
Universiteit Leiden

Leiden University

Advances in Data Mining

# Homework Assignment 1

*Authors:*

Group 32

Eric Prehn, s2724731

Chloe Gao

October 16, 2020

# 1 Introduction and Background Theory

The MovieLens 1m data set contains 1,000,027 movie ratings given by 6040 users to 3706 movies, this data set is used throughout the assignment and 3 different recommender system methods are trained and tested using 5-fold cross validation.

## 1.1 Rating Matrix

The rating matrix $M$ or utility matrix, stores the movie ratings given by each user. Below is an example of a rating matrix, the gaps represent movies that the user has not rated yet.

| | Movie 1 | Movie 2 | Movie 3 |
|---|---|---|---|
| User 1 | 3 | 4 | 5 |
| User 2 | 2 | | 5 |
| User 3 | 3 | 3 | 5 |
| User 4 | | 4 | 5 |

Table 1: Small utility matrix example

The aim of the recommender system is to predict the rating for the gaps in the utility matrix, and ultimately recommend the highest scoring gaps to the corresponding user. The utility matrix is usually sparse, following from the fact that the majority of users won't have seen/rated the majority of movies.

## 1.2 Cross Validation

This resampling procedure is used throughout this assignment, the data set is split into k-folds ($k = 5$) and each of the 5 splits is used as a test set, whilst the reminder of the set is used as the training set. Thus, each of the training sets (made up of $\frac{4}{5}$ of the data) and their corresponding test sets (made up of $\frac{1}{5}$ of the data) are evaluated and the average accuracy across all test sets is taken as a reasonable estimate of the accuracy of the final model. This final model could then be used on a new test set which has not been seen before.

## 1.3 Accuracy

The accuracy of the model in question is evaluated using the two metrics below, these metrics can be used to measure model accuracy's on both training and test sets.
**Root Mean Square Error:**

$$RMSE = \sqrt{\frac{\sum_i^n (M_i - P_i)^2}{n}} \tag{1}$$

Where $P$ is the predicted ratings matrix and the sum runs over all $n$ entries that are present in the matrix $M$ in question (can be test set $M$ or training set $M$).
Similarly for **Mean Absolute Error**:

$$MAE = \frac{\sum_i^n |M_i - P_i|}{n} \tag{2}$$

## 1.4 Recommender System Methods

### 1.4.1 Naive Approach

The 4 variations of the naive approach involve the following rating prediction methods:

- R_global(User, Item)=mean(all ratings)
- R_item(User, Item)=mean(all ratings for Item)
- R_user(User, Item)=mean(all ratings for User)
- R_user-item(User, Item)= $\alpha$R_user(User, Item) + $\beta$R_item(User, Item) + $\gamma$

Here R denotes rating, the first 3 methods involve taking averages across users, items or all ratings and using these for predictions.
The last approach combines the average user and average item ratings, via calculating the parameters $x = [\alpha, \beta, \gamma]^T$ using linear regression, that minimize:

$$|Ax - b|$$

Where A is a matrix made up of rows [R_user,R_item,1] and $b$ is the column of corresponding correct ratings.
The predicted rating is then given by the linear combination seen above, using the parameters $x = [\alpha, \beta, \gamma]^T$ that were obtained using the training set.

### 1.4.2 UV Decomposition

This method uses gradient descent to approximate the utility matrix $M$ as:

$$M \approx UV = P \tag{3}$$

Where M is of dimensions (n=6040,m=3706) , U of dimension (n=6040,d=2) and V of dimension (d=2,m=3706). The idea behind this method is conjecture that a small set of features of items and users determine the reaction of most users to items [1].
The entries of U and V are updated one at a time, such that the RMSE, along a column or row (update of a V entry or U entry respectively), between non-blank entries of M and P is reduced. The update rules are given by:

$$u_{r,s} \longrightarrow x = \frac{\sum_j v_{sj}(m_{r,j} - \sum_{k \neq s} u_{rk} v_{kj})}{\sum_j v_{sj}^2} \tag{4}$$

$$v_{r,s} \longrightarrow x = \frac{\sum_i u_{ir}(m_{i,s} - \sum_{k \neq r} u_{ik} v_{ks})}{\sum_i u_{ir}^2} \tag{5}$$

Where the sum over $i$ corresponds to summing over the non-blank column entries and summing over $j$ corresponds to summing over the non-blank row entries.

### 1.5 Matrix Factorisation

This method makes the assumption that every rating is a product of two feature vectors, such that:

$$M[i,j] = \vec{u}_i \cdot \vec{v}_j \tag{6}$$

Where $\vec{u}_i$ is the feature vector that represents User $i$ and $\vec{v}_j$ is the feature vector that represents Movie $j$. This way, you look at the data from both the perspective of the user and the movie. By gradient descent the vectors $\vec{u}_i$ and $\vec{v}_j$ are updated. The steps in the algorithm are: [2]

1. Initialise $\vec{u}$ and $\vec{v}$ at random
2. Run steps 3,4,5 until convergence
3. Calculate the error of training set entry $M[i,j]$: $E = Rating - \vec{u}_i \cdot \vec{v}_j$
4. Update: $\vec{u}_i = \vec{u}_i + L(E\vec{v}_j - \lambda\vec{u}_i)$
5. Update: $\vec{v}_j = \vec{v}_j + L(E\vec{u}_i - \lambda\vec{v}_j)$

The algorithm loops through non-blank entries of the training rating matrix until the RMSE converges. L is the learning rate and $\lambda$ is the regularisation term. $\lambda$ helps prevent overfitting and L determines how large a step in the direction of the steepest descent is taken for every update. Multiple epochs over the training set can be taken and are referred to as iterations.

### 1.6 Complexity and Sparse Matrices

How the run time grows with growing number of Users $n$ and Movies $m$ is an important question, since companies such as Netflix receive millions of ratings a day. An interesting property to note is that as $n$ and $m$ increase the utility Matrix $M$ will become increasingly sparse. This sparsity will help with computational efficiency with regards to cpu-time and memory. To see this, consider the 1M set:

The max number of entries of $M$ is $6040 \times 3706 = 22384240$ (all users have seen every movie), however in reality only $R = 1,000,027$ ratings (or entries) are present. So, less than 0.5% of the entries are filled. A User on average has watched 165 movies. Consider now, adding extra users $X$ and thereby movies $Y$:
From the law of large numbers the average number of movies watched by all Users remains of the order of 165. Therefore the number of ratings will increase by: O($165X$)=O(X), however the max number of elements of $M$ will increase by: $(n+X)(m+Y) - nm = O(nY + mX + XY)$ Therefore the matrix becomes increasingly sparse with growing dimensions.
Creating the rating matrices $M$ can be achieved by looping over the number of ratings $R$ and then efficient Numpy functions can act on $M$. By using Numpy functions on the sparse matrix representations created, the complexity of problems can be reduced:

For example: In Matrix Factorisation, looping over only the non-blank entries of $M$ for step 3 seen above, ( using efficiently calculated indices stored in an array), resulting in O(R) for all error calculations.

Therefore, throughout this assignment calculating the RMSE and MAE is of time complexity O(R) and memory complexity 0(1).

# 2 Procedures and Results

## 2.1 General Pre-Processing and Post-Processing

- The data set was converted to a Numpy array and throughout the assignment Numpy operations were used for efficiency.
- The indices of the Users in the data set run from 1 to 6040 and all 6040 users are present in the set, each with at least 20 ratings. The Movie indices run from 1 to 3952, however only 3706 movies are actually present in the data set. Therefore the movies were renumbered using a dictionary that uniquely mapped every original movie Id to within [1,3706].
- During the evaluation of a model on the test set, it is possible that, for example, the R_Item average for a Movie is missing since this movie was not present in the training set. The same could occur for R_User values. A fall-back rule was chosen that would set the value to be the R_global value in these cases. In an attempt to reduce these occurrences, the 5 sets were split such that every user was present in every set.
- The 5 training and their corresponding test sets were created, additionally the rating matrices M(n=6040,m=3706) for the 5 training sets were stored in a 3 dimensional ndarray. The gaps were filled with NaN values (to make use of Numpy nan functions such as nanmean() ). The memory required to store this 3d array is O(n,m), the cpu-time cost of creating and accessing elements of this array depends on the efficient Numpy functions such as argwhere(), isfinite() ,np.full() etc.
- Post-Processing: Due to the fact that movie ratings in this data set range discretely from 1 to 5 The predicted values resulting from any of the methods are squeezed into the range[1,5].

## 2.2 Naive Approach

The average RMSE and MAE scores following cross validation, along with the execution time taking the naive approach are given below in Table 2:

| Naive Approach Results | R_Global | R_User | R_Item | R_User_Item |
|---|---|---|---|---|
| RMSE: Train | 1.117 | 1.028 | 0.974 | 0.915 |
| RMSE: Test | 1.117 | 1.034 | 0.979 | 0.924 |
| MAE: Train | 0.934 | 0.823 | 0.778 | 0.725 |
| MAE: Test | 0.934 | 0.828 | 0.782 | 0.732 |
| Wall time: Test | 119 ms | 166 ms | 165 ms | 1.48 s |
| Wall time: Train | 120 ms | 674 ms | 665 ms | 1.48 s |

Table 2: Naive Approach Results

Note: The execution times in Table 2 are for one of the folds. Any predictions above 5 or below 1 were set to 5 and 1 respectively. There was a significant increase in accuracy with every new method taken. Overall the Naive Approaches worked surprisingly well.

**Naive Approach Complexity** The Complexities are given in Table 3, where 'Worst Case' correspond to a general approach involving looping over all relevant dimensions. The complexities in our case 'Script' are dependent on the Numpy library and exact answers could not be found by searching online, due to the way Numpy is implemented. The time complexity of a general least squares regression depends on the method chosen and there exist many.

| Naive Approach Complexities | Worst Case:Time | Worst Case:Memory | Script:Time | Script:Memory |
|---|---|---|---|---|
| Ratings Matrix $M$ | O(R) | O(NM) | O(R) | O(NM) or O(np.full(NM)) |
| Global Averages | O(R) | O(1) | O(R) | O(1) |
| User Averages | O(NM) | O(N) | O( N(np.nanmean(M) ) | O(N) |
| Item Averages | O(NM) | O(M) | O( M(np.nanmean(N) | O(M) |
| Regression: Creating Matrix $A$ | O(R) | O(3R)=O(R) | O(R) | O(3R)=O(R) |
| Regression: Parameters | O(Method?(R,3)) | O(3) | O(np.linalg.lstsq(R,3)) | O(3) |

Table 3: Naive Approach Complexities

## 3 UV Decomposition

The algorithm for UV Decomposition requires repeatedly accessing non-blank entries of the rows or columns of the rating matrix $M$. To avoid looping over this entire row or column for every single update, the indices of non-blank entries across rows or columns were calculated and stored in two arrays. The algorithm then only loops over the relevant indices for the column or row in question that is being summed over in equations (4) and (5).

Additionally, the order in which the entries of U and V are updated had to be chosen. A random permutation (seed=10) was created for every epoch and used by the UV algorithm.

The initial entries U and V matrices were initialised to be randomly drawn from a range $\sqrt{\frac{a}{d}} \pm \epsilon$, where d=2 and $a$ is the global average of the set in question, so that initially P entries are randomly centred about the global average.
To avoid overfitting the algorithm set the updated entry to be halfway between it's original and updated value, so that the dependency on the first entries updated was reduced. Also, after every epoch the RMSE was calculated and the algorithm 'converges' and stops when:

$$\text{RMSE}_{new} - \text{RMSE}_{old} < t \tag{7}$$

With $t$ being a chosen tolerance.

Thus, the results depend on the set in question, the tolerance $t$, along with the random seed and $\epsilon$ when initialising U and V.

In order to produce a good graph for choosing when to halt the process a new function for plotting was created that calculated the RMSE every 1000 updates:

One epoch corresponds to 19492 ( $2(6040 + 3706)$ ) updates. A small $\epsilon = 0.001$ results in an initial $P = UV$ matrix that is approximately composed of global averages, as can be seen in Figure 1, the initial RMSE matches that of the global averages Naive Approach. As the algorithm processes more updates the RMSE decreases rapidly until it reaches approximately 40,000 updates, where the

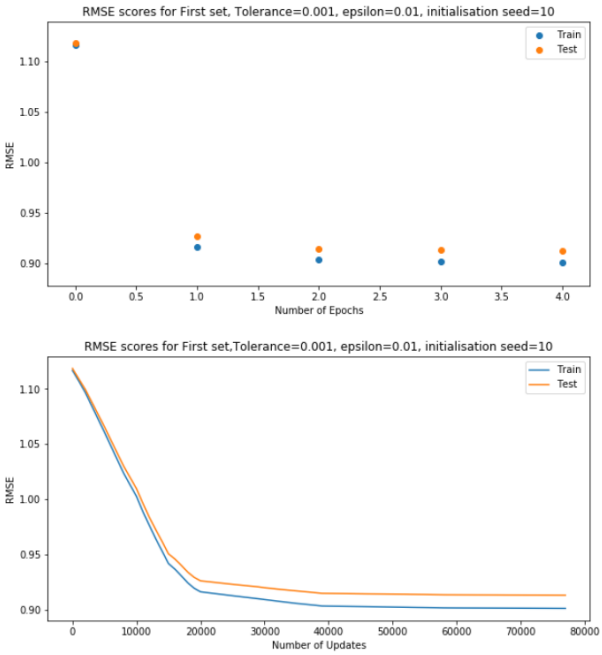


Figure 1: Graph of RMSE calculated values

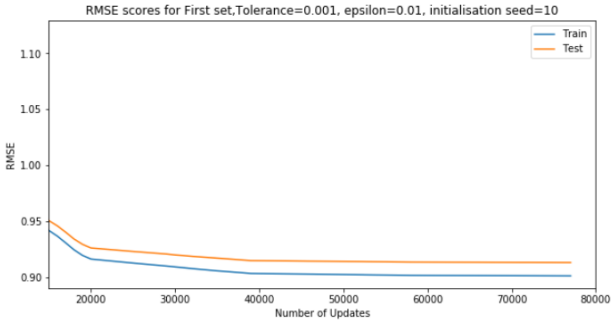


Figure 2: Graph investigating RMSE after 15000 updates

RMSE stabilises.
It was decided to choose a random seed=10, $\epsilon = 0.5$ and a tolerance $t$ of 0.001 for cross validation. The average results across the 5 sets are shown below:

| UV Decomposition Results | Train | Test | Execution Time per set |
|---|---|---|---|
| RMSE | 0.902 | 0.912 | 40s - 2 min |
| MAE | 0.844 | 0.849 | 40 s - 2 min |

Table 4: Cross Validation UV Decomposition Results ($t = 0.01$, $\epsilon = 0.5$, random seed=10)

The execution time is dependent on the number of epochs that occur, which is a result of the tolerance chosen. The execution times in Table 3 correspond to ranges of 3 to 5 epochs.

**UV Decomposition Complexity:**
The Memory Complexity involves storing the P matrix and is therefore O(NM).
Our method involved, for each row in $M$: storing the column indices that will need to be summed over in equation 4. This required O(M(np.argwhere(np.isfinite(N))) time and O($\frac{R}{N}$M)≈O(165M)≈O(M) Memory. And similarly
O(N(np.argwhere(np.isfinite(M))) time and O(N$\frac{R}{M}$)≈O(N) Memory, for the columns. See Section 1.6 for details.

This was done once. Then O(2N +2M)=O(N+M) updates take place, each summing over one of the arrays of indices created for the relevant column or row. The worst case scenario would be a user that has seen every movie , and then the sum would be over the entire row(3706 movies). However, over the entire set the approximate number of indices, corresponding to the Memory values above can be used. The resulting approximate time complexity per epoch is then:
O( M(np.argwhere(np.isfinite(N))) + N(np.argwhere(np.isfinite(M))) + 2NM +2MN)=O(NM)

### 3.1 Matrix Factorisation

The following parameters were chosen for running the Matrix Factorisation algorithm: [features=10, iterations=75, L=0.05, $\lambda$=0.005 and seed=10].

The algorithm randomly initialises 6040 user vectors $\vec{u}$ and 3706 movies vectors $\vec{v}$. Every entry of the feature vectors being approximately equal to $\sqrt{\frac{a}{10}}$, similarly to in UV decompostion, this results in the initialisations that will approximately yield the global average. Thus, any movie vectors not updated in training that appear in testing will obey the fall back rule.
The algorithm loops over all non-blank entries, updating featrure vectors as described in Section 1.5. This was repeated 75 times, the RMSE and MAE were calculated after the 75 iterations for every set and cross validation averages are calculated. The matrix factorisation results are shown on the next page in Table 5 and display an substantial increase in accuracy.

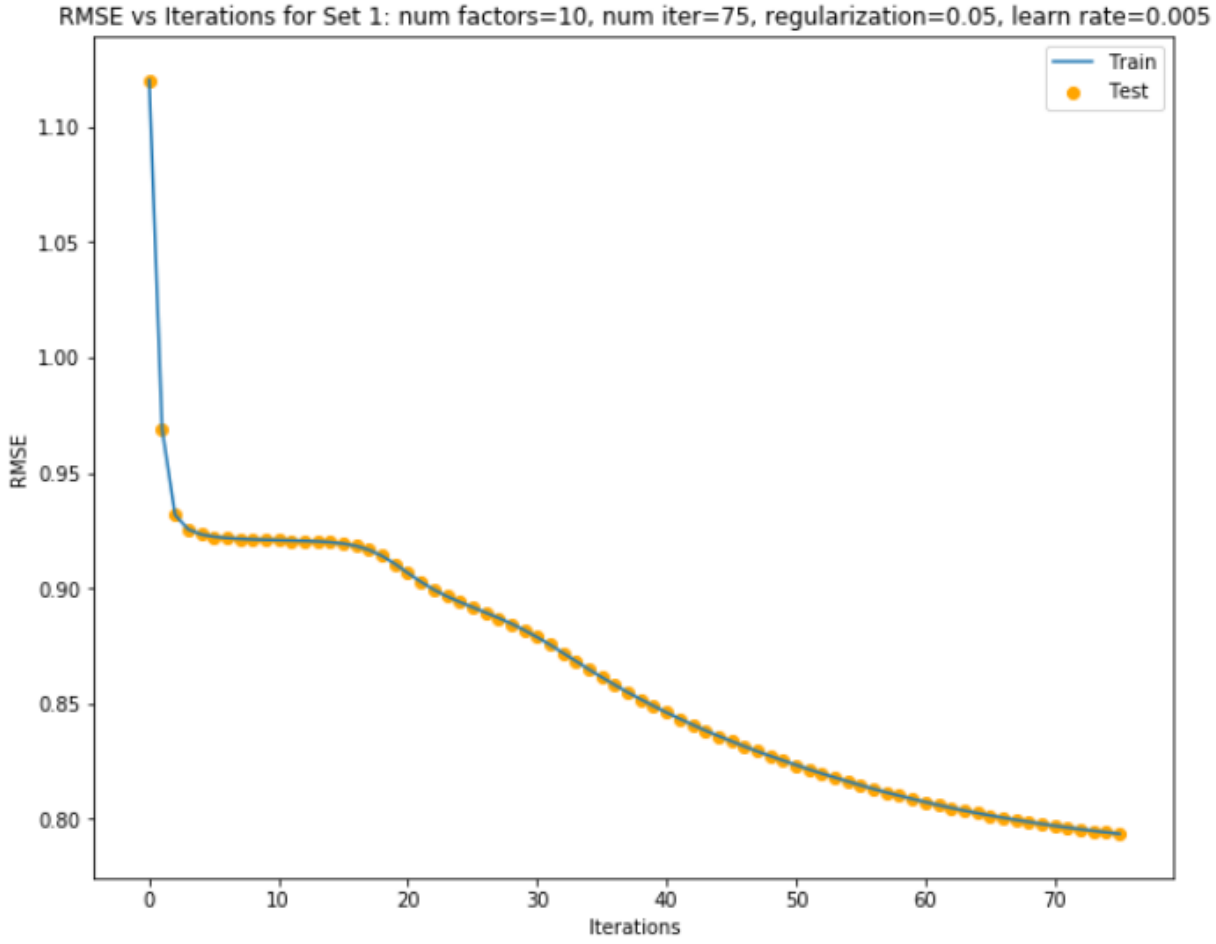


Figure 3: RMSE vs Iterations Set 1

| Matrix Factorisation | Train | Test | Execution Time for all sets (min) |
|---|---|---|---|
| RMSE | 0.794 | 0.794 | 70 ± 2 |
| MAE | 0.793 | 0.793 | 70 ± 2 |

Table 5: Cross Validation Matrix Factorisation Results

To produce Figure 3, the algorithm is edited to calculate the RMSE of a test and training set after every iteration. This increases the run time from 13 to 20 minutes. As can be seen in Figure 3 , the RMSE starts off at approximately the same value as for the global averages Naive Approach and then begins to decrease. After 70 iterations the RMSE appears to slow its decrease and may consequently increase after 75 iterations, suggesting that 75 iterations is a good input parameter.

It has to be noted that the results are better than the results found in [4], this is a cause of concern and perhaps the script was not calculating RMSE and MAE scores correctly. However, a problem with the calculatios of RMSE and MAE could not be found following inspection. We therefore cautiously present our results in Table 5.

**Matrix Factorisation Complexity:**
The memory is of complexity O(10N+10M)=O(N+M), from storing the $\vec{u}$ and $\vec{v}$ sets.
Our method loops over all entries O(R) and vector operations are applied, this process occurs for 75 iterations. Per iteration the time complexity is:
O( R(function(features)) ), where function(features) is a some combination of the cost of applying np.dot along with vector additions and multiplications.

## 4 Conclusion and Discussion

The conclusion can be made that Matrix Factorisation outperformed the Naive Approach and UV Decomposition methods with regards to accuracy. The downside of Matrix Factorisation is the longer computational time. However, if the results for Matrix Factorisation are indeed correct, then they are excellent scores.

In hindsight additional pre-processing of the utility matrices should have been included for UV Decomposition by means of normalisation discussed in MMDS chapter 9 [4]. The UV decomposition should also have been run over many more epochs, and this should have been repeated for different initial random configurations of $U$ and $V$, as our model may have converged in a local minimum rather than the desired global minimum. Lastly, Scipy.sparse matrices should be implemented next time for improved operations and memory storage of the large sparse utility matrices.

## 5 References

[1] Course Lecture Notes: AiDM_Recommenders'
[2] https://www.cs.uic.edu/~liub/KDD-cup-2007/proceedings/gravity-Tikk.pdf
[3] http://mymedialite.net/examples/datasets.html
[4] http://infolab.stanford.edu/~ullman/mmds/ch9.pdf