Movie recommendation algorithm using the MovieLens dataset

Alessandro D'Auria, Emanuele Di Maio, Olesia Nitsovych, Tiziano Turco

University of Naples Federico II

Hardware and Software Architectures for Big Data mod. B

A.A 2020/2021

Contents

Introduction

A recommendation system is a type of information filtering system which attempts to predict the preferences of a user, and make suggests based on these preferences.

There are a wide variety of applications for recommendation systems. These have become increasingly popular over the last few years and are now utilized in most online platforms that we use. The content of such platforms varies from movies, music, books and videos, to friends and stories on social media platforms, to products on e-commerce websites, to people on professional and dating websites, to search results returned on Google.

Often, these systems are able to collect information about users choices, and can use this kind of data to improve their suggestions in the future. For example, Facebook can monitor interactions with various stories on our feed in order to learn what types of stories appeal to us. Sometimes, the recommender systems can make improvements based on the activities of a large number of people. For example, if Amazon observes that a large number of customers who buy the latest Apple Macbook also buy a USB-C-to-USB Adapter, they can recommend the Adapter to a new user who has just added a Macbook to his cart.

Due to the advances in recommender systems, users constantly expect good recommendations. They have a low threshold for services that are not able to make appropriate suggestions. If a music streaming app is not able to predict and play music that the user likes, then the user will simply stop using it. This has led to a high emphasis by tech companies on improving their recommendation systems. However, the problem is more complex than it seems. Every user has different preferences and likes. In addition, even the taste of a single user can vary depending on a large number of factors, such as mood, season, or type of activity the user is doing. For example, the type of music one would like to hear while exercising differs greatly from the type of music she would listen to when cooking dinner. Another issue that recommendation systems have to solve is the exploration vs exploitation problem. They must explore new domains to discover more about the user, while still making the most of what is already known about the user.

Recommendation systems can be loosely broken down into three categories: ***content based systems***, ***collaborative filtering systems***, and ***hybrid systems*** (which use a combination of the other two).

Content based approach utilizes a series of discrete characteristics of an item in order to recommend additional items with similar properties.

Collaborative filtering approach builds a model from a user's past behaviors (items previously purchased or selected and/or numerical ratings given to those items) as well as similar decisions made by other users. This model is then used to predict items (or ratings for items) that the user may have an interest in.

Hybrid approach combines the previous two approaches. Most businesses probably use hybrid approach in their production recommendation systems.

In our project, we are going to use collaborative filtering systems, on which we are going to explain more on section 3.

1 Challenge description

In this project, our aim will be building a **recommendation algorithm for suggesting movies to a potential user, based on his personal preferences**. Current literature is actually very rich in terms of movie recommendation algorithms: for our purposes, we are therefore going to test two different methods, in order to scan which one can be better effective in terms of precision, computational time and quality of the result. These techniques are:

1. **K-nearest Neighbor** or KNN, is a machine learning algorithm which is used to find similar users among clusters of many individuals by their likes, ratings or reviews given by each specific user and make predictions according to the shortest distance among them.

2. **Alternating Least Squares**, or ALS, which is the algorithm that aims to find other users in the training set with preferences similar to the current selected user. Recommendations

for the current user are then created based on the preferences of such similar profiles. This means that we need a profile for the current user to match the profiles of other existing users in the training set.

The effectiveness and accuracy of these methods have been evaluated following these criteria:

1. RMSE (Root Mean Square Error) value: given that the RMSE would represent how our prediction deviates from the actual observation, we want the RMSE to be as lowest as possible in order to follow the user preferences as close as we can;
2. Computational time: the training dataset we have been using is quite large, and it can take some time for the prediction to be evaluated. The computational time must be taken into account if the algorithm is to be used for user experiences;
3. Accuracy of the response: aside from the numerical factors and the precision of the response, our algorithm must also be capable of filtering results which could be statistically ineffective or uninteresting for the user, like movie ratings with a small number of recommendations.

2 Architecture and Environment

In this first section we are going to discuss about the technologies and the tools used for our analysis. In particular, we are going to describe:

1. **The dataset** from which we have taken the movie information needed to build the model;
2. **The database** where we stored and queryed all the datasets;
3. **The environment, framework and libraries** used to perform the calculations;

2.1 Dataset description

In order to build the training model needed for our predictions, we used the official **MovieLens dataset**, which is a large set of tables containing information about movies and reviews taken from the movie recommendation website MovieLens. The dataset used has been collected by the GroupLens research lab from the University of Minnesota [1]. Following the official documentation on the GroupLens website,

> *"[The dataset] contains 20000263 ratings and 465564 tag applications across 27278 movies. These data were created by 138493 users between January 09, 1995 and March 31, 2015. This dataset was generated on October 17, 2016. Users were selected at random for inclusion. All selected users had rated at least 20 movies. No demographic information is included. Each user is represented by an id, and no other information is provided."*

A detailed documentation can be found at https://files.grouplens.org/datasets/movielens/ml-20m-README.html. For our purposes, we only needed two tables from the MovieLenses database:

- The movies.csv table, which contains a list of movie titles and the corresponding genres. Each movie is identified by a numerical id value.
- The ratings.csv table, which contains a list of user Ids and, for each of them, a movie id (in a foreign key relation with the movies.csv table) and the rating (from 1 to 5, with a pace of 0.5) the user assigned to that movie. We have dropped timestamp column in order to achieve better time-performance of algorithm, since we do not use it in analysis part.

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

Table 1: The movies.csv table and its first three rows

userId	movieId	rating
1	296	5.0
1	306	3.5
1	307	5.0

Table 2: The ratings.csv table and its first three rows

2.2 Database

Following the project constraints, we uplaoded our datasets to a NoSQL database. In particular, for our purposes we chose **MongoDB** [2], for its easy-to-use API and its online services which made possible to share data on the cloud service MongoDB Atlas. Notice that, however, the cloud service has been used only for the testing phase due to its hard drive memory restriction (500 MB maximum), forcing us, for the actual experimental phase, to use MongoDB locally on our machines.
Being a NoSQL database, MongoDB is structured in collections and documents rather than databases and tables: each group of objects (a document) is represented by a set of JSON-like structure (the MongoDB BSON), like the following:

```
{
    "_id": {
        "$oid": "60c2bfee1a839b460814aacc"
    },
    "movieId": "1",
    "title": "Toy Story (1995)",
    "genres": "Adventure|Animation|Children|Comedy|Fantasy"
}
```

MongoDB has also been used to keeping track of our progresses, storing all the predictions made with our models.

2.3 Programming languages, libraries and environment

All algorithms have been written in **Python3.9**, due to its large variety of machine learning and data manipulation libraries. In particular, we have been using the following libraries:

- **pyspark**, for its distributed computing features based on the **Spark framework** and its implementation with MongoDB;
- **MLlib**, for its machine learning features available in Spark;
- **pandas**, for its simple dataframes and data handling;
- **pymongo**, in order to connect to our MongoDB database and perform all the queries (when pyspark was not worth using due to the small number of rows to query);
- **sklearn**, for its machine learning features;
- **scipy**, for some useful matrix manipulation.

Due to its native application to distributed processing, it would have been useful to perform our calculations on a cluster hosted by one of the most famous online provider for cloud computing (AWS, Microsoft Azure, etc.). This idea, however, was quickly discarded due to the lack of effective free tier offered by these services. Notice, for example, how the free cluster hosted by Databricks (the community edition) only offers a 15.3 GB memory machine with two cores and no scaling features: our local Windows computers revealed better performances, so we discarded the possibility of cloud computation.

2.4 Proposed Architecture for Movie Recommender System

We hereby summarize the architecture of our models:

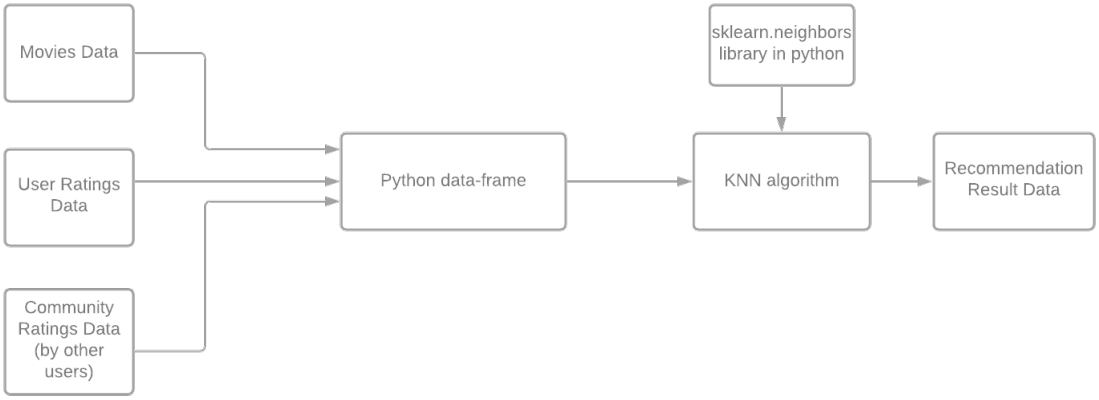

Figure 1: Architecture for movie recommendation system using KNN algorithm

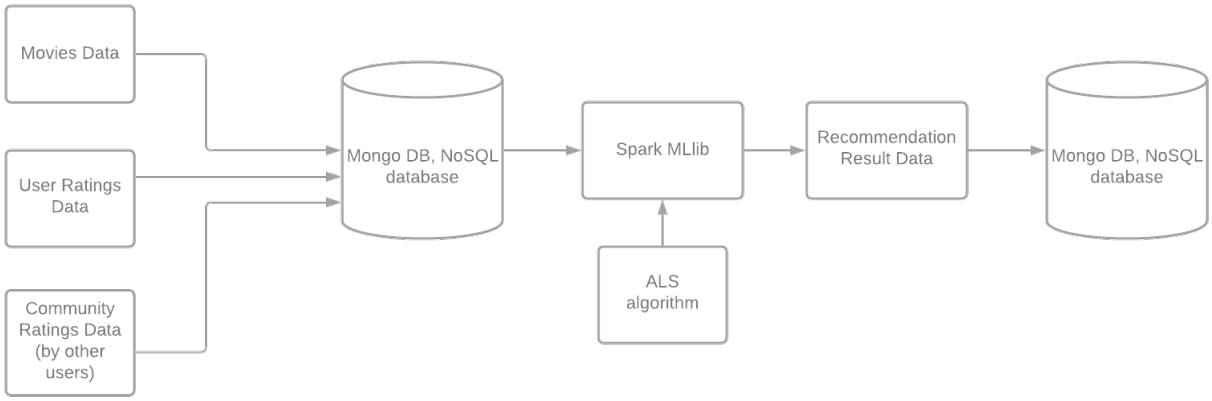

Figure 2: Architecture for movie recommendation system using ALS algorithm

3 Methodology

We are now going to analyze the two algorithms used to build our recommendation system. The goals of these methods is to forecast, based on certain number of movies reviewed by the user, how much the same user would enjoy other movies that he has not reviewed yet. In order to achieve this result, we based our algorithms on the so called **collaborative filtering technique** (CF), which is commonly used in recommendation systems problems. In the collaborative filtering, user interests are predicted comparing his tastes to other users preferences: it is indeed fair to assume that, if a person A likes movies one, two and three, and another person B likes

movies one, two, three and four, then A is probably going to like movie four as well.

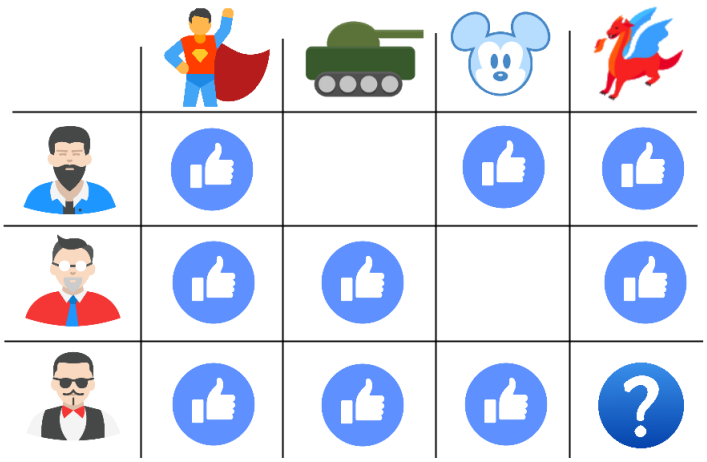

Figure 3: Collaborative filtering explained with a simple example: user 1 likes movies about superheroes, cartoons and fantasy; user 2 likes movies about superheroes, war and fantasy. User 3, which shares very close preferences with user 1 and 2 for his likes in superheroes, war and animation, is then likely to enjoy fantasy movies as well.

The matrix represented schematically in Figure 3 shows the core concept of the collaborative filtering: filling the gaps in user preferences following the preferences of similar users. In our case, our dataset provides different movie ratings for several individuals, thus our matrix can be made up of elements A_{ij} representing ratings from 1 to 5, where i represents the user index, while j represents the movie index. In general, the manipulation of the matrix above can be quite computationally challenging, given that the number of users and movies can be quite big; furthermore, this matrix is generally *sparse*, i.e. most of its entries are actually empty due to the fact that, of course, not every user shows his preferences to all possible movies, thus the algorithm used must also be capable of making inference with a small sample of available data.

movieId	1	47	50	110	260	296	318	356	480	527	589	593	858	1196	1198	1210	2571	2858	2959	4993	5952	7153
userId																						
1	NaN	NaN	NaN	NaN	NaN	5.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4.0	NaN
2	3.5	NaN	NaN	5.0	5.0	NaN	5.0	4.5	2.0	5.0	4.0	NaN	3.5	5.0	4.0	5.0	5.0	NaN	NaN	5.0	5.0	5.0
3	4.0	NaN	5.0	NaN	4.0	5.0	4.0	4.0	2.0	4.0	4.0	4.0	5.0	4.0	3.0	4.0	4.0	5.0	5.0	4.0	4.0	4.0
4	3.0	NaN	NaN	NaN	3.5	4.0	NaN	NaN	NaN	NaN	4.0	NaN	NaN	4.0	4.0	3.0	4.5	NaN	NaN	4.5	4.5	4.0
5	4.0	5.0	5.0	NaN	5.0	4.0	NaN	4.0	NaN	NaN	NaN	5.0	NaN	5.0	NaN	5.0	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
162537	NaN	NaN	NaN	NaN	NaN	NaN	5.0	5.0	NaN	NaN	NaN	5.0	NaN	NaN	NaN	NaN	NaN	NaN	5.0	NaN	NaN	NaN
162538	2.0	4.0	NaN	NaN	1.5	2.5	NaN	4.0	NaN	4.5	NaN	4.5	NaN	NaN	NaN	NaN	2.5	4.0	2.5	4.0	4.0	NaN
162539	NaN	NaN	NaN	4.0	NaN	NaN	NaN	5.0	4.0	NaN	NaN	NaN	NaN	4.0	5.0	4.0	NaN	NaN	NaN	NaN	NaN	NaN
162540	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	5.0	NaN	NaN
162541	NaN	4.5	5.0	NaN	5.0	3.5	4.0	NaN	NaN	4.5	NaN	4.0	NaN	5.0	4.5	3.5	5.0	5.0	5.0	5.0	5.0	5.0

Figure 4: A small subset of our dataset represented by a sparse matrix. Notice how the majority of enters are missing ratings.

3.1 KNN Algorithm

For our first attempt we have developed a collaborative filtering system by using **KNN** algorithm with item-based approach.

The k-nearest neighbor algorithm stores all the available data and classifies a new data point based on the similarity measure (e.g., distance functions), then it can be easily classified into a well-suited category using KNN algorithm.

If we suppose that there are two classes, i.e., Class A and Class B, and we have a new unknown data point ?, we aim to determine in which class this data point will lie. With the help of KNN, we can easily identify the class of a particular dataset. The data point is classified by

a majority vote of its neighbors, with the data point being assigned to the most common class amongst its K nearest neighbors measured by a distance function.

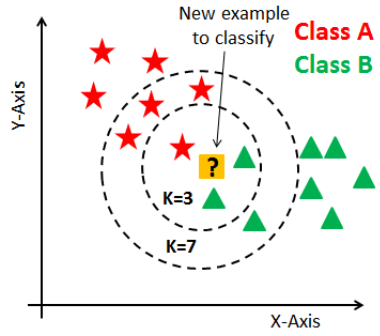

Figure 5: Example of KNN algorithm

Here, we can see that if k = 3, then based on the **distance function** used, the nearest three neighbors of the data point is found and based on the majority votes of its neighbors, the data point is classified into a class. In the case of k = 3, for the above diagram, it is Class B. Similarly, when k = 7, for the above diagram, based on the majority votes of its neighbors, the data point is classified into Class A.

The basic idea of our system is that, when KNN makes inference about a movie, it will calculate the distance between the target movie and every other movie in its database, then it ranks its distances and returns the top K nearest neighbor movies as the most similar movie recommendations.

For more efficient calculation and less memory footprint, we need to transform the values into a sparse matrix. As it was shown in Figure 4 has a very high dimensionality. KNNs performance will suffer from "curse of dimensionality" if the euclidean distance will be used, so we will use ***cosine similarity*** for nearest neighbor search, widely used for ranking problems, because it outperforms the other metric systems when the dimention of the vectors is over a thousand.

Cosine Similarity is a measurement that quantifies the similarity between two vectors. It is the cosine of the angle between the two vectors, considering them typically non-zero and within an inner product space. The cosine similarity is described mathematically as the division between the dot product of vectors and the product of the euclidean norms of each vector:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}||||\mathbf{B}||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{1}$$

In our case the Cosine Similarity is a value that is bound by a range between 0 and 1.

Supposing that the angle between the two vectors is 90 degrees, the similarity will have a value of 0; this means that the two vectors are orthogonal or perpendicular to each other and very dissimilar.

As the angle between the two vector gets smaller, the cosine similarity measurement gets closer to 1. The figure below depicts how we should interpret our vectors (movies):

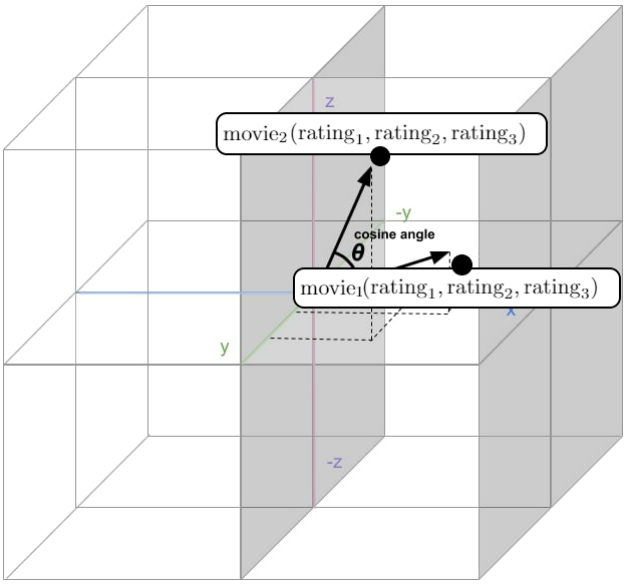

Figure 6: The angle between two rated vectors associated to three different users, for the simple case where there are only two movies (and thus three possible ratings as components of the vectors)

We start our item-based KNN algorithm by importing the all needed libraries (os, math, pandas, numpy, scipy.sparse, sklearn.neighbors) and importing the two dataset: "movies.csv" and "rating.csv" that were described in Section 2. Then, we did some exploratory data analysis, which is described in details in Jupyter notebook that is attached to this project. Next, for practical reasons, to not fall in a memory issue and to improve KNN perfomance, since we have so many movies and so many missing values (considering that an user can't rate all the movies), we'll set an arbitrary threshold for popularity. After dropping a part of the movies in our dataset, we still have a very large dataset. So next we can filter users in the same way we did before, to further reduce the size of data, since we can assume that a majority of users aren't interested in rating movies. So we can limit users, with a threshold for activity limiting in this way the number of missing values that will only bring noise to the analysis.

For K-Nearest Neighbors, we want the data to be in an (movie, user) matrix, where each row is a movie and each column is a different user.

In our case the movies (rows) will be the vectors, and the values of the ratings are their components on the axis; we are interested in finding which movies are similar to a specific one so that we can make reccomendations for an user basing our analysis on the highest rated movie (by that user) comparing his rating (a component on the axis) with rating of other users of the same movie.

To reshape the dataframe, we will pivot the dataframe with movies as rows and users as columns, using rating as values. Then we'll fill the missing observations with 0s since we're going to be performing linear algebra operations (calculating distances between vectors). Finally, we transform the values of the dataframe into a scipy sparse matrix for more efficient calculations. By specifying the *metric = cosine* (for the aforementioned reasons), the model will measure similarity between movie vectors by using cosine similarity.

We have also implemented fuzzy matching, which is a technique used in computer-assisted translation. It works with matches that may be less than 100% perfect when finding correspondences between segments of a text and entries in a database of previous translations. We have done it for the user who is inserting a name of movie, and system is searching for its name in the dataset.

Finally, the system implements KNN based on cosine similarity, which means that the smaller the angle between vectors, the more similar they are. As already mentioned this similarity has bounds from 0 to 1. At the same time, the cosine distance can be defined as *1 - cosine similarity*. KNN algorithm takes into account exactly the distance. The more similar the movies are, closer

the distance will be to 0.

We can test our programm considering the movie "*Sex and the City*". First of all, the program will give the output of fuzzy matching, which is:

"You have input movie: Sex and the city Found possible matches in our database: "Sex and the City (2008)*",* "Sex and the City 2 (2010)"*,* "Babe: Pig in the City (1998)"

This is a great result which means that fuzzy matching is correctly working.

In addition, we print out a matrix of cosine distances and the corresponding movieIds checking that the first value is 0, since it is the distance of the movie from itself, thus explained why we take this as validating value:

```
[[0.         0.62896717 0.69190425 0.6960783  0.7111819  0.71603847
  0.71903324 0.7424477  0.7566228  0.75702155 0.75815475]] [[8770 9517 8047 8809 9040 8215 8633 8958 8631 6907 8302]]
```

Finally, we retrieve the list of top 10 recommendations (9 if we don't consider the first value):

```
1: Sex and the City (2008), with distance of 0.0
2: Sex and the City 2 (2010), with distance of 0.6289671659469604
3: Devil Wears Prada, The (2006), with distance of 0.6919042468070984
4: Mamma Mia! (2008), with distance of 0.6960783004760742
5: He's Just Not That Into You (2009), with distance of 0.7111818790435791
6: Holiday, The (2006), with distance of 0.7160384654998779
7: 27 Dresses (2008), with distance of 0.7190332412719727
8: Twilight (2008), with distance of 0.7424476742744446
9: P.S. I Love You (2007), with distance of 0.7566227912902832
10: Bridget Jones: The Edge of Reason (2004), with distance of 0.75702154636383
```

Figure 7: The result of KNN algorithm based on movie "*Sex and the city*"

The script runs really fast taking 1 minute and 6,864706 seconds to complete all the tasks, but we can notice that the distances obtained (without considering the first value that is obviously 0 since it relates the chosen movie to itself) are pretty high: a distance of $0,62$ translates in a similarity (we can consider the similarity as an estimate of the accuracy) of 38% that is not the most accurate result that we can obtain for this task.

3.2 ALS

In our second attempt we solved the sparse matrix problem with the **alternating least square** method, which is a matrix factorization algorithm firstly used by [3] during the second Netflix Price Challenge [4]. The ALS method, which bases its core algorithm on the research of a local minimum for a function F using gradient descent, it is easily scalable, thus it is particularly suitable for large cluster computation.

By matrix factorization we denote a technique that can reduce a $n \times m$ matrix $\mathbf{A}$ in two matrix $\mathbf{B}$ and $\mathbf{C}$ with dimensions $n \times f$ and $f \times m$ respectively, for which $\mathbf{A} = \mathbf{B} \cdot \mathbf{C}$, where "$\cdot$" is the matrix dot product. If we apply this kind of methods to our rating sparse matrix $\mathbf{R}$, we can obtain two other matrices $\mathbf{M}$ and $\mathbf{U}$ whose dot product $\hat{\mathbf{R}}$ can fill (approximately) the missing spots and, thus, can give us a prediction of the missing reviews for certain users.

$$\begin{pmatrix} \text{NaN} & 5 & 2.0 \\ 4.0 & \text{NaN} & \text{NaN} \\ \text{NaN} & 5.0 & \text{NaN} \end{pmatrix} \sim \begin{pmatrix} 3.2 & 1.5 \\ 2 & 0.6 \\ 4.2 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1.5 & 0.9 & 0.1 \\ 1.7 & 1.4 & 1.1 \end{pmatrix}$$

Figure 8: An example of decomposition of a matrix 3×3 with two latent factors.

Our problem is thus reduced to the determination of matrices $\mathbf{M}$ and $\mathbf{U}$ for which:

$$\mathbf{R} \sim \hat{\mathbf{R}} = \mathbf{M} \cdot \mathbf{U} \rightarrow R_{ij} = \sum_{k}^{f} M_{ik} U_{kj} \tag{2}$$

The number f is a free parameter of our model, and is generally indicated as **the number of latent factors**, i.e. the hidden variables of our problem. Latent factors, even if purely mathematical artifacts, can be interpreted as a certain number of properties which can describe both users and movies: for example, a single latent factor could represent, in the movie matrix $\mathbf{M}$, the tendency for a certain movie to belong to a specific category, while the same latent factor in the user matrix could represent the interest of a certain user to that category. This way, their dot product would be a good representation of the rating a user would assign to that movie.
For a fixed f, it is natural to ask $\mathbf{U}$ and $\mathbf{M}$ to make the *mean square error* (MSE) between $\mathbf{R}$ and $\hat{\mathbf{R}}$ as small as possible. In other word, $\mathbf{U}$ and $\mathbf{M}$ can be found finding the minima of the function F defined as follows:

$$F(\mathbf{U},\mathbf{M}) = \frac{1}{n}\sum_{i}^{n_u}\sum_{j}^{n_m}(R_{ij} - (\mathbf{M}\cdot\mathbf{U})_{ij})^2 \tag{3}$$

or equivalently:

$$(\mathbf{U},\mathbf{M}) = \arg[\min_{(\mathbf{U},\mathbf{M})}F(\mathbf{U},\mathbf{M})] \tag{4}$$

where n_u is the total number of users, while n_m is the total number of movies. Do notice that in this case there are $(n_u + n_m)\cdot f$ free parameters to be evaluated. On the other hand, matrix $\mathbf{R}$ is sparse, thus the number of actual entries are way less than $n_u n_m$. Solving equation 4 with these criteria generally leads to overfitting, thus forcing ourselves to introduce some regularization parameters. In the ALS method, these are chosen as follows:

$$F(\mathbf{U},\mathbf{M}) = \frac{1}{n}\sum_{i,j}(R_{ij} - (\mathbf{M}\cdot\mathbf{U})_{ij})^2 + \lambda\left(\sum_i n_{u_i}||\mathbf{u}_i||^2 + \sum_j n_{m_j}||\mathbf{m}_j||^2\right) \tag{5}$$

Where n_{u_i} and n_{m_j} denote the number of ratings of user i and movie j respectively, $\mathbf{u}_i = (U_{1i}, U_{2i}, ..., U_{n_u i})$, $\mathbf{m}_i = (M_{i1}, M_{i2}, ..., M_{in_m})$ and λ is a free parameter, indicated as the **regularization parameter**, which can be determined by experimental results.
Therefore, at this point, we just have to evaluate the local minimum of F varying f and λ until we obtain an acceptable value for the MSE or, equivalently, its root square, the RMSE.
Fortunately, the PySpark library already provides a method for ALS calculations RMSE evaluation as well. We started applying the ALS to a test sample (a random subset the size of the 30% of the whole dataset, where, for computational simplicity, we cut off movies with less than 1000 ratings) for different values of λ and for different numbers of latent factors, obtaining the graph in Figure 9.

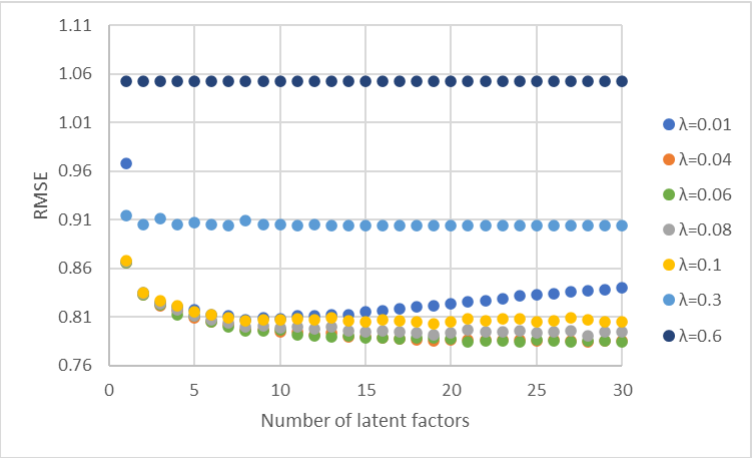

Figure 9: RMSE trend as a function of the number of latent factors, for different values of λ

Notice how the RMSE reaches its lowest values for $\lambda = 0.04 \sim 0.06$. Furthermore, for more than 25 latent factors it seems not to vary significantly. Therefore, we are going to use, for our

ALS algorithm, $\lambda = 0.06$ and $f = 25$.
Let us see how this model performs with these parameters. Suppose we defined the following ratings for some action and animation movies, as follows:

userId	movieId	title	rating
1000000	2	JUMANJI (1995)	5.0
1000000	260	STAR WARS: EPISODE IV - A NEW HOPE (1977)	5.0
1000000	783	THE HUNCHBACK OF NOTREDAME (1996)	4.0
1000000	1562	BATMAN AND ROBIN (1997)	3.0
1000000	5349	SPIDER-MAN (2002)	5.0

Table 3: Our movie preferences

Based on these recommendations, the ALS was able to return the following previsions (where, for sake of visualization, we reported only the first ten highest rated rows):

userId	title	original rating	predicted rating
1000000	Star Wars: Episode IV - A New Hope (1977)	5	4.9
1000000	Star Wars: Episode V - The Empire Strikes Back (1980)		4.9
1000000	Spider-Man 2 (2004)		4.9
1000000	Spider-Man (2002)	5	4.8
1000000	Toy Story (1995)		4.7
1000000	Toy Story 2 (1999)		4.7
1000000	Star Wars: Episode VI - Return of the Jedi (1983)		4.6
1000000	Indiana Jones and the Raiders of the Lost Ark (1981)		4.6
1000000	X2: X-Men United (2003)		4.6
1000000	Avengers, The (2012)		4.6

Table 4: Predicted ratings by ALS based on our preferences

With a RMSE value of 0.8 (0.7847208599784037 to be precise, but of course we considered a sensitivity of 0.5, which is the pace of the possible ratings). Notice how the predicted ratings are quite close to the original ones and, most importantly, we have been able to predict ratings for movies that we have not rated yet. All the recommendations are on point: the ALS algorithm is suggesting us other action movies which are closely related to the original ones (for example, other Star Wars Movies or other superheroe movies). These are indeed movies that we would really enjoy.
In order to appreciate the precision of the model, we will also show some random prediction for different users:

userId	title	original rating	predicted rating
14831	Awfully Big Adventure, An (1995)	3	2.8
93796	Guilty as Sin (1993)	3	3.2
152341	A Midsummer Night's Dream (1999)	3	1.9
57927	Ghost (1990)	4.5	3.9
129787	Star Trek: Generations (1994)	3	3.4
106989	Babe (1995)	5	3.8
99586	Silence of the Lambs, The (1991)	2.5	2.6
555	Pulp Fiction (1994)	3	4.1
92493	Playing by Heart (1998)	4	3.7
118707	Dracula (Bram Stoker's Dracula) (1992)	3	3.5

Table 5: Predicted ratings for random users

4 Conclusion

Let us conclude with some considerations about the methods we used, i.e. ALS and KNN:

- Both methods seem to furnish reasonable suggestions for a certain group of movies the user has previously selected;
- ALS can predict user ratings with a RMSE of 0.8, which is less than a whole step on the rating scale, making it very precise. KNN cannot produce an actual rating prediction and, thus, a RMSE to evaluate;
- Even with PySpark, the matrix decomposition process can take some time, up to 8 minutes on a medium windows machine. KNN, on the other hand, is way quicker, and it takes up to two minutes to run the algorithm.

Therefore, the ALS method seems to be more precise and reliable, but maybe too slow for real time services: if we were to build, for example, a movie recommendation website, we could not make the users wait for so much, and the KNN would reasonably seem more effective.

References

[1] F. Maxwell Harper and Joseph A. Konstan. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems.2015. [Internet]: https://doi.org/10.1145/2827872.

[2] MongoDB Inc. MongoDB NoSql Database.2007. [Internet]: https://www.mongodb.com/

[3] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber and Rong Pan. Large-scale Parallel Collaborative Filtering for the Netix Prize. 2009.

[4] Netflixprize. Netflix.2009. [Internet]: https://www.netflixprize.com/.