratings.csv) a list of users who have given ratings to various movies in a range of 1-5 along with the respective movie IDs and the timestamps. We also make use of another data set (movies.csv) to obtain the information about which movie ID corresponds to what movie.

Combining these two csv files, we would like to create a nxm matrix where n corresponds to the no.of users and m corresponds to the total movie count. Use simple python tools and functions to first extract the data and then iteratively fill the values in the sparse matrix whose other elements in which we don't know the values yet are filled with zeros by default.



```
userId,movieId,rating,timestamp
1,1,4.0,964982703
1,3,4.0,964981247
1,6,4.0,964982224
1,47,5.0,964983815
1,50,5.0,964982931
1,70,3.0,964982400
1,101,5.0,964980868
1,110,4.0,964982176
1,151,5.0,964984041
1,157,5.0,964984100
1,163,5.0,964983650
1,216,5.0,964981208
1,223,3.0,964980985
1,231,5.0,964981179
1,235,4.0,964980908
1,260,5.0,964981680
1,296,3.0,964982967
1,316,3.0,964982310
1,333,5.0,964981179
1,349,4.0,964982563
1,356,4.0,964980962
```

(a) ratings.csv data set



```
movieId,title,genres
1,Toy Story (1995),Adventure|Animation|Children|Comedy|Fantasy
2,Jumanji (1995),Adventure|Children|Fantasy
3,Grumpier Old Men (1995),Comedy|Romance
4,Waiting to Exhale (1995),Comedy|Drama|Romance
5,Father of the Bride Part II (1995),Comedy
6,Heat (1995),Action|Crime|Thriller
7,Sabrina (1995),Comedy|Romance
8,Tom and Huck (1995),Adventure|Children
9,Sudden Death (1995),Action
10,GoldenEye (1995),Action|Adventure|Thriller
11,"American President, The (1995)",Comedy|Drama|Romance
12,Dracula: Dead and Loving It (1995),Comedy|Horror
13,Balto (1995),Adventure|Animation|Children
14,Nixon (1995),Drama
15,Cutthroat Island (1995),Action|Adventure|Romance
16,Casino (1995),Crime|Drama
17,Sense and Sensibility (1995),Drama|Romance
```

(b) movies.csv data set

After the ratings matrix (**mat_array in the code**), given a positive value k, we need to find dense matrices M_{mxk} and U_{kxn} such that,

$$R = MU$$

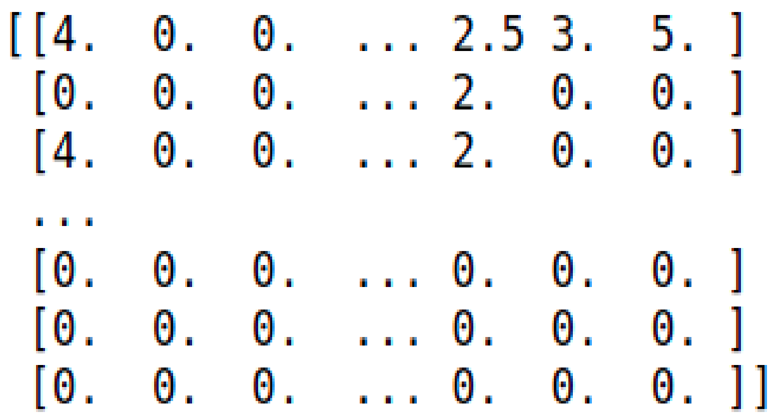
The rational behind this factorization is to is to predict the missing values in the ratings matrix and thus suggest to the users a new movie based on his previous interests in other movies and the ratings which he/she will like. In few situations, its not possible to arrive at a solution for the equation $||R - MU||$, in that case we try to minimise the error as much as possible by recurring functions and arrive at a nominal answer for the missing values of the movie.
To achieve the same, we make use of few factorization algorithms and predict the unknown values of the matrix.

1.1 Libraries used:

```
import pandas as pd
import numpy as np
import csv
from scipy.linalg import svd, sqrtm
import math
import matplotlib.pyplot as plt
from sklearn.decomposition import NMF
from numpy.linalg import solve
import numpy as np
```

1.2 Code Snippet for Matrix Construction:

```
# to open and access the data set ratings.csv
with open('/home/deepak/Documents/Maths for ML/ratings.csv') as File1:
    reader = csv.reader(File1, delimiter=',', quotechar=',', quoting=csv.QUOTE_MINIMAL)
    user_ID = []
    header = next(reader)
    for uid in reader:
        user_ID.append(uid)

# to open and access the data set movies.csv
with open('/home/deepak/Documents/Maths for ML/movies.csv') as File2:
    reader = csv.reader(File2, delimiter=',', quotechar=',', quoting=csv.QUOTE_MINIMAL)
    movie_ID = []
    header = next(reader)
    list_num = list(range(1,9743))
    for j in list_num:
        for i in reader:
            i.append(j)
            j+=1
            movie_ID.append(i)

movie_dict = {}
for i in movie_ID:
    movie_dict[i[0]] = i[-1]

mat_array = np.zeros((9742,610))

for i in range(len(user_ID)):
    mat_array[movie_dict[user_ID[i][1]]-1][int(user_ID[i][0])-1]=float(user_ID[i][2]
```



```
[[4.  0.  0.  ... 2.5 3.  5. ]
 [0.  0.  0.  ... 2.  0.  0. ]
 [4.  0.  0.  ... 2.  0.  0. ]
 ...
 [0.  0.  0.  ... 0.  0.  0. ]
 [0.  0.  0.  ... 0.  0.  0. ]
 [0.  0.  0.  ... 0.  0.  0. ]]
```

Figure 1.2: Ratings mxn matrix

2 TASK 2: Construct a SVD system and analyse the results.

This section focuses on the implementation of SVD based factorization system to the ratings matrix as a first recommendation system to users and find the other missing values in the matrix. SVD, also expanded as the **Single Value Decomposition** technique is a very much appreciated and well known method to factorize a matrix into 3 sub matrix components. The first and last sub matrices are orthogonal in nature and the middle matrix is a sparse matrix with the only non zero elements being the singular value of the original ratings matrix. With the help of a default function **linalg.svd** from the numpy library, we can easily make use of the SVD function with just a line of code.

2.1 Algorithm

The main idea of the SVD implementation is to first effectively find the three factor matrices. And later with a variable k whose value always lies between the 0 to no.of users, will be used to splice the three matrices and approximately find the new matrix. To do so, we make a list called **k_values** whose range is from 5 to 201 with a step size of 15 in this code to make the computation part easy and fast. We define a function **uSv** whose argument is the ratings matrix array we created in the previous section. For each iterative values of K, we splice the matrices u, s, v and then later find the square root of the middle singular value matrix s. Multiplying u and the square root of s using dot product yields us a new matrix usk (M dense matrix). Similarly we do the same for s and v and arrive at skv matrix (U dense matrix. Now finally dot product usk and skv to obtain **usv**.

The matrix thus obtained is the approximation of the original ratings matrix but with each increasing values of k, it is expected that this matrix converges to the values of the original matrix. Later we find the error between both the matrices for teach values of k and evaluate the Root Mean Square Error and plot it.

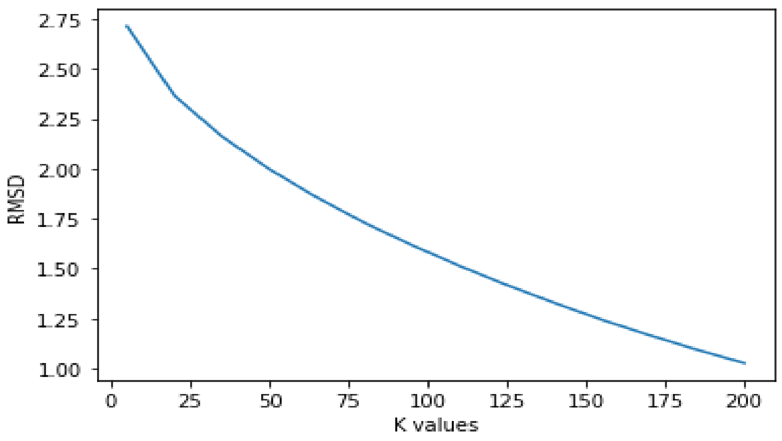


Figure 2.1: RMSE plot

The above plot clearly shows us that with an increase in the value of k, the factorized matrices actually converge towards the original ratings matrix. This is seen as a decrease in the plot, as the error decreases with an increase in the k value.

2.2 Code Snippet for SVD:

```
k_values = list(range(5,201,15))
def uSv(mat_array):

    RMSE_list = []

    for k in k_values:
        u,s,v = np.linalg.svd(mat_array, full_matrices=False)
        s = np.diag(s) #to make the 1D array into a square diagonal matrix
        #for factorization
        s = s[0:k, 0:k] #splicing the first k elements
        u = u[:,0:k]
        v = v[0:k,:]

        s_root = sqrtm(s) #possible because matrix s is always non negative

        usk = np.dot(u,s_root)
        skv = np.dot(s_root,v)
        usv = np.dot(usk,skv)#simplified ratings matrix
        with usk=M and skv=U being the dense matrices

        error_mat = []

        for i in range(9742):
            for j in range(610):
                if mat_array[i][j]!= 0: #finding all the non zero elements
                    error_mat.append((mat_array[i][j] - usv[i][j])**2) #subtracting
                    corresponding values from matrix

        RMSE_list.append(math.sqrt(sum(error_mat)/100836)) #RMSE formula
    return RMSE_list

s = uSv(mat_array)

plt.plot(k_values,s)
plt.xlabel('K␣values')
plt.ylabel('RMSD')
plt.show()
```

3 TASK 3: Recommendation System using Non Negative Matrix Factorization (NMF)

It is always a preferred option to have the factorized matrices to have all non-negative entries so that we can arrive at a logical interpretation of the data. In the previous SVD procedure we saw that the original ratings matrix had all non-negative entries, nevertheless, the three factorized matrices had no guarantee that all the entries will be non-negative. So we employ another algorithm which involves the usage of Non Negative Matrix factorization for finding parts based, linear representation of non-negative data.

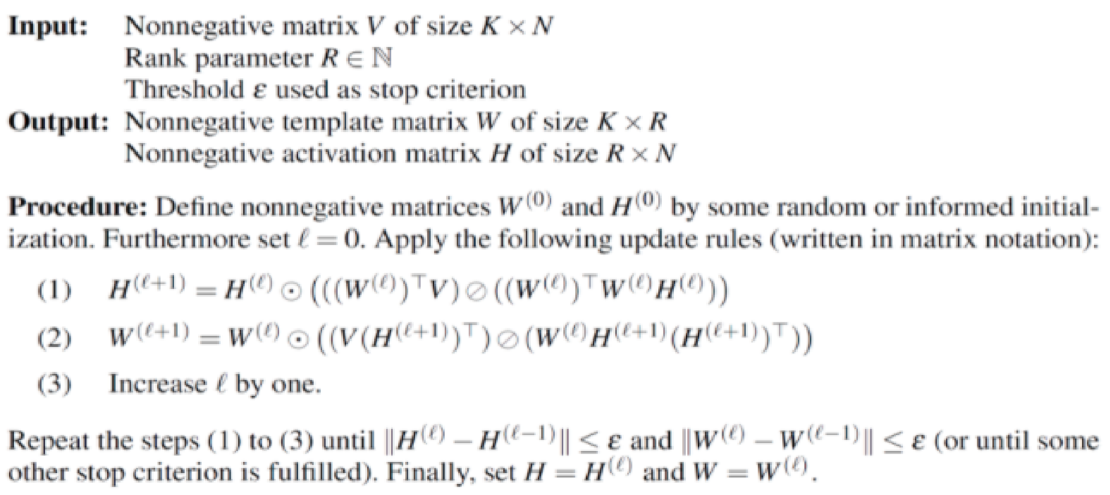
Given a number k ∈ ℕ smaller than both m and n, the goal of NMF is to find two non negative matrices W $\in \mathrm{R}_{>0}^{mxk}$ and H $\in \mathrm{R}_{>0}^{kxn}$ such that

$$V \approx W.H \tag{3.1}$$

3.1 NMF using Euclidean Distance

To find the approximate factorization of V, we need to distance function to quantify how close the approximation is made at n^{th} iteration. In this algorithm, we use the **Euclidean Distance** method to achieve the same. In other words, given a non negative matrix V and a rank parameter k, we try to minimize $||V - WH||^2$.Based on the standard gradient descent approach applied to the minimization problem of (3.1) we go further implementing the method. Since this joint optimization is quite hard, we can either fix the matrix W and then optimize with regard to H and later, this learned factor H further optimizes W.

3.1.1 Algorithm



Input: Nonnegative matrix V of size $K \times N$
Rank parameter $R \in \mathbb{N}$
Threshold ε used as stop criterion
Output: Nonnegative template matrix W of size $K \times R$
Nonnegative activation matrix H of size $R \times N$

Procedure: Define nonnegative matrices $W^{(0)}$ and $H^{(0)}$ by some random or informed initialization. Furthermore set $\ell = 0$. Apply the following update rules (written in matrix notation):

```
(1)   $H^{(\ell+1)} = H^{(\ell)} \odot (((W^{(\ell)})^\top V) \oslash ((W^{(\ell)})^\top W^{(\ell)} H^{(\ell)}))$
(2)   $W^{(\ell+1)} = W^{(\ell)} \odot ((V (H^{(\ell+1)})^\top) \oslash (W^{(\ell)} H^{(\ell+1)} (H^{(\ell+1)})^\top))$
(3)   Increase $\ell$ by one.
```

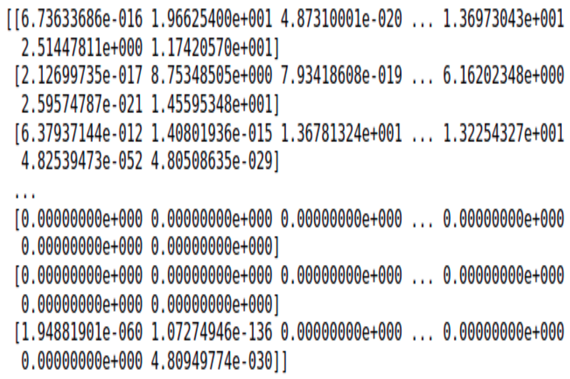
Repeat the steps (1) to (3) until $\|H^{(\ell)} - H^{(\ell-1)}\| \leq \varepsilon$ and $\|W^{(\ell)} - W^{(\ell-1)}\| \leq \varepsilon$ (or until some other stop criterion is fulfilled). Finally, set $H = H^{(\ell)}$ and $W = W^{(\ell)}$.

3.1.2 Code Snippet for NMF using Euclidean Distance

```
def nmf(V, R, threshold=0.001, n=200, W=None, H=None):

    if W is None:
        W = np.random.rand(V.shape[0], R) #initialize a random matrix
    if H is None:
        H = np.random.rand(R, V.shape[1]) #initialize a random matrix

    niter = 0
    is_below_threshold = False

    epsilon_value = sys.float_info.epsilon #system generated epsilon value
    while not is_below_threshold and niter <= n: #iteration conditions
        H_niter = H
        W_niter = W
        H = H * (W.transpose().dot(V) / (W.transpose().dot(W).dot(H) + epsilon_value))
        W = W * (V.dot(H.transpose()) / (W.dot(H).dot(H.transpose()) + epsilon_value))

        H_error = np.linalg.norm(H - H_niter, ord = 2) #norm of new and old H
        W_error = np.linalg.norm(W - W_niter, ord = 2) #norm of new and old W

        if H_error < threshold and W_error < threshold:
            is_below_threshold = True

        niter +=1

    V_approx = W.dot(H) #approximate value of V

    return W, H, V_approx

[NMF_W, NMF_H, NMF_V_approx] = nmf(mat_array,610)
print(NMF_W)
print(NMF_H)
print(NMF_V_approx)
```

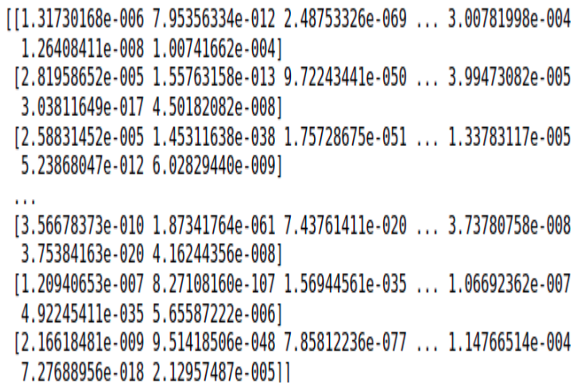
The idea is that we first initialize two random matrices with all non negative entries (W and H). With two mathematical operations done on matrix H and W, we don't compromise on loosing the non negativeness of the matrices and also we are able to converge the errors near to 0. We make use of the system generated epsilon values at the denominator of two matrix operation steps to avoid division by zero error if the other part of the denominator becomes zero. Value R in the function's argument is the rank parameter and can be any positive number between 1-610 based on the level of approximation.

3.1.3 Results



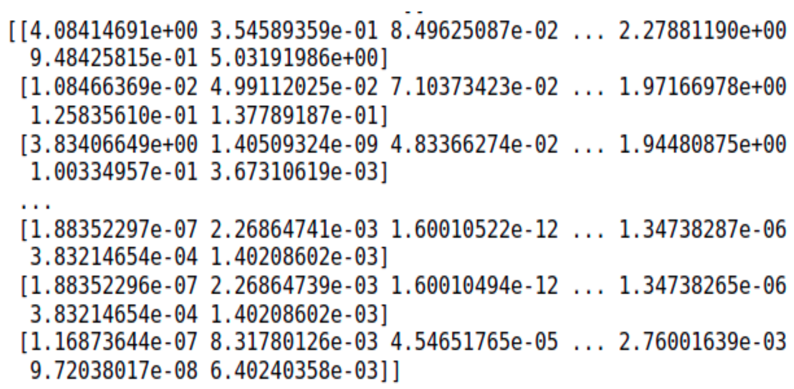
```
[[6.73633686e-016 1.96625400e+001 4.87310001e-020 ... 1.36973043e+001
  2.51447811e+000 1.17420570e+001]
 [2.12699735e-017 8.75348505e+000 7.93418608e-019 ... 6.16202348e+000
  2.59574787e-021 1.45595348e+001]
 [6.37937144e-012 1.40801936e-015 1.36781324e+001 ... 1.32254327e+001
  4.82539473e-052 4.80508635e-029]
 ...
 [0.00000000e+000 0.00000000e+000 0.00000000e+000 ... 0.00000000e+000
  0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000 0.00000000e+000 ... 0.00000000e+000
  0.00000000e+000 0.00000000e+000]
 [1.94881901e-060 1.07274946e-136 0.00000000e+000 ... 0.00000000e+000
  0.00000000e+000 4.80949774e-030]]
```

(a) Non negative matrix H



```
[[1.31730168e-006 7.95356334e-012 2.48753326e-069 ... 3.00781998e-004
  1.26408411e-008 1.00741662e-004]
 [2.81958652e-005 1.55763158e-013 9.72243441e-050 ... 3.99473082e-005
  3.03811649e-017 4.50182082e-008]
 [2.58831452e-005 1.45311638e-038 1.75728675e-051 ... 1.33783117e-005
  5.23868047e-012 6.02829440e-009]
 ...
 [3.56678373e-010 1.87341764e-061 7.43761411e-020 ... 3.73780758e-008
  3.75384163e-020 4.16244356e-008]
 [1.20940653e-007 8.27108160e-107 1.56944561e-035 ... 1.06692362e-007
  4.92245411e-035 5.65587222e-006]
 [2.16618481e-009 9.51418506e-048 7.85812236e-077 ... 1.14766514e-004
  7.27688956e-018 2.12957487e-005]]
```

(b) Non negative matrix W



```
[[4.08414691e+00 3.54589359e-01 8.49625087e-02 ... 2.27881190e+00
  9.48425815e-01 5.03191986e+00]
 [1.08466369e-02 4.99112025e-02 7.10373423e-02 ... 1.97166978e+00
  1.25835610e-01 1.37789187e-01]
 [3.83406649e+00 1.40509324e-09 4.83366274e-02 ... 1.94480875e+00
  1.00334957e-01 3.67310619e-03]
 ...
 [1.88352297e-07 2.26864741e-03 1.60010522e-12 ... 1.34738287e-06
  3.83214654e-04 1.40208602e-03]
 [1.88352296e-07 2.26864739e-03 1.60010494e-12 ... 1.34738265e-06
  3.83214654e-04 1.40208602e-03]
 [1.16873644e-07 8.31780126e-03 4.54651765e-05 ... 2.76001639e-03
  9.72038017e-08 6.40240358e-03]]
```

Figure 3.2: Approximated matrix V

3.2 NMF using Alternating Least Squares

Before seeing how this algorithm works, we need to know the concept of latent vectors in matrix factorization. Let k be a number which represents the relevant features of a matrix. The matrix factorization approximation of a k-attribute user can be in mathematical terms said to take a form of k dimensional vector say x_u. Similarly, an item i can be a k dimensional vector y_i. User u's predicted rating for the item i is the dot product of the k feature vectors.

$$\hat{r}_{ui} = X_u^T.Y_i = \Sigma x_{uk} y_{ki}$$

These two user and item vectors are called the latent vectors/low dimensional embedding. The k attributes are called the latent factors. Knowing this, our job is now to minimize the square of the difference between all ratings in the dataset and our predictions. The loss function thus obtained will be of the form

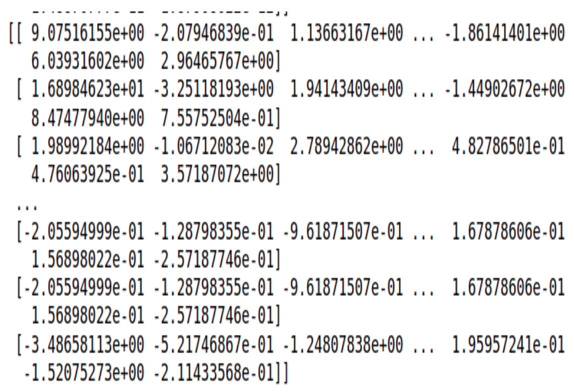
$$L = \Sigma(r_{ui} - x_u^T.y_i)^2 + \lambda_x \Sigma ||x_u||^2 + \lambda_y \Sigma ||y_i||^2$$

In ALS minimization, we'll hold one set of vectors constant and we take the derivative of the loss function of the other vector with respect to the constant vector and set the equation to zero so as to find the minima of the function. Now after this, we then hold these newly solved vectors constant and differentiate the loss function of the other vector. This way we alternate between the two latent vectors and eventually attain a minima.

3.2.1 Code Snippet for NMF using Alternating Least Squares

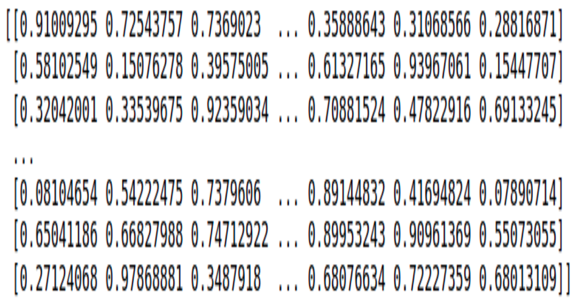
```
def als(latent, fixed, ratings, lambda, type):
    if type == 'movie': #for creating movie latent vector

        lambda_I = np.eye(fixed.transpose().dot(fixed).shape[0])*lambda
        for i in range(latent.shape[0]):

            latent[i, :] = solve(( fixed.transpose().dot(fixed) + lambda_I),
            ratings[i, :].dot(fixed))

    elif type == 'user': #for creating user latent vector

        lambda_I = np.eye(fixed.transpose().dot(fixed).shape[0])*lambda

        for j in range(latent.shape[0]):
            latent[j, :] = solve((fixed.transpose().dot(fixed) + lambda_I),
            ratings[:, j].T.dot(fixed))

    return latent

n_movies = 9762
n_factors = 610
n_users = 610
n_iter =  10
user_lambda = 0.0
movie_lambda = 0.0
#create a random non negative matrix for movies
movie_vec = np.random.rand(mat_array.shape[0],n_factors)
#create a random non negative matrix for users
users_vec = np.random.rand(n_users, mat_array.shape[1])
#zero matrix with dimension mxn
als_mat = np.zeros((mat_array.shape[0], mat_array.shape[1]))

for j in range(n_iter):
    movie_vec = als(movie_vec, users_vec, mat_array, movie_lambda, type='movie')

    users_vec = als(users_vec, movie_vec, mat_array, user_lambda, type = 'user')


for i in range(movie_vec.shape[0]):
    for j in range(users_vec.shape[0]):
        als_mat[i,j] = movie_vec[i,:].dot(users_vec[j,:].transpose())

print(als_mat)
print(movie_vec)
print(users_vec)
```

3.2.2 Results



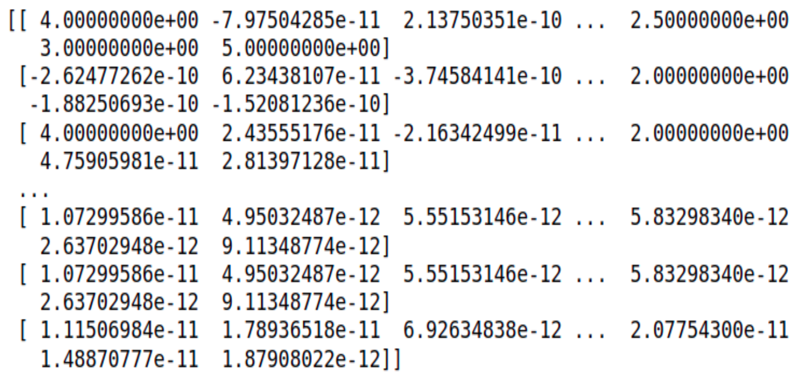
```
[[ 9.07516155e+00 -2.07946839e-01  1.13663167e+00 ... -1.86141401e+00
   6.03931602e+00  2.96465767e+00]
 [ 1.68984623e+01 -3.25118193e+00  1.94143409e+00 ... -1.44902672e+00
   8.47477940e+00  7.55752504e-01]
 [ 1.98992184e+00 -1.06712083e-02  2.78942862e+00 ...  4.82786501e-01
   4.76063925e-01  3.57187072e+00]
 ...
 [-2.05594999e-01 -1.28798355e-01 -9.61871507e-01 ...  1.67878606e-01
   1.56898022e-01 -2.57187746e-01]
 [-2.05594999e-01 -1.28798355e-01 -9.61871507e-01 ...  1.67878606e-01
   1.56898022e-01 -2.57187746e-01]
 [-3.48658113e+00 -5.21746867e-01 -1.24807838e+00 ...  1.95957241e-01
  -1.52075273e+00 -2.11433568e-01]]
```

(a) Movies matrix



```
[[0.91009295 0.72543757 0.7369023  ... 0.35888643 0.31068566 0.28816871]
 [0.58102549 0.15076278 0.39575005 ... 0.61327165 0.93967061 0.15447707]
 [0.32042001 0.33539675 0.92359034 ... 0.70881524 0.47822916 0.69133245]
 ...
 [0.08104654 0.54222475 0.7379606  ... 0.89144832 0.41694824 0.07890714]
 [0.65041186 0.66827988 0.74712922 ... 0.89953243 0.90961369 0.55073055]
 [0.27124068 0.97868881 0.3487918  ... 0.68076634 0.72227359 0.68013109]]
```

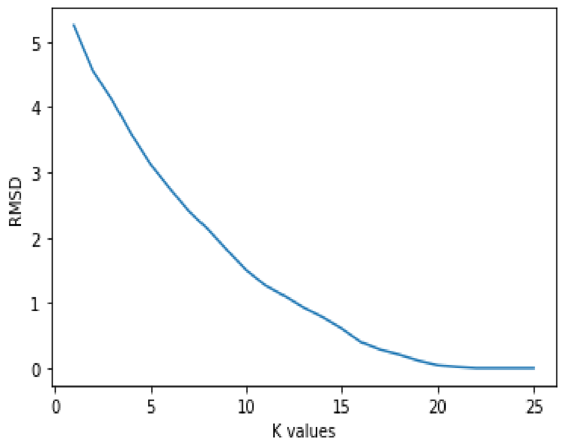
(b) users matrix



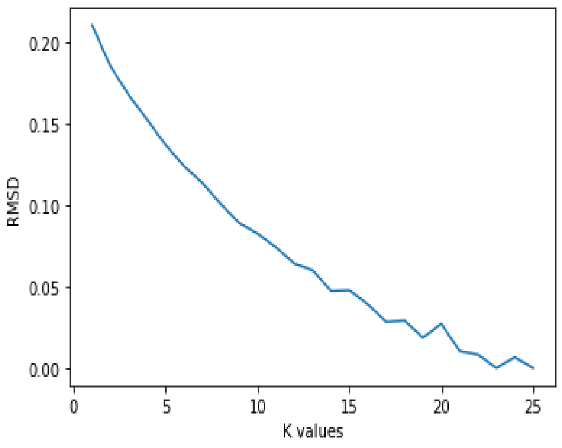
```
[[ 4.00000000e+00 -7.97504285e-11  2.13750351e-10 ...  2.50000000e+00
   3.00000000e+00  5.00000000e+00]
 [-2.62477262e-10  6.23438107e-11 -3.74584141e-10 ...  2.00000000e+00
  -1.88250693e-10 -1.52081236e-10]
 [ 4.00000000e+00  2.43555176e-11 -2.16342499e-11 ...  2.00000000e+00
   4.75905981e-11  2.81397128e-11]
 ...
 [ 1.07299586e-11  4.95032487e-12  5.55153146e-12 ...  5.83298340e-12
   2.63702948e-12  9.11348774e-12]
 [ 1.07299586e-11  4.95032487e-12  5.55153146e-12 ...  5.83298340e-12
   2.63702948e-12  9.11348774e-12]
 [ 1.11506984e-11  1.78936518e-11  6.92634838e-12 ...  2.07754300e-11
   1.48870777e-11  1.87908022e-12]]
```

Figure 3.4: Approximated matrix V

4 Comparison between three methods



(a) Error Plot for SVD



(b) Error Plot for NMF algorithm

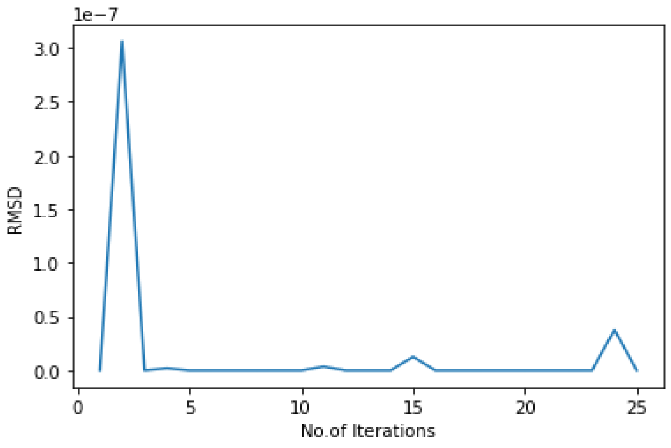


Figure 4.1: Error plot for ALS

We build a random toy ratings matrix S with a dimension of 25x25 to make the computations easy to obtain the following plots. Using this matrix S, we compare and contrast the relative error trend between the actual ratings matrix and one obtained after using the algorithm. RMSD vs K values plot for both SVD and NMF using Euclidean distance algorithms show a steep decrease and reached near 0 in the curve as the value of k increases. This is as expected because with increase in the rank parameter, we cover all the elements in the matrix and thus the probability of the newer matrix resembling the previous one will be higher. Whereas in the case of ALS, we are not able to arrive at any conclusions about the behaviour of the curve as it is completely random and dynamic. By changing the no.of iterations we can tinker around approximations.

5 Conclusion

Through this project, we were clearly able to appreciate how various matrix factorization algorithms can be employed to effectively give a movie recommendation system to a user based on his/her previous ratings and interest of genres. We have used three different algorithms in this project called the **Single Value Decomposition**, **Non negative matrix factorization using Euclidean distance** and **Non negative matrix factorization using Alternating Least square method**. The first method is based on factorizing the matrix in to three components whereas the rest two focuses on factorizing the main matrix in to two non negative sub matrices. By plotting the error functions for each of the three algorithms, we can clearly see the performance of the algorithms in real life. Matrix Factorization is not just limited in to recommendation systems, but can also be extended to represent many other types of dyadic data (relating two types of entities), such as a term-document matrix in a document corpus a pixel value-picture matrix for an image corpus.