

# Spring 2019 - Independent Project (MAS 6V00)

-Darshil Sanghvi (Net ID: drs170530)

**Problem Statement:** To predict the user ratings of the movies analyzing the Netflix dataset.

Netflix held the Netflix Prize open competition for the best algorithm to predict user ratings for films. The grand prize was \$1,000,000 and was won by BellKor's Pragmatic Chaos team. The dataset provided for this analysis was of the size of 10 gb. The winning team implemented a novel approach to solving this recommendation problem by using the Matrix Factorization technique instead of the complicated & computational Neural Networks. This report explains and describes the two types of recommender system methods in brief and later the implementation of the algorithm on a small piece of the dataset.

**Content:** This is the dataset that was used in that competition.

## 1. TRAINING DATASET FILE DESCRIPTION

The file "training\_set.tar" is a tar of a directory containing 17770 files, one per movie. The first line of each file contains the movie id followed by a colon. Each subsequent line in the file corresponds to a rating from a customer and its date in the following format:

CustomerID,Rating,Date

- MovieIDs range from 1 to 17770 sequentially.
- CustomerIDs range from 1 to 2649429, with gaps. There are 480189 users.
- Ratings are on a five star (integral) scale from 1 to 5.
- Dates have the format YYYY-MM-DD.

## 2. MOVIES FILE DESCRIPTION

Movie information in "movie\_titles.txt" is in the following format:

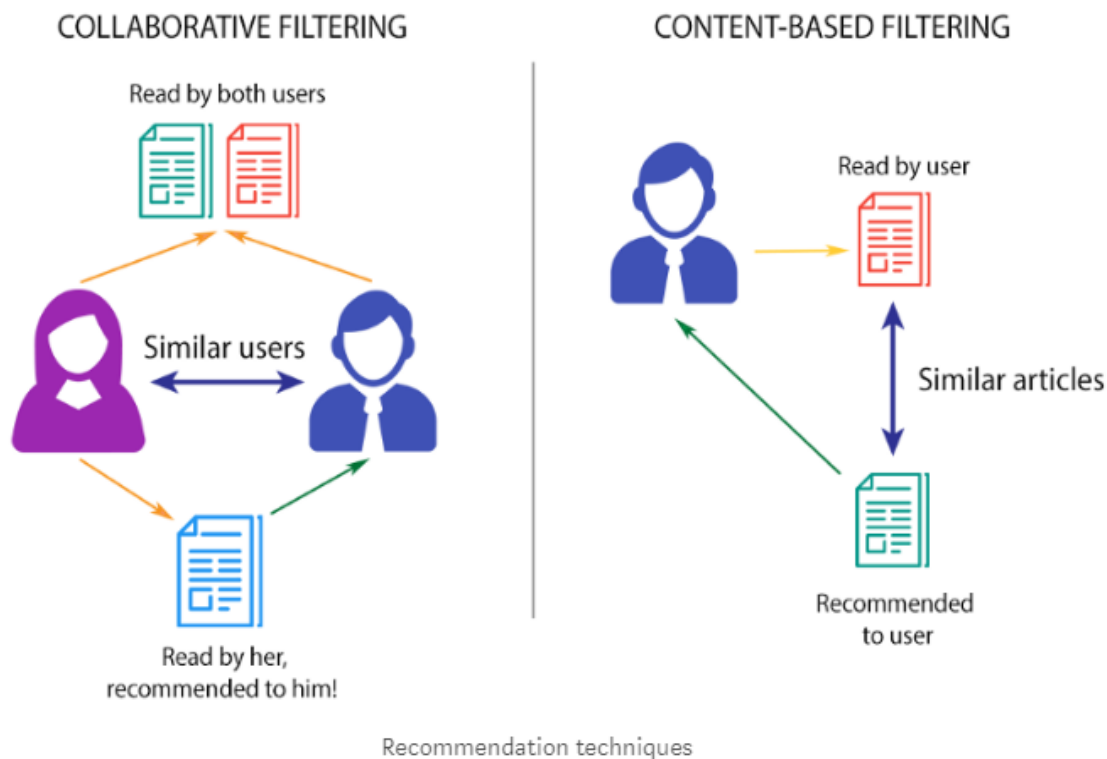
MovieID,YearOfRelease,Title

- MovieID do not correspond to actual Netflix movie ids or IMDB movie ids.
- YearOfRelease can range from 1890 to 2005 and may correspond to the release of corresponding DVD, not necessarily its theatrical release.
- Title is the Netflix movie title and may not correspond to titles used on other sites. Titles are in English.

## 3. IMDB Datasets: <https://datasets.imdbws.com/>

Explanation of each dataset: <https://www.imdb.com/interfaces/>

**Recommender Systems:** Nowadays, companies like Amazon, Walmart, Pandora, Netflix, Spotify use recommender systems to assist their consumers identify new and relevant products to suffice their requirements. It helps provide a seamless experience for the users while consuming the corresponding item/product. Users are always on the lookout for new products in this dynamic world. The recommender engine does justice to it by capturing their characteristics, likes/dislikes based on purchasing history and hence, filtering out the closely related products which might arise their interest and keep them at the edge of their seats wanting for more. At the same time, it helps boost the seller revenue and online web traffic. Here are the two major types of systems:



Type of methods-

1. **Content Based Filtering:** The system finds the similarity between items based on its context, description and its characteristics. The user's previous history is taken into account to find similar products the user may like. For example, if a user likes movies such as 'The Notebook' then we can recommend him similar romantic movies like 'PS. I Love You'. Filtering can be

performed based on the description or if extensive dataset is available, we can deep dive into the characteristics of the movie like genre, movie time, year, top actors/cast/support cast, ratings, reviews.

It used the concept of Cosine Similarity:

The dot product between two vectors is equal to the projection of one of them on the other. Therefore, the dot product between two identical vectors (i.e. with identical components) is equal to their squared module, while if the two are perpendicular (i.e. they do not share any directions), the dot product is zero. Generally, for  $n$ -dimensional vectors, the dot product can be calculated as shown below:

$$\mathbf{u} \cdot \mathbf{v} = [u_1 \ u_2 \ \dots \ u_n] \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = u_1 v_1 + u_2 v_2 + \dots + u_n v_n = \sum_{i=1}^n u_i v_i$$

Dot product.

The dot product is important when defining the similarity, as it is directly connected to it. The definition of similarity between two vectors  $\mathbf{u}$  and  $\mathbf{v}$  is, in fact, the ratio between their dot product and the product of their magnitudes.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|} = \frac{\sum_{i=1}^n u_i v_i}{\sqrt{\sum_{i=1}^n u_i^2} \sqrt{\sum_{i=1}^n v_i^2}}$$

By applying the definition of similarity, this will be in fact equal to 1 if the two vectors are identical, and it will be 0 if the two are orthogonal. In other words, the similarity is a number bounded between 0 and 1 that tells us how much the two vectors are similar.

IMPLEMENTATION:

i) **Gather the data:** Reading the 17000 files containing movie ratings at customer level (Training Dataset). I join it with the 'Movie File Description' data based on 'movie\_title' field. Further, we merge this dataset with the IMDB datasets to fetch the corresponding characteristics of each movie.

Out[24]:

	year	movie_name	ratings	runtimeMinutes	genres	region	language	directors	writers	primaryName	knownForTitles
0	1915	Les Vampires	3.314961	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	1916	20000 Leagues Under the Sea	3.572047	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	1916	Intolerance	3.465849	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	1918	Broken Blossoms	3.399425	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	1918	Chaplin	3.109504	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

ii) **Data Cleaning:** We concatenate the string of words in each of the columns except the movie\_name and year to create a single column 'bag of words'.

```
In [25]: ## concatenation & removing of spaces & creation of final dataframe with 'movie name' & all its characteristics
## in second field

df11 = pd.DataFrame( df10.year.astype(str).str.cat(df10.movie_name.astype(str), sep='_') )
df10['primaryName'] = df10['primaryName'].str.replace(' ', '')
df11['bagOfWords'] = df10['ratings'].map(str) + ' ' + df10['runtimeMinutes'].map(str) + ' ' + df10['genres'].map(str) + ' ' + df10['region'].map(str)
## df11 = df10.drop(['nconst', 'tconst'], axis=1)
df11.columns = ['movie', 'bagOfWords']
df11.head()
```

Out[25]:

	movie	bagOfWords
0	1915_Les Vampires	3.3149606299212597 nan nan nan nan nan nan...
1	1916_20000 Leagues Under the Sea	3.5720470006184293 nan nan nan nan nan nan...
2	1916_Intolerance	3.4658493870402802 nan nan nan nan nan nan...
3	1918_Broken Blossoms	3.3994252873563218 nan nan nan nan nan nan...
4	1918_Chaplin	3.1095041322314048 nan nan nan nan nan nan...

iii) **Modeling:** We use countVectorizer to create vectors of fields and cosine\_similarity function to take a cosine computation of the two vectors and test for similarity later. We get the following matrix:

$$\begin{matrix} & \begin{matrix} Movie_1 & Movie_2 & Movie_3 & \cdots & Movie_n \end{matrix} \\ \begin{matrix} Movie_1 \\ Movie_2 \\ Movie_3 \\ \vdots \\ Movie_n \end{matrix} & \begin{pmatrix} 1 & 0.158 & 0.138 & \cdots & 0.056 \\ 0.158 & 1 & 0.367 & \cdots & 0.056 \\ 0.138 & 0.367 & 1 & \cdots & 0.049 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0.056 & 0.056 & 0.049 & \cdots & 1 \end{pmatrix} \end{matrix}$$

Similarity matrix.

```

In [33]: from sklearn.metrics.pairwise import cosine_similarity
         from sklearn.feature_extraction.text import CountVectorizer

         # instantiating and generating the count matrix
         count = CountVectorizer()
         count_matrix = count.fit_transform(df12['bagOfWords'])

         # generating the cosine similarity matrix
         cosine_sim = cosine_similarity(count_matrix, count_matrix)

In [34]: print(cosine_sim)

[[1.          0.98461538 0.98461538 ... 0.          0.          0.          ]
 [0.98461538 1.          0.98461538 ... 0.          0.          0.          ]
 [0.98461538 0.98461538 1.          ... 0.          0.          0.          ]
 ...
 [0.          0.          0.          ... 1.          0.8         0.75        ]
 [0.          0.          0.          ... 0.8         1.          0.75        ]
 [0.          0.          0.          ... 0.75        0.75        1.          ]]

```

The recommendation function, in fact, once receives the input, detects the 10 highest numbers within the row corresponding to the movie entered, gets the correspondent indexes and matches them to the movie titles series, to return the list of recommended movies.

```

In [35]: # creating a Series for the movie titles so they are associated to an ordered numerical
         # list I will use in the function to match the indexes
         indices = pd.Series(df12.index)

         ## ALGORITHM FOR CONTENT BASES FILTERING
         # defining the function that takes in movie title
         # as input and returns the top 10 recommended movies
         def recommendations(title, cosine_sim = cosine_sim):

             # initializing the empty list of recommended movies
             recommended_movies = []

             # getting the index of the movie that matches the title
             idx = indices[indices == title].index[0]

             # creating a Series with the similarity scores in descending order
             score_series = pd.Series(cosine_sim[idx]).sort_values(ascending = False)

             # getting the indexes of the 10 most similar movies
             top_10_indexes = list(score_series.iloc[1:11].index)

             # populating the list with the titles of the best 10 matching movies
             for i in top_10_indexes:
                 recommended_movies.append(list(df.index)[i])

             return recommended_movies

```

```
In [36]: df13=pd.DataFrame(indices)
df13['movie']=df11['movie']
df13.columns=['indices', 'movie']
df13.head()
```

```
Out[36]:
```

	indices	movie
0	0	1915_Les Vampires
1	1	1916_20000 Leagues Under the Sea
2	2	1916_Intolerance
3	3	1918_Broken Blossoms
4	4	1918_Chaplin

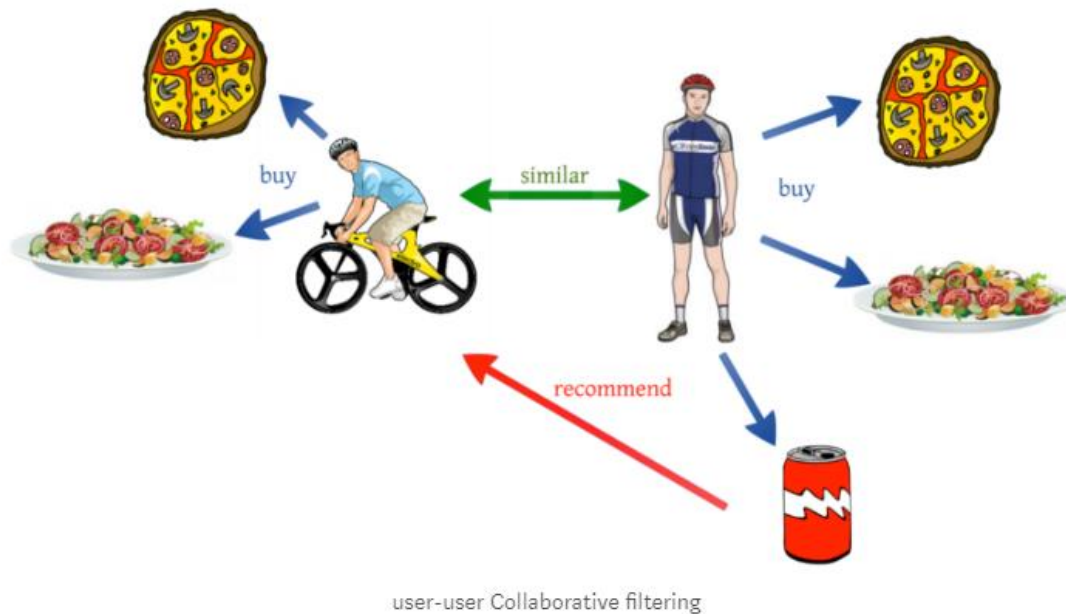
```
In [37]: recommendations(3)
```

```
Out[37]: [0, 1172, 9099, 9100, 5050, 696, 1005, 1006, 1007, 1008]
```

From the above function, we see that we get top 10 recommended similar movies for the movie index 3 i.e. '1918\_Broken Blossoms'

## 2. Collaborative Based Filtering:

Collaborative filtering systems make recommendations based on historic users' preference for items (clicked, watched, purchased, liked, rated, etc.). The preference can be presented as a user-item matrix. History of the user plays an important role. For example, if the user 'A' likes 'Pizzas, and 'Salads' while the user 'B' likes 'Pizzas', 'Salads' and 'Canned Drink' then they have similar interests. So, there is a huge probability that the user 'A' would like 'Canned Drink'. This is the way collaborative filtering is done.





The preference can be presented as a user-item matrix. Here is an example of a matrix describing the preference of 4 users on 5 items, where  $p_{12}$  is the user 1's preference on item 2.

$$P = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} & p_{15} \\ p_{21} & p_{22} & p_{23} & p_{24} & p_{25} \\ p_{31} & p_{32} & p_{33} & p_{34} & p_{35} \\ p_{41} & p_{42} & p_{43} & p_{44} & p_{45} \end{bmatrix}$$

**Matrix factorization** is to, obviously, factorize a matrix, i.e. to find out two (or more) matrices such that when you multiply them you will get back the original matrix. From an application point of view, matrix factorization can be used to discover latent features underlying the interactions between two different kinds of entities.

## Matrix Factorization

	M1	M2	M3	M4	M5
 Comedy	3	1	1	3	1
 Action	1	2	4	1	3

	 Comedy	 Action
A 	✓	✗
B 	✗	✓
C 	✓	✗
D 	✓	✓

	M1	M2	M3	M4	M5
A 	3	1	1	3	1
B 	1	2	4	1	3
C 	3	1	1	3	1
D 	4	3	5	4	4

Given that each users have rated some items in the system, we would like to predict how the users would rate the items that they have not yet rated, such that we can make recommendations to the users.

	M1	M2	M3	M4	M5
A 	3		1		1
B 	1		4	1	
C 	3	1		3	1
D 		3		4	4

Hence, the task of predicting the missing ratings can be considered as filling in the blanks (the hyphens in the matrix) such that the values would be consistent with the existing ratings in the matrix.



### The mathematics of matrix factorization:

Firstly, we have a set  $U$  of users, and a set  $D$  of items. Let  $\mathbf{R}$  of size  $|U| \times |D|$  be the matrix that contains all the ratings that the users have assigned to the items. Also, we assume that we would like to discover  $K$  latent features. Our task, then, is to find two matrices  $\mathbf{P}$  (a  $|U| \times K$  matrix) and  $\mathbf{Q}$  (a  $|D| \times K$  matrix) such that their product approximates  $\mathbf{R}$ :

$$\mathbf{R} \approx \mathbf{P} \times \mathbf{Q}^T = \hat{\mathbf{R}}$$

In this way, each row of  $\mathbf{P}$  would represent the strength of the associations between a user and the features. Similarly, each row of  $\mathbf{Q}$  would represent the strength of the associations between an item and the features. To get the prediction of a rating of an item  $d_j$  by  $u_i$ , we can calculate the dot product of the two vectors corresponding to  $u_i$  and  $d_j$ :

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^K p_{ik} q_{kj}$$

Now, we have to find a way to obtain  $\mathbf{P}$  and  $\mathbf{Q}$ . One way to approach this problem is the first initialize the two matrices with some values, calculate how 'different' their product is to  $\mathbf{M}$ , and then try to minimize this difference iteratively. Such a method is called **gradient descent**, aiming at finding a local minimum of the difference.

The difference here, usually called the error between the estimated rating and the real rating, can be calculated by the following equation for each user-item pair:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^K p_{ik} q_{kj})^2$$

Here we consider the squared error because the estimated rating can be either higher or lower than the real rating.

### Regularization

The above algorithm is a very basic algorithm for factorizing a matrix. There are a lot of methods to make things look more complicated. A common extension to this basic algorithm is to introduce regularization to avoid overfitting. This is done by adding a parameter  $\beta$  to capture noise and modify the squared error as follows:

$$e_{ij}^2 = (r_{ij} - \sum_{k=1}^K p_{ik}q_{kj})^2 + \frac{\beta}{2} \sum_{k=1}^K (\|P\|^2 + \|Q\|^2)$$

```
In [115]: ## ALGORITHM FOR COLLABORATIVE BASED FILTERING - Matrix Factorization
import numpy

def matrix_factorization(R, P, Q, K, steps=5000, alpha=0.0002, beta=0.02):
    Q = Q.T
    for step in range(steps):
        for i in range(len(R)):
            for j in range(len(R[i])):
                if R[i][j] > 0:
                    eij = R[i][j] - numpy.dot(P[i,:],Q[:,j])
                    for k in range(K):
                        P[i][k] = P[i][k] + alpha * (2 * eij * Q[k][j] - beta * P[i][k])
                        Q[k][j] = Q[k][j] + alpha * (2 * eij * P[i][k] - beta * Q[k][j])
    eR = numpy.dot(P,Q)
    e = 0
    for i in range(len(R)):
        for j in range(len(R[i])):
            if R[i][j] > 0:
                e = e + pow(R[i][j] - numpy.dot(P[i,:],Q[:,j]), 2)
                for k in range(K):
                    e = e + (beta/2) * (pow(P[i][k],2) + pow(Q[k][j],2))

    if e < 0.001:
        break
    return P, Q.T
```

```
In [*]: N = len(R)
M = len(R[0])
K = 2

P = numpy.random.rand(N,K)
Q = numpy.random.rand(M,K)

nP, nQ = matrix_factorization(R, P, Q, K)
nR = numpy.dot(nP, nQ.T)
```

```
In [ ]:
```

```
In [143]: nR[0:10]
```

```
In [149]: nR[0:10]
```

```
Out[149]: array([[2.92682866, 3.20431696, 2.91443719, 2.30530473, 2.92097717,
3.02209524, 1.42878326, 2.73790652, 1.85763084, 3.0041128 ,
2.03293342, 2.66461818, 3.52871035, 2.94001052, 2.2023178 ,
2.32480922, 2.44226385, 2.80499489, 2.93187128, 3.13288255,
2.31580371, 1.49385164, 2.84842603, 2.37225729, 3.16739883,
2.38927379, 2.19787561, 3.05444978, 2.77073642, 2.97006243,
2.35540326, 3.51026923, 2.88715389, 2.64808315, 2.4345027 ,
2.59090581, 3.10525294, 1.6538087 , 1.97567632, 2.00175195,
1.52539591, 2.37365215, 2.22927937, 2.67179322, 2.16236379,
2.7339392 , 2.52672851, 2.62057596, 4.3539779 , 2.18006228,
2.57969063],
[4.10104431, 4.59707979, 3.91133326, 3.30224872, 4.98989541,
4.97234627, 2.03373531, 4.66275928, 2.9005851 , 4.59840295,
2.81462481, 4.00531766, 5.0338387 , 4.78495478, 2.68370848,
3.02814063, 3.94341702, 3.80775381, 4.76007561, 4.30415824,
3.1962994 , 1.90227637, 4.66599074, 3.32570436, 4.5058678 ,
4.00511033, 3.62913903, 4.81333167, 4.66272608, 4.41651298,
3.11207798, 5.89385278, 3.72878523, 3.95104138, 4.08002399,
3.61192656, 4.90003864, 1.94853022, 2.74353292, 2.56462799,
2.4034522 , 2.84543403, 2.47804160, 2.60030200, 2.81033000])
```

**Conclusion:** Hence, we receive an optimum solution with the least possible error using gradient descent to optimize Matrix Factorization algorithm. We can see that for existing ratings we have the approximations very close to the true values, and we also get some 'predictions' of the unknown values. Accurate recommendations cannot be made for new users/items with no or little information. This is referred to as the **cold start problem**. This is a typical issue for collaborative filtering systems that rely on user-item interactions. Some heuristics can be used. For a new user, the most popular items in the user's area could be recommended. For a new item, some rule-based similarity criteria can be defined. Besides, scalability, sampling, interoperability are few other challenges faced while implementing such recommender systems.

**REFERENCES:** <https://towardsdatascience.com/how-to-build-from-scratch-a-content-based-movie-recommender-with-natural-language-processing-25ad400eb243>

<https://www.youtube.com/watch?v=ZspR5PZemcs>

<http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/>