Project 1 - Matrix Factorization and Collaborative Filtering

Course: Mathematics for Machine Learning - 2^{nd} Semester 21/22

Telmo Cunha[†] (73487)

Master's in Mathematics and Applications - Instituto Superior Técnico

Contents

[†]Email: telmocunha@gmail.com

1. Introduction

In this project the intention is to build a movie recommendation system via collaborative filtering, i.e. a system which determines relationships between users and movies, in order to find new user-movie associations from known movie ratings. The latent model approach is used, where ratings will be dependent on a certain number of (latent) factors. These factors might be associated to particular genres (action/romance/sci-fi/etc.) or if the movie is geared toward children for example. Most likely however they represent a combination of factors which may not be easy to interpret. For users, the values on the corresponding factor measure how the user aligns with that particular feature, if the value is high the user likes aligns with the factor and if the value is low the user does not align with the factor [3].

There are three distinct implementations, the first system relies on **singular value decomposition (SVD)**, the second on a **non-negative matrix factorization** algorithm from [1] and the third on **non-negative alternating least squares** [4].

The dataset to be used is the "Movie Lens Small Latest Dataset"[1] which is extensively described in [2]. For our purposes it is useful to know the following facts: the ratings fall in a scale of 0.5 - 5.0 with possible steps of 0.5 and there are a total of 100836 ratings for 610 users distributed among 9724 movies.

2. Constructing the Ratings Matrix

The first goal is to build a matrix R of size (m, n) where m is the number of rated movies (9724 for this particular dataset) and n is the number of users (610 for this particular dataset). This means that the entry R_{ij} represents the rating of movie i as evaluated by user j. This will be a largely sparse matrix since we have $\frac{100836}{610\times 9724} = 1.7\%$ of the total possible ratings. The construction of the matrix is described in the following steps:

- First the data is imported as a list of strings, which is done with the function **read_csv()**. Each element of the list is a string of the form "user_id, movie_id, rating, timestamp" except for the first line which contains the data string "userId,movieId,rating,timestamp".

- Using the **delete_header()** function the first line is removed from this list, containing "userId,movieId,rating,timestamp".

- Next, the values are extracted from the each line by using the **line_split()** function, the output is a list of lists, call it *data_list*, where each element is a string corresponding to the values of interest. Thus, *data_list*[i][j] corresponds to the data point i and tag j, where $j \in \{0, 1, 2\}$ (we ignore the *timestamp* j=3 since there won't be a need for it), where $j = 0$ corresponds to the *user_id*, $j = 1$ to the *movie_id* and $j = 2$ to the *rating*.

- The function **counting_user_ids()** obtains the number of unique user ids, this is done in order to avoid hardcoding the actual number of users and obtain code which can be used on different datasets. In this particular case all user ids, from 1 to 610, are present so there is no need to take special care.

- To find the number of unique movie ids the function **counting_movie_ids()** is used. It appends to a starting empty list all unique movie ids found in the dataset. Then this list is sorted in increasing order obtaining a correspondence between the index of the list and the movie id. An example, suppose we have ratings of the movie ids $\{12, 7, 19, 3\}$, we sort the list $\{3, 7, 12, 19\}$. Now, index 0 on this list corresponds to movie 3, index 1 to movie 7 and etc. This will be the order on our ratings matrix R with the first line (of index 0) corresponding to movie 3, the second line to movie 7 and so on. This is particularly important so we can later on recover the titles based on the movie id stored on this list.

- Finally, the function **generate_R_matrix()** generates the matrix R, where R_{ij} corresponds to the rating of user j to movie i. This function has an optional *value* argument which fills the unknown

[1] https://www.kaggle.com/datasets/shubhammehta21/movie-lens-small-latest-dataset

entries of the matrix i.e. the unknown ratings, with that specific *value*. This value is defined from now on as the *filling value* $\equiv f$.

- To use on f the average movie ratings the function **movie_avg()** is built. It performs the following, for all unknown ratings u of movie i we have $R_{(i,u)} = \frac{\sum_{j=1}^{n} R_{ij}}{n}$, where n is the number of users who rated that particular movie. For nuances regarding the particular implementation one can read the comments in the code.

3. Constructing Training & Test Sets

To construct training and test sets the function **build_sets()** is built. It receives the original list of data after **line_split()** and a float *percentage* corresponding to the fraction of data we want to assign to a test set (by default this value is 10%). It works in the following way:

- First it generates a $len(data) \times percentage$ number of distinct integers, using the **random** library, from a maximum of 100836 which corresponds to the number of data points.

- Next, the data points in the format "userId,movieId,rating,timestamp", corresponding to those integers, is assigned to a new list named *test_set*.

- All those data points which were not assigned to the *test_set* are then appended to a new list, the *train_set* list, containing the training data. Just a remark, here one should be careful and create a copy of the original list so as to not modify the list that is simultaneously being searched on.

- Finally these lists are converted into matrices by using the **generate_R_matrix()** function, described above, but now passing as arguments the *test_set* list to generate the test matrix (*R_test*), containing the test data, and passing *train_set* to generate the training matrix (*R_train*), containing the training data.

4. Singular Value Decomposition (SVD)

4.1. SVD: Factorization

Having constructed the ratings matrix R, the first recommendation system will be based on singular value decomposition (SVD). The SVD, which is applicable to any matrix, factorizes the matrix $R_{m\times n}$ in the form $R = M\Sigma U^T$ where, in the full SVD, we have $M_{m\times m}$, $\Sigma_{m\times n}$ and $U^T_{n\times n}$. However, if we are interested in a rank k approximation to R we consider $R \approx M_{m\times r}\Sigma_{r\times r}U^T_{r\times n}$ which, by the Eckart–Young–Mirsky theorem, is known to be the best rank k approximation to R in the following norms (which we will consider):

$$\begin{aligned} \|R\|_F &= \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|R_{ij}|^2} = \sqrt{\sum_{i=1}^{\min\{m,n\}}\sigma_i^2(R)} \quad \text{(Frobenius norm)} \\ \|R\|_* &= \text{trace}\left(\sqrt{R^TR}\right) = \sum_{i=1}^{\min\{m,n\}}\sigma_i(R) \qquad \text{(Nuclear norm)} \end{aligned} \tag{4.1}$$

where the σ_i corresponds to the singular values of R.

In this section the following functions were implemented:

- The **svd()** function just computes the SVD by using the *numpy* library function **np.linalg.svd()** and takes a matrix as the argument. This function returns the matrices M, Σ and U^T. Note that Σ is returned just as a list containing the singular values in decreasing order and not as a diagonal matrix.

- The rank k approximation is given by:

$$R \approx \sum_{i=1}^{k} \sigma_i m_{(:,i)} u^T_{(i,:)} \tag{4.2}$$

using the python notation to represent the column vector $m_{(:,i)}$ and the row vector $u^T_{(i,:)}$. To obtain this approximation we define the function **svd_dimreduction()**, which cuts the matrix M up to column k, the matrix U^T up to row k and retains only the first k singular values in Σ. The function **rk_reconstruct()** is also defined and performs the calculation in (4.2).

- The function **calculate_energy()** determines the rank k which explains a certain fraction of the data, taken as input (*energy_value*), i.e. finds k such that:

$$k := min\left\{ k \in \mathbb{N} \,|\, energy\_value \leq \frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{n} \sigma_i} \right\} \tag{4.3}$$

4.2. SVD: Reconstruction Errors

In this section the reconstruction errors are calculated considering the two norms of equation (4.1) and for different *filling values* f (for brevity we consider ($f = 0.5$) and ($f = movie\_avg$) only).

The procedure is the following, first obtain R_{train}, with 90% of the data, and R_{test} with 10% of the data by using the **build_sets()** function. Next, compute the SVD on R_{train} obtaining $R_{train} = M_{train}\Sigma_{train}U^T_{train}$. Then, calculate $R_{train}(k)$ by using equation (4.2), where k is the rank of R_{train} (in the code this corresponds to the function **rk_reconstruct()**). Finally, compute the norms for each value of k. This is all implemented using the functions **svd_convergence_k()** and **testset_comparison_k()** which return dictionaries with the value of k and the associated norm calculation. A remark, when considering the norm with respect to the test data (R_{test}) we only compare the actual entries assigned to the test set, this will always be the case.

- $\|R_{train}(k) - R\|_F$ **and** $\|R_{train}(k) - R_{test}\|_F$ **for** $f = 0.5$**:**

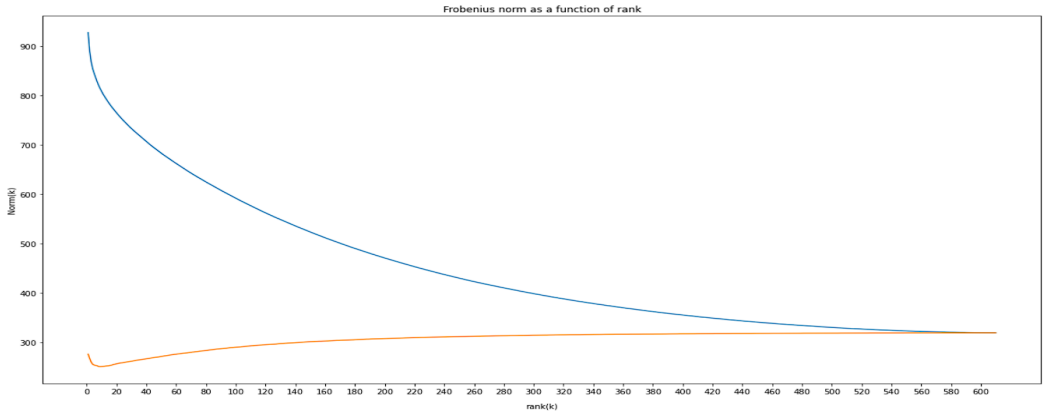

Figure 1.: $\|R_{train} - R\|_F(k)$ (blue line) and $\|R_{train} - R_{test}\|_F(k)$ (orange line)

The result is as expected, the blue line, measuring $\|R_{train} - R\|_F(k)$, simply shows that an higher rank k corresponds to a closer approximation of the original matrix R, the one containing all the data. This is precisely what SVD does, i.e. computes $R_{train}(k)$ which coincides with R on 90% of the data when $k = \min\{m, n\}$. The minimum value obtained here is precisely the error on those entries which were removed when constructing R_{train} and were assigned to the R_{test} matrix. The minimum value of the norm (N_{min}), obtained at rank $k = 610$, was $N_{min} = 319.1$ (which would coincide with $\|R_{test}\|_F$ had we used $f = 0$).

The orange line, measuring $\|R_{train}(k) - R_{test}\|_F$, shows something slightly more interesting. First, as expected, the value of the norm at rank $k = 610$ coincides with the minimum on the blue line. This is due to the comparison being only on the entries assigned to the R_{test} matrix and that, for maximum rank, we have a perfect approximation on the training data via SVD. Now however, there is a minimum for the norm around $k = 9$ with $N_{min} = 250.6$. This reflects the fact that the larger singular values (corresponding to the first k's) capture the nature of the data better than simply minimizing the difference of the train set with the original data, which converges on the values f we defined for the unknown values. Similarly we could say that higher values for k are essentially overfitting the data.

- $\|R_{train}(k) - R\|_*$ **and** $\|R_{train}(k) - R_{test}\|_*$ **for** $f = 0.5$**:**

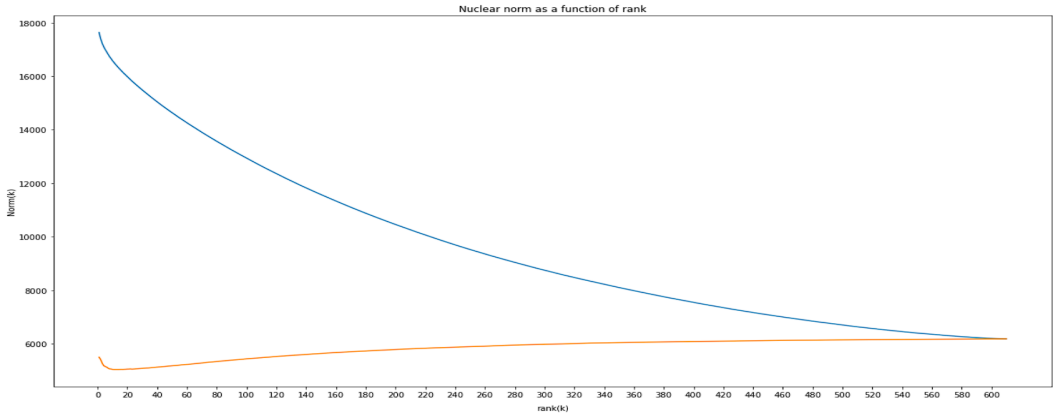

Figure 2.: $\|R_{train} - R\|_*(k)$ (blue line) and $\|R_{train} - R_{test}\|_*(k)$ (orange line)

The results obtained here are essentially the same. Since the singular values are always non-negative the behaviour does not change in comparison to the Frobenius norm.

Using movie averages as *filling values*:

Considering the same evaluation as before but with ($f = movie\_avg$) the following results were obtained (only using the Frobenius norm for brevity) :

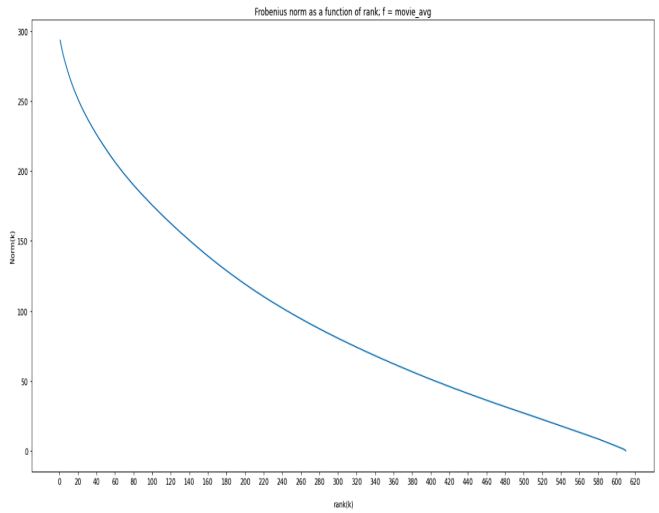

Figure 3.: $\|R_{train} - R\|_F(k)$

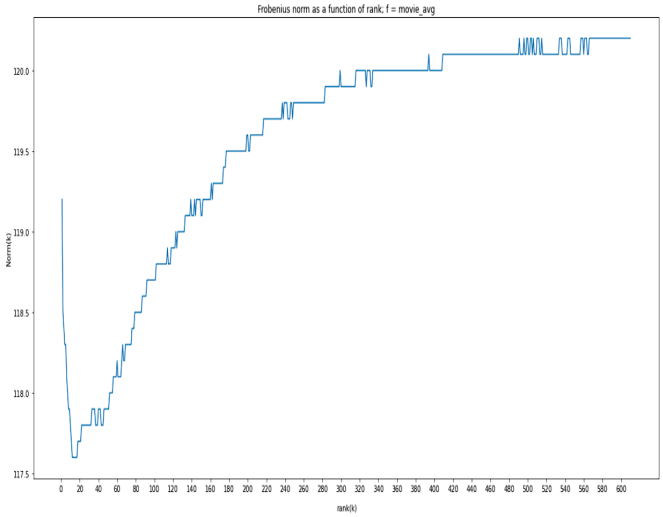

Figure 4.: $\|R_{train} - R_{test}\|_F(k)$

The left plot shows convergence of the rank k approximation to the training matrix, with the norm difference going to 0 as $k \to 610$. The norm on the line on the right does not change significantly as a function of k (it lies in the set $(117.5, 121)$), this suggests that taking the average ratings is actually a decent

approach to predict unknown ratings.

The following table shows the k values for $N_{min} \equiv \min_k \|R_{train} - R_{test}\|_F(k)$ for five different pairs of training and test data:

Test #	min k	N_{min}
1	11	119.5
2	11	119.7
3	11	121.9
4	11	116.3
5	14	120.3

On average we have $k \approx 11.6 \pm 1.3$ with a norm $N_{min} \approx 119.5 \pm 2.0$ for the Frobenius norm. This means that a value of $k = 12$ and ($f = movie\_avg$) is ideal to build the recommendation systems, presented in the next section.

4.3. SVD: Recommendation System

Important remark: Since columns on '.csv' files are split by commas, and some movie names on the file 'movies.csv' have commas on their names, we made the change (','→'.') directly on the 'movies.csv' file to avoid problems parsing the data. Example: ("Matrix, The (1999)" → "Matrix. The (1999)").

The following two types of recommendation systems were implemented:

- **Recommendation by movie**: Here the input is a particular movie id and the top ten movies (which would be most similar to it in the latent space) are shown, according to the model being considered.

- **Recommendation by user**: In this case the input is an user id. Out of the movies which the user has not seen, ten movies which would be better rated by this user are shown, according to the considered model.

4.3.1. Recommendation System by Movie

Here a function called **recommend_movies_based_on_movie_svd()** is defined and receives as input the id of the movie we are interested in comparing (in practice we built it to also receive the matrix of data (*R_matrix_avg* or *R_matrix_value*) and the list which converts the index of the rows of the matrix to the actual movie ids *unique_movies_list*). The breakdown is as follows:

- **SVD:** First we perform the SVD factorization of the full R matrix, with rank $k = 12$ and ($f = movie\_avg$), since this was the best configuration found when comparing with test sets on the previous section.

- **Similarity function:** An auxiliary function **cosine_sim()** is built which, given vectors (u, v), returns $cos(\theta) = \frac{<u,v>}{\|u\|\|v\|}$. This essentially measures how vectors corresponding to different movies align in the latent space of dimension k. An higher value implies that the movies are similar, in particular if we use as inputs (u, u), i.e. vectors corresponding to the same movie, we obtain $cos(\theta) = 1.0$.

- **Notes on implementation:** An empty list with two columns is created, (sim), which stores on the first column the similarity values obtained from **cosine_sim()** and on the second column the movie id to which we are comparing our input. This requires the *unique_movies_list* list to retrieve the correct movie ids from the row indices. Next both columns are sorted using the column similarities, sorted in decreasing order. Finally, one can obtain the movie names, from the "movies.csv" file by using the movie ids of the second column, and the calculated similarities.

A particular example of this implementation is now shown with $k = 12$ and ($f = movie\_avg$). The particular choice of the movie "Lord of the Rings: The Fellowship of the Ring"(movie id=4993) is due to the fact that it has two sequels and allows one to gauge if those are being recommended highly, which would be ideal

since they should essentially have the same features (in terms of genre, target audience, etc.). This is true provided the sequels were of the same "quality" (quality here being a general feature which encompass several variables, i.e. a sequel could be a terrible movie compared to the first even if similar in genre, target audience, etc.).

```
Recommendations similar to: Lord of the Rings: The Fellowship of the Ring. The (2001)

1 ) Lord of the Rings: The Two Towers. The (2002) Similarity: 0.96
2 ) Lord of the Rings: The Return of the King. The (2003) Similarity: 0.88
3 ) Memento (2000) Similarity: 0.85
4 ) Lemony Snicket's A Series of Unfortunate Events (2004) Similarity: 0.84
5 ) District 9 (2009) Similarity: 0.82
6 ) Number 23. The (2007) Similarity: 0.82
7 ) Seven (a.k.a. Se7en) (1995) Similarity: 0.8
8 ) Zootopia (2016) Similarity: 0.78
9 ) Rise of the Planet of the Apes (2011) Similarity: 0.78
10 ) Iron Man 3 (2013) Similarity: 0.76
```

As desirable, sequels come recommended in first place. The remaining recommendations are mostly in the genre of sci-fi, fantasy and thrillers. Some options are questionable in terms of comparing genres alone ("Adventure-Fantasy" vs "Action-Mystery-Thriller") but we can not actually know what the latent space is measuring. Furthermore, the model is of course dependent on the popularity of specific movies among available users.

4.3.2. Recommendation System by User

For this type of recommendation system a function called **recommend_movies_based_on_user_svd()** was built. It essentially receives as input the ID of the user we are interested in. In practice however it also receives the matrix of data (*R_matrix_value/R_matrix_avg*), (*R_matrix_orig*) to identify which movies were not seen already by the user and (*unique_movies_list*) for the same reasons as before. The breakdown is as follows:

- **SVD:** Again it starts by performing the SVD factorization of the full R matrix with rank $k = 12$ and ($f = movie\_avg$) but now we also compute the reconstructed rank k approximation R_k.

- **Predicted ratings:** Since the $[R_k]_{ij}$ entry has the rating of user j to movie i one needs to find, on the column corresponding to the input user id, the largest values out of those movies which were not rated by that user. This is the reason why (*R_matrix_orig*) is also provided, which contains zeros on all movies which were not rated.

- **Notes on implementation:** Much like before an empty list with two columns is created, (*ratings_movieid*), which stores on the first column the R_{ij} values of input user j and on the second column the corresponding movie id, obtained using (*unique_movies_list*). Finally, sort the list based on the columns rating and print the corresponding movie names and expected rating for the top 10 values. Note that it is possible to have an expected rating larger than 5 since there were no constraints on the factorization.

We now show a particular example of this implementation. We chose *user_id*=134 and obtained the following:

```
Expected ratings for user: 134
1 : Shawshank Redemption. The (1994) -> Expected rating: 4.42
2 : Streetcar Named Desire. A (1951) -> Expected rating: 4.27
3 : Godfather. The (1972) -> Expected rating: 4.26
4 : Fight Club (1999) -> Expected rating: 4.24
5 : Godfather: Part II. The (1974) -> Expected rating: 4.22
6 : Goodfellas (1990) -> Expected rating: 4.22
7 : Three Billboards Outside Ebbing. Missouri (2017) -> Expected rating: 4.22
```

```
8 : Usual Suspects. The (1995) -> Expected rating: 4.22
9 : Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964) -> Expected ratin
10 : Secrets & Lies (1996) -> Expected rating: 4.21
```

The obtained recommendations make sense since they are mainly comprised of extremely well known movies which are also highly rated typically. The movies "The Shawshank Redemption", "The Godfather" and "The Godfather: Part II" for example are in the top 5 highest rated movies on IMDB. The others are also highly rated as well (8.0+ on IMDB).

4.4. SVD: Latent Space Projections

For the latent space projection one just needs to recover the entries of a particular movie. For example, if we choose $k = 2$ and perform the SVD $R = M\Sigma U^T$ then M is a $(9724, 2)$ matrix. Then, the entries $M[m][0]$ and $M[m][1]$ are precisely the projections of movie m on the latent space of dimension 2 corresponding to the first and second singular values respectively.

Two functions were implemented, the first, **latent_space_proj_svd()** chooses n random movies and outputs two lists, one containing the movie coordinates over the latent space and the other the corresponding movie id. The second function **latent_space_proj_svd_test()** basically does the same thing as the previous one but for specific movies that the authors selected. This allows one to possibly better interpret the results over known movies instead of random ones.

The plots were obtained considering $k = 12$ and $(f = movie\_avg)$. For 10 randomly chosen movies and 10 movies chosen by the authors the results obtained were the following (the movie names might be a difficult to read, apologies, this was done to save space):

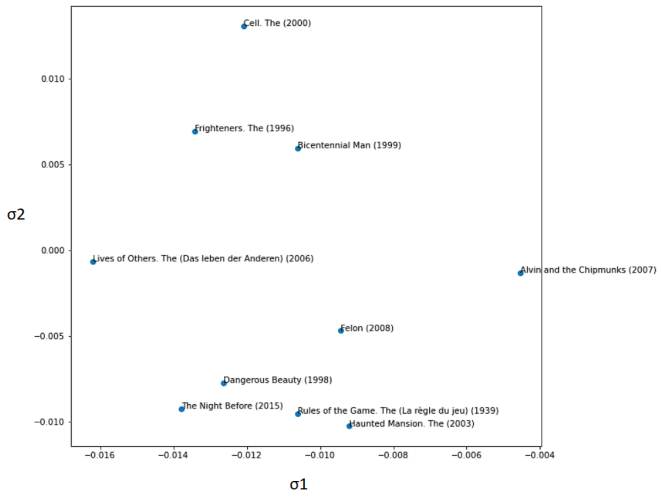

Figure 5.: Latent space proj. for 10 random movies

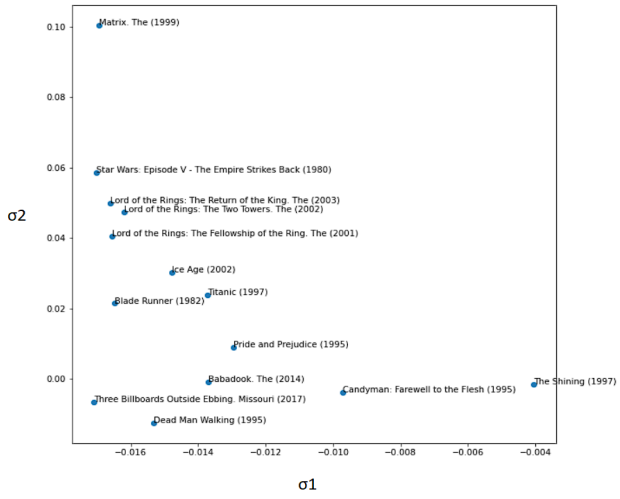

Figure 6.: Latent space proj. for 14 specific movies

On fig. 6 one can see a few fantasy, sci-fi, horror, drama/romance, comedy/animation and non-fiction movies plotted. However, it is quite difficult to infer, based on these results, what the factors may actually mean. Nonetheless, the Lord of the Rings trilogy is in the same region on the plot which means they agree on both of the latent factors, which is a desirable result.

For 100 randomly chosen movies and 1000 randomly chosen movies the results obtained were the following (movie names are not printed otherwise it is a mess):

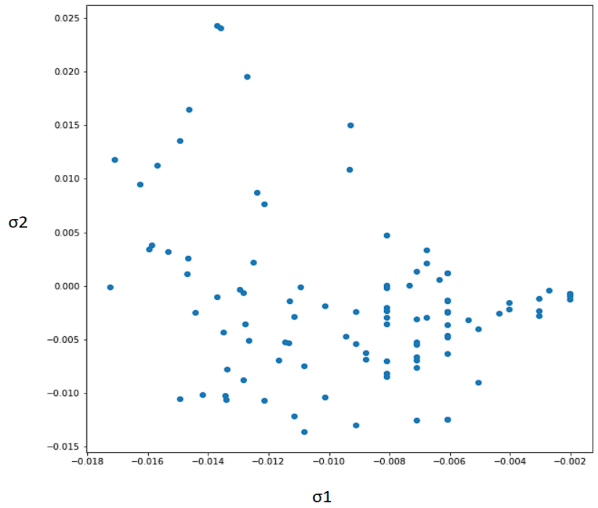

Figure 7.: Latent space proj. for 100 random movies

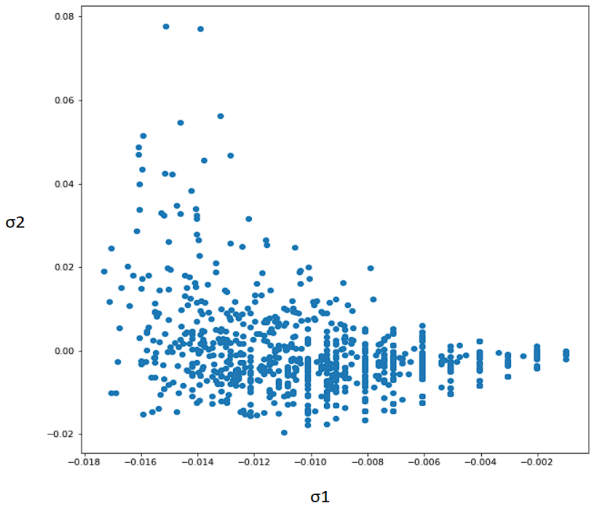

Figure 8.: Latent space proj. for 1000 random movies

There is a somewhat even dispersion over the first singular value direction but not over the second, with most movies being placed on the lower left region of the plot.

5. Non-negative Matrix Factorization

Since the original ratings matrix has only non-negative values it makes sense to construct an approximated factorization for R using only matrices with non-negative entries. First, we describe and use the algorithm found in [1], which we call henceforth Algorithm 1, and perform the same evaluation as in the SVD case, i.e. computing reconstruction errors, building a recommendation system and projecting on latent space. After, we shall use and describe the non-negative alternating least squares (NNALS) method and the corresponding results. Since the recommendation methods and the latent space projection are all similar, except considering the new factorization, we will not describe them again.

5.1. Algorithm 1: Non-negative Matrix Factorization

If R is a $m \times n$ matrix the idea is to pick $k < min\{n, m\}$ and factor $R \approx MU$, where M is a $m \times k$ matrix and U is a $k \times n$ matrix, such that we obtain a "compressed version of the original matrix" R (with k representing the dimensions of the latent space). The implemented algorithm, under the function name **nnmf()** is the following:

Algorithm 1 Non-negative matrix factorization for $R \approx MU$

Input: R_{train}: Training data in matrix format; k: rank of approximation used;
Output: MU: Approximated matrix;
Initialization: U,M: Initialized as random matrices with values in range $[0, 1]$; error$= 10$; $\epsilon = 1$;
$R_0 = MU$
while *(error*$> \epsilon$*)* **do**
  $U_{ij} = U_{ij} * (M^T R)_{ij}/(M^T MU)_{ij}$
  $M_{jk} = M_{jk} * (RU^T)_{jk}/(MUU^T)_{jk}$
  $R_1 = MU$
  $error = \|R_1 - R_0\|_F$
  $R_0 = R_1$

By theorem 1 in [1], the norm $\|R - MU\|_F$ is non-increasing under the update rules used in **algorithm 1**. Furthermore, this norm is invariant if and only if M and U are at a stationary point of the distance, which could however be a local minimum since the problem in non-convex as stated also in [1], section 3. To perform the calculations in a parallel way, instead of using for loops, the implementation is done

using the numpy library, in particular the element wise division **np.divide()** and element wise product **np.multiply()** are used. We should remark that there are some problems with the division step, often resulting in *nan* entries. Because of this, at each step all *nan* entries are replaced by zeros which could have the problem of the algorithm no longer respecting theorem 1 of [1].

5.2. Algorithm 1: Reconstruction Errors

In this section we compute the reconstruction errors just as we did before for the SVD. For this we use the function **nnmf_test()**. The blue line again represents $\|R_{train}(k) - R\|_F$ while the orange line shows $\|R_{train}(k) - R_{test}\|_F$. We should remark now that the calculations were made using rank jumps of 1, for $k \in [1, 50]$ with subsequent jumps of 10 for $k \in [50, 610]$. This is due to the fact that the algorithm takes quite some time to converge for each k, for $k \gtrsim 200$ each value of k takes between 60 to 200 seconds to compute.

- $\|MU - R_{train}\|_F$ **and** $\|MU - R_{test}\|_F$ **for** $f = 0.5$**:**

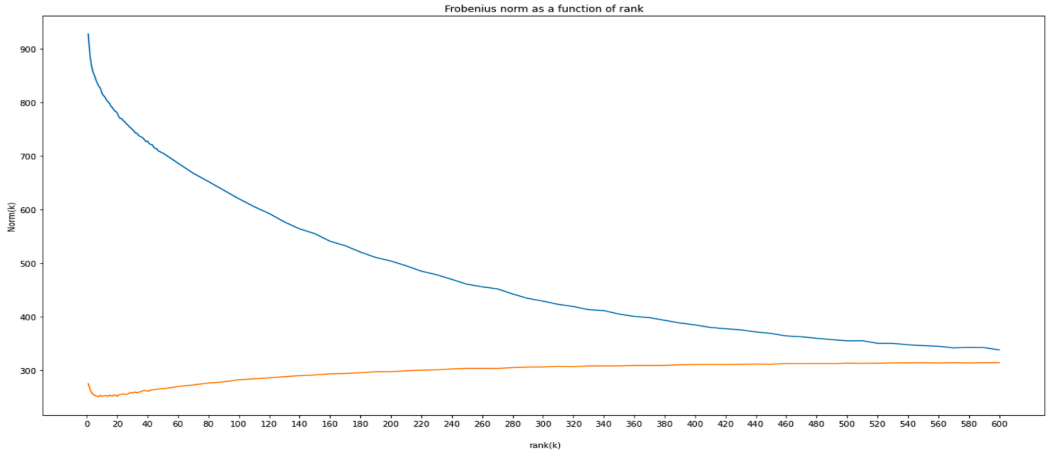

Figure 9.: $\|R_{train} - R\|_F(k)$ (blue line) and $\|R_{train} - R_{test}\|_F(k)$ (orange line)

- $\|MU - R_{train}\|_*$ **and** $\|MU - R_{test}\|_*$ **for** $f = 0.5$**:**

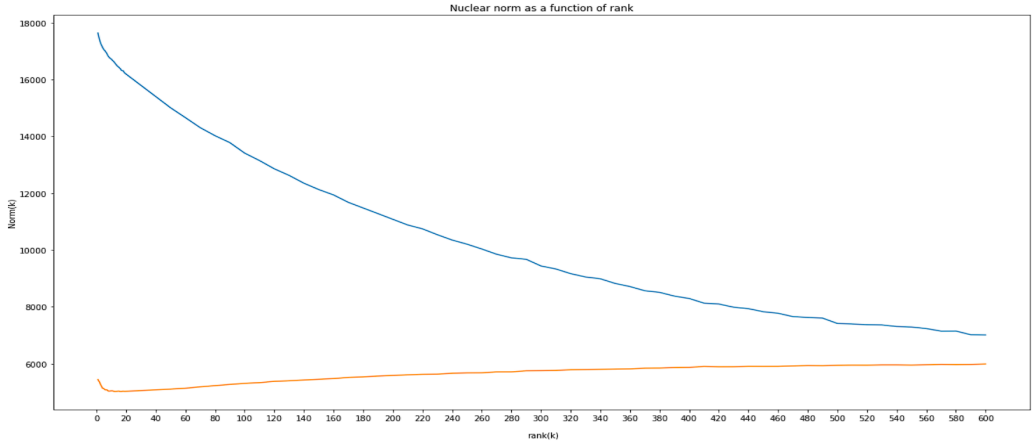

Figure 10.: $\|R_{train} - R\|_*(k)$ (blue line) and $\|R_{train} - R_{test}\|_*(k)$ (orange line)

The behaviour is pretty similar to what we had previously but now the two lines do not meet as $k \to 610$, this reflects the fact that this factorization will not perfectly approach the original training matrix unlike in the SVD case. The next table shows several runs of the algorithm with the corresponding minimum values

obtained for k and N_{min} (the left table corresponds to the Frobenius norm and the right table to the nuclear norm).

Test #	min k	N_{min}
1	8	250.6
2	12	250.2
3	15	250.4
4	14	250.4
5	14	250.0

Test #	min k	N_{min}
1	16	5021.8
2	18	5011.2
3	15	5009.5
4	18	5004.1
5	15	5012.0

On average we have $k \approx 12.6 \pm 1.1$ with a norm $N_{min} \approx 250.3 \pm 0.2$ for the Frobenius norm and $k \approx 16.4 \pm 0.6$ with a norm $N_{min} \approx 5011.7 \pm 2.6$ for the nuclear norm.

- $\|MU - R_{train}\|_F$ **and** $\|MU - R_{test}\|_F$ **for** $f = movie\_avg$**:**

Remark: the calculations were made using rank jumps of 1, for $k \in [1, 50]$ with subsequent jumps of 10 for $k \in [50, 300]$ (due to convergence time).

When using $f = movie\_avg$ we unfortunately ran into some issues. The algorithm, which should never increase the Frobenius norm at each step, when comparing to the original matrix, actually did so. One possible reason for this is the division step issue we mentioned previously. It is somewhat weird that the problem occurs only when using $f = movie\_avg$ and it might also be some specific implementation bug we could not find out in time. The results were the following:

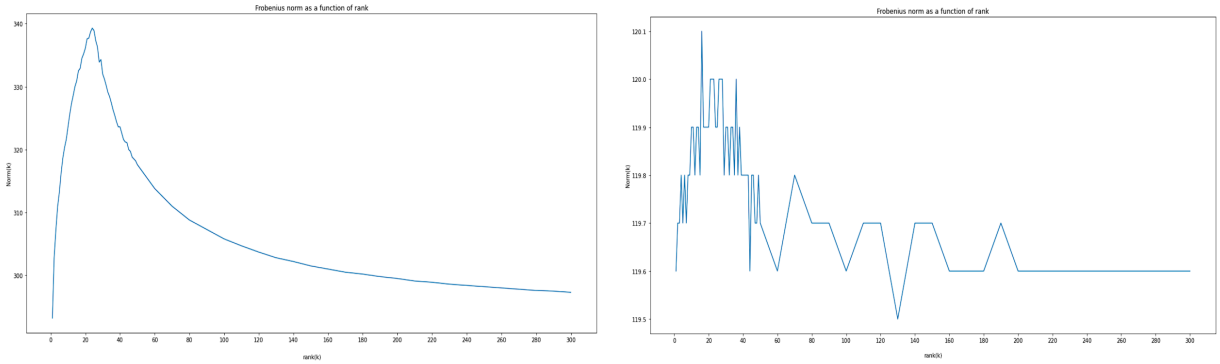

Figure 11.: $\|MU - R_{train}\|_F(k)$ (left) and $\|MU - R_{test}\|_F(k)$ (right)

Since the results are somewhat questionable we defined the function **nnmf_sk()** which uses the sci-kit library function **non_negative_factorization()** with $f = movie\_avg$, the results were the following:

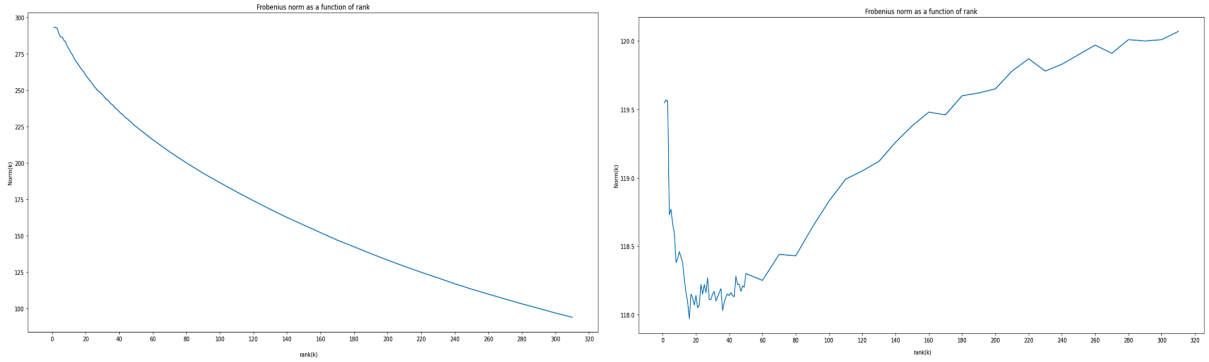

Figure 12.: $\|MU - R_{train}\|_F(k)$ (left) and $\|MU - R_{test}\|_F(k)$ (right)

Clearly the results of fig. 12 make a lot more sense than those of fig. 11. On the left we have non-increasing

convergence towards the training matrix, as expected, and on the right we have a similar result as the SVD case when comparing to the test set, i.e. showing that a lower latent space dimension is better at capturing the nature of the data since for higher values we are essentially overfitting.

For one run of the algorithm (for brevity), we obtain $k = 16$ with a norm $N_{min} = 117.97$, using the Frobenius norm.

5.3. Algorithm 1: Recommendation System

The recommendation systems are the same as described in the case of SVD but now, of course, one uses the approximation $R \approx MU$ obtained from **algorithm 1**. We build similar functions **recommend_movies_based_on_movie_nnmf()** and **recommend_movies_based_on_user_nnmf()** which are essentially the same as the SVD case but where at the start instead of using the **svd()** function we use **nnmf()** (ideally the recommender systems would receive as input the factorization but we chose to implement separate functions since it facilitates debugging). Using the same example as before we obtain (with $f = 0.5$ and $k = 13$):

```
Recommendations similar to: Lord of the Rings: The Fellowship of the Ring. The (2001)

1 ) Lord of the Rings: The Two Towers. The (2002) Similarity: 1.0
2 ) Lord of the Rings: The Return of the King. The (2003) Similarity: 0.99
3 ) Fight Club (1999) Similarity: 0.92
4 ) Departed. The (2006) Similarity: 0.92
5 ) Good Will Hunting (1997) Similarity: 0.92
6 ) Green Mile. The (1999) Similarity: 0.91
7 ) V for Vendetta (2006) Similarity: 0.91
8 ) Batman Begins (2005) Similarity: 0.91
9 ) Bourne Identity. The (2002) Similarity: 0.9
10 ) Beautiful Mind. A (2001) Similarity: 0.89
```

Again the sequels are highly recommended, however among the remaining recommendations we have popular well rated movies but which do not really fall under the same genre (e.g. "Fight Club", "The Departed", "Good Will Hunting", "A Beautiful Mind").

Using now $f = movie\_avg$ and $k = 16$ (with the sci-kit non-negative factorization function) we obtain:

```
Recommendations similar to: Lord of the Rings: The Fellowship of the Ring. The (2001)

1 ) Lord of the Rings: The Two Towers. The (2002) Similarity: 1.0
2 ) Lord of the Rings: The Return of the King. The (2003) Similarity: 0.99
3 ) Incredibles. The (2004) Similarity: 0.98
4 ) Spider-Man 2 (2004) Similarity: 0.98
5 ) Fifth Element. The (1997) Similarity: 0.98
6 ) Mask of Zorro. The (1998) Similarity: 0.98
7 ) X2: X-Men United (2003) Similarity: 0.98
8 ) Spider-Man (2002) Similarity: 0.97
9 ) Memento (2000) Similarity: 0.97
10 ) Silence of the Lambs. The (1991) Similarity: 0.97
```

These results might be more desirable since they fall under the fantasy/sci-fi genre with the exception of "Memento" and "The Silence of the Lambs" which are nonetheless popular and well rated movies.

Performing the user recommendation, using again *user_id*=134 for comparison, we obtain the following:

```
Expected ratings for user: 134
1 : Shawshank Redemption. The (1994) -> Expected rating: 4.54
2 : Matrix. The (1999) -> Expected rating: 4.42
3 : Star Wars: Episode IV - A New Hope (1977) -> Expected rating: 4.4
4 : Godfather. The (1972) -> Expected rating: 4.37
5 : Schindler's List (1993) -> Expected rating: 4.35
6 : Star Wars: Episode V - The Empire Strikes Back (1980) -> Expected rating: 4.31
7 : Fight Club (1999) -> Expected rating: 4.3
8 : Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964) -> Expected ratin
9 : Apocalypse Now (1979) -> Expected rating: 4.26
10 : Princess Bride. The (1987) -> Expected rating: 4.26
```

We have again popular and highly rated movies but now with the presence of a few sci-fi movies which might show a particular preference of user 134 (we did not actually confirm this).

5.4. Algorithm 1: Latent Space Projection

The following plots were obtained considering the function **nnmf_sk()** with $k = 16$ and for $f = 0.5$ on the left fig. (19) and $f = movie\_avg$ on the right fig. (20). For the 14 movies chosen by the authors the results obtained were the following:

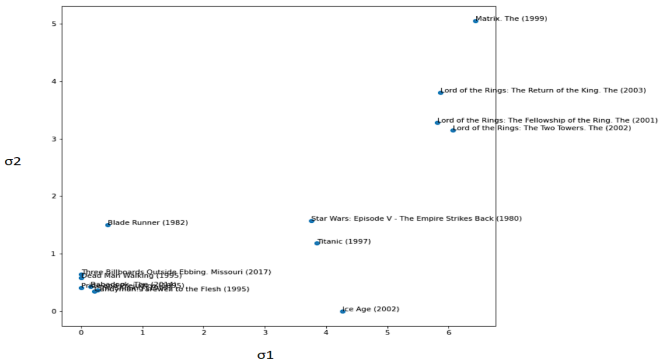

Figure 13.: Latent space proj. for 14 specific movies with $f = 0.5$

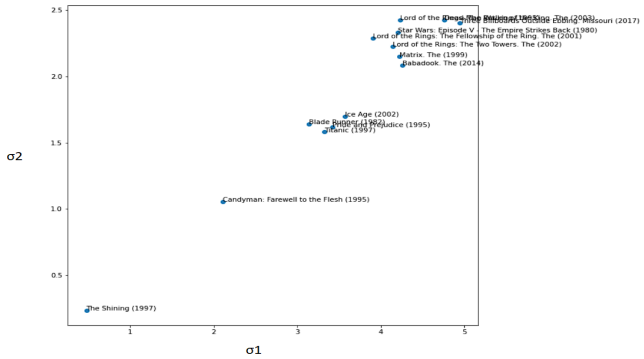

Figure 14.: Latent space proj. for 14 specific movies with $f = movie\_avg$

For 100 randomly chosen movies using $f = 0.5$ and $f = movie\_avg$ the results obtained were:

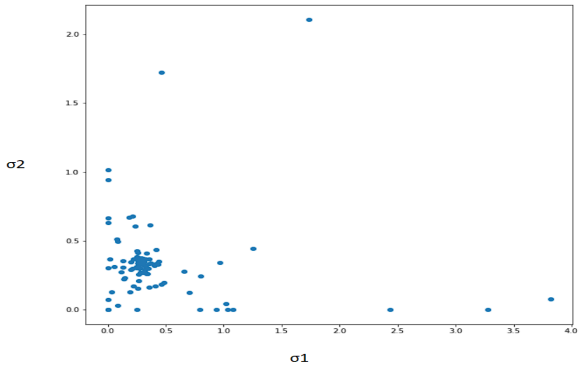

Figure 15.: Latent space proj. for 100 movies using $f = 0.5$

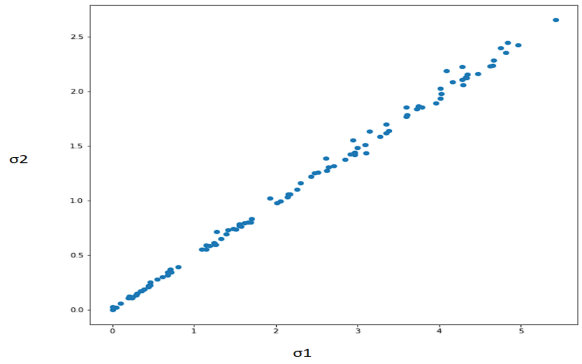

Figure 16.: Latent space proj. for 100 movies using $f = movie\_avg$

Similarly to the SVD case the left shows a majority of movies falling along the left lower portion of the plot, however with less dispersion over the first singular value direction. The right plot shows there is a huge correlation between the first and second latent factors which suggests the information could be essentially captured on a single latent factor, if our intuition is correct. This is perhaps due to the fact that taking the averages already give decent results but we are not quite sure how to interpret this.

5.5. NNALS: Factorization

For the non-negative alternating least squares we follow [4], however the actual algorithm we implement is a simpler version, in particular we did not implement Tikhonov regularization since we do not perform cross-validation to determine the best value for this hyperparameter.

Just like in the previous case we have R, a $m \times n$ matrix, and the idea is to pick $k < min\{n, m\}$ (again the latent space dimension) and factor $R \approx MU$, where M is a $m \times k$ matrix and U is a $k \times n$ matrix with non-negative entries. The idea is to use least squares in an alternating fashion, to first determine U and then M for example, while avoiding non-negative entries, and then iterate the process until we converge on a matrix which is close to R under the Frobenius norm. Suppose we fix a starting M and we want to determine U, we can use least squares, i.e. find:

$$\arg\min_{u_1} \|Mu_1 - r_1\|_2$$

where $u_1 = U_{(:,1)}$ and $r_1 = R_{(:,1)}$, i.e. the first columns of U and R respectively. Doing this for all columns we obtain:

$$\arg\min_{U} \|MU - R\|_2$$

After this step, we replace all obtained negative entries in U with zeros. Next we determine M from this new matrix U. It is easy to see from the dimensions that we need to compute the following:

$$\arg\min_{M^T} \|U^T M^T - R^T\|_2$$

Again, we replace all obtained negative entries in M with zeros. We proceed doing this iteratively until the following norm is smaller than some stopping condition ϵ:

$$\|MU - R\|_F < \epsilon$$

The algorithm implementation is done with the function **NNALS()**, which uses numpy's function **np.linalg.lstsq()**, and does the following:

Algorithm 2 Non-negative least squares matrix factorization for $R \approx MU$

Input: R_{train}: Training data in matrix format; k: rank of approximation used (or latent space dimension);
Output: M, U: Factorization matrices;
Initialization: M,U: Initialized as random matrices with values in range $[0, 1]$; error= 10; $\epsilon = 1$;
$R_0 = MU$
while *(error*$> \epsilon$*)* **do**
  $U = np.linalg.lstsq(M, R)$
  $U[U < 0] = 0$
  $M = np.linalg.lstsq(U^T, R^T)$
  $M[M < 0] = 0$
  $R_1 = MU$
  $error = \|R_1 - R_0\|_F$
  $R_0 = R_1$

5.6. NNALS: Reconstruction Errors

Similar to what was done with the previous methods we now use the function **NNALS_comparison** to compute the reconstruction errors. The resulting plots show the the blue line representing $\|MU(k) - R_{train}\|_F$ and the orange line representing $\|MU(k) - R_{test}\|_F$. The calculations were made using rank jumps of 1, for $k \in [1, 50]$ with subsequent jumps of 10 for $k \in [50, 300]$. The limit of 300 was used due to the fact that the algorithm takes a long time to converge and this time increases with k.

- $\|MU(k) - R_{train}\|_F$ **and** $\|MU(k) - R_{test}\|_F$ **for** $f = 0.5$**:**

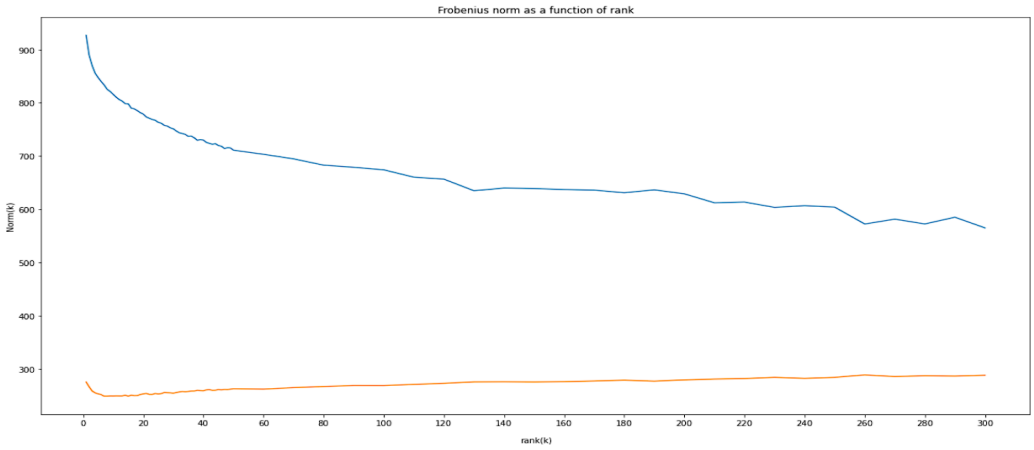

Figure 17.: $\|MU(k) - R_{train}\|_F$ (blue line) and $\|MU(k) - R_{test}\|_F$ (orange line)

- $\|MU(k) - R_{train}\|_F$ **and** $\|MU(k) - R_{test}\|_F$ **for** $f = 0.5$**:**

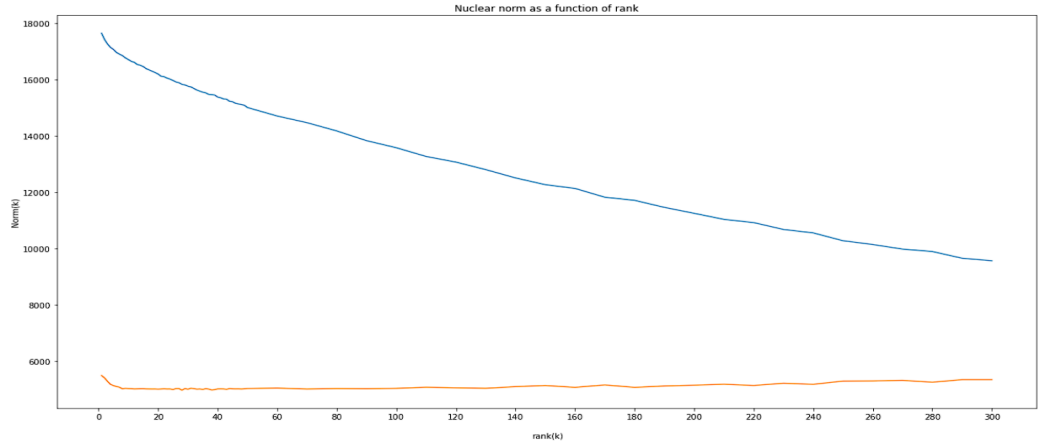

Figure 18.: $\|MU(k) - R_{train}\|_*$ (blue line) and $\|MU(k) - R_{test}\|_*$ (orange line)

The behaviour is similar to the previous methods with more pronounced non-monotonic convergence for this particular method (seen more clearly in fig. 17).

The following tables show several runs of the algorithm and the lowest norm obtained when compared to a test set, with corresponding latent space dimension k. The left table corresponds to the Frobenius norm and the right table to the nuclear norm.

Test #	min k	N_{min}
1	7	248.6
2	9	248.1
3	12	248.3

Test #	min k	N_{min}
1	28	4959.0
2	32	4975.4
3	31	4961.7

On average we have $k \approx 9.3 \pm 1.8$ with a norm $N_{min} \approx 247.8 \pm 1.1$ for the Frobenius norm (computing using 7 tests, we show only 3) and for the nuclear norm we obtain $k \approx 30.3 \pm 2.1$ with a norm $N_{min} \approx 4965.4 \pm 8.8$.

5.7. NNALS: Recommendation System

The recommendation systems are the same as described in the previous sections but now using the approximation $R \approx MU$ obtained from the **NNALS algorithm**. We build similar functions **recommend_movies_based_on_movie_nnals()** and **recommend_movies_based_on_user_nnals()** which are essentially the same as the previous cases but where at the start we use **NNALS()** for factorization. Using the same example as before we obtain (with $f = 0.5$ and $k = 9$):

```
Recommendations similar to: Lord of the Rings: The Fellowship of the Ring. The (2001)

1 ) Lord of the Rings: The Two Towers. The (2002) Similarity: 0.9985
2 ) Lord of the Rings: The Return of the King. The (2003) Similarity: 0.9926
3 ) Bourne Identity. The (2002) Similarity: 0.9907
4 ) Matrix Reloaded. The (2003) Similarity: 0.9816
5 ) Gladiator (2000) Similarity: 0.9809
6 ) Kill Bill: Vol. 2 (2004) Similarity: 0.9803
7 ) Matrix Revolutions. The (2003) Similarity: 0.9799
8 ) Departed. The (2006) Similarity: 0.9799
9 ) Ocean's Eleven (2001) Similarity: 0.9798
10 ) Minority Report (2002) Similarity: 0.9766
```

The results are sensible, again the sequels coming first as we already had previously, but also several other somewhat fantasy/sci-fi movies being recommended, except for "Gladiator", "The Bourne Identity", "The Departed" and "Ocean's Eleven which are nonetheless popular and well rated movies. Note also that the similarity values are all quite high being necessary to have 3 decimal places to actually see a difference in most cases.

Using now $f = movie\_avg$ and $k = 9$ we obtain:

```
Recommendations similar to: Lord of the Rings: The Fellowship of the Ring. The (2001)

1 ) It Follows (2014) Similarity: 1.0
2 ) Dracula: Dead and Loving It (1995) Similarity: 1.0
3 ) Rocky Balboa (2006) Similarity: 1.0
4 ) Basic (2003) Similarity: 0.9999999999
5 ) Princess Caraboo (1994) Similarity: 0.9999999999
6 ) Blood Work (2002) Similarity: 0.9999999998
7 ) You Only Live Twice (1967) Similarity: 0.9999999998
8 ) Fog. The (1980) Similarity: 0.9999999998
9 ) I'm Not There (2007) Similarity: 0.9999999998
10 ) Skin I Live In. The (La piel que habito) (2011) Similarity: 0.9999999997
```

This is the worst outcome so far, all movies have almost the same similarity (being necessary 10 decimal places to see a difference) and with the recommendations seeming somewhat random. We are not quite sure why this happens.

Performing the user recommendation, using again $user\_id$=134 for comparison, we obtain the following:

```
Expected ratings for user: 134
1 : Forrest Gump (1994) -> Expected rating: 4.57
2 : Shawshank Redemption. The (1994) -> Expected rating: 4.45
3 : Casablanca (1942) -> Expected rating: 4.28
```

```
4 : Streetcar Named Desire. A (1951) -> Expected rating: 4.27
5 : Godfather. The (1972) -> Expected rating: 4.26
6 : Fight Club (1999) -> Expected rating: 4.25
7 : Sting. The (1973) -> Expected rating: 4.25
8 : Usual Suspects. The (1995) -> Expected rating: 4.22
9 : Groundhog Day (1993) -> Expected rating: 4.22
10 : Godfather: Part II. The (1974) -> Expected rating: 4.22
```

This is again similar to what we obtained before, recommending popular and highly rated movies.

5.8. NNALS: Latent Space Projections

The following plots were obtained considering the function **latent_space_proj_nnals_test()** with $k = 9$ and for $f = 0.5$ (we do not show $f = movie\_avg$ for brevity and because the results were poor). For the 14 movies (we just show 8 since the missing names overlapped) chosen by the authors the results obtained were the following:

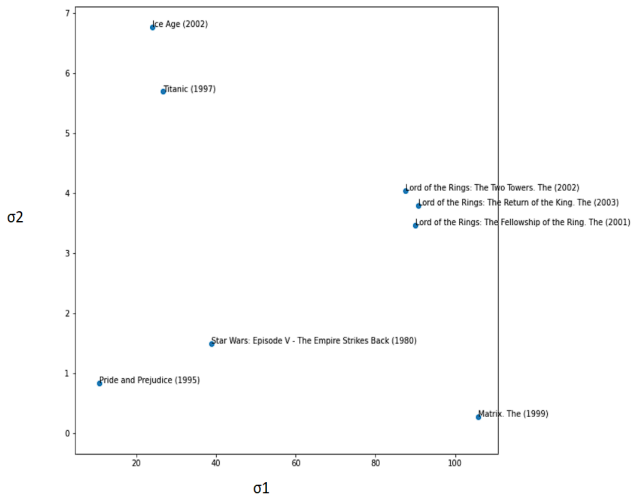

Figure 19.: Latent space proj. for 8 specific movies with $f = 0.5$

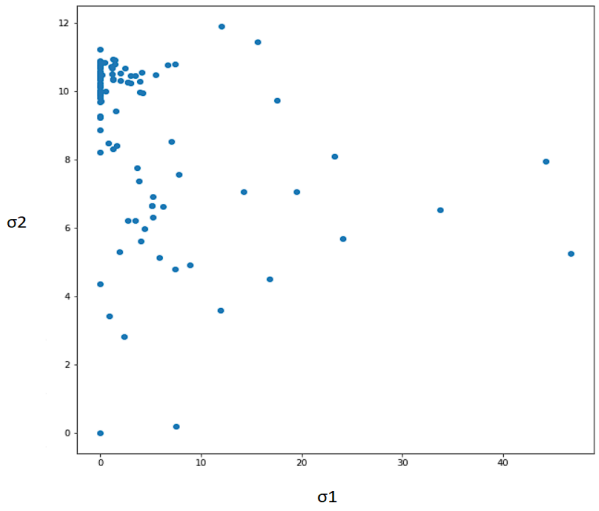

Figure 20.: Latent space proj. for 100 random movies with $f = 0.5$

The movies again agglomerate, this time in the upper left section of the plot and we see, on the left, that the "Lord of the Rings" trilogy is quite similar on the latent space projection, especially along the first factor.

6. Conclusions

All three factorization algorithms (SVD/Algorithm 1 (NNMF)/Non-negative alternating least squares) have resulted in decent recommendation systems with similar outcomes when using as filling value either $f = 0.5$ or $f = movie\_avg$. This can be seen from the obtained similar movie recommendations and for the similar movie recommendations for users. However, in the last case of $f = movie\_avg$ we ran into some issues when using the non-negative matrix factorization algorithms, particularly the alternating least squares method (NNALS). We are not quite sure why this is the case since all explanations seem quite contradictory[2], perhaps this is a result of some bug in the code which we could not find out.

Regarding the latent space projection, it is quite difficult to assign a meaning to each particular factor in terms of genre. After projecting a few different genres we often see very distinct movies (genre-wise) in similar regions within the 2-dimensional latent space plots. Nonetheless, we still obtain very similar movies within proximity of each other. This is true for example for "The Lord of Rings" trilogy with all algorithms

[2]We thought that assigning movie averages might be severely biased, particularly for movies with very few ratings, but still it does not explain why it works so well when comparing with a test set and why it works as a recommendation system for the other factorizations.

tested.

There are several avenues for improving this work. Particularly regarding the implementation of the non-negative matrix factorization (NNMF) algorithms. In the specific case of **algorithm 1** there are some numerical conditioning problems on the division step which could be better handled. On both of the NNMF algorithms the running time was very large, particularly for large k, so there might be some possible optimization on that regard as well.

7. Bibliography

[1] Daniel D. Lee and H. Sebastian Seung: Algorithms for Non-negative Matrix Factorization. In: *Advances in Neural Information Processing Systems 13 (NIPS)* (February, 2001)

[2] F. Maxwell Harper and Joseph A. Konstan: The MovieLens Datasets: History and Context. In: *ACM Transactions on Interactive Intelligent Systems* Vol. 5, No. 4 (January, 2016)

[3] Yehuda Koren, Robert Bell and Chris Volinsky: Matrix Factorization Techniques for Recommender Systems. In: *Computer* Vol. 42, Issue. 8 (August, 2009)

[4] Yunhong Zhou and Robert Schreiber: Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In: *Algorithmic Aspects in Information and Management* (June, 2008)