Movie Recommendation system

With both collaborative and content-based filtering using python programming language

By: Kimia Esmaili

Professor: AmirHoshang Hoseinpour Dehkordi

January 2021

Table of contents:

Abstract:

In this project we made a movie recommendation system using python programming language. Our project was done with both filtering methods out there for a recommending system: collaborative and content-based. For the collaborative filtering, we tried three different methods: user-user based, item-item based and SVD (singular value decomposition).

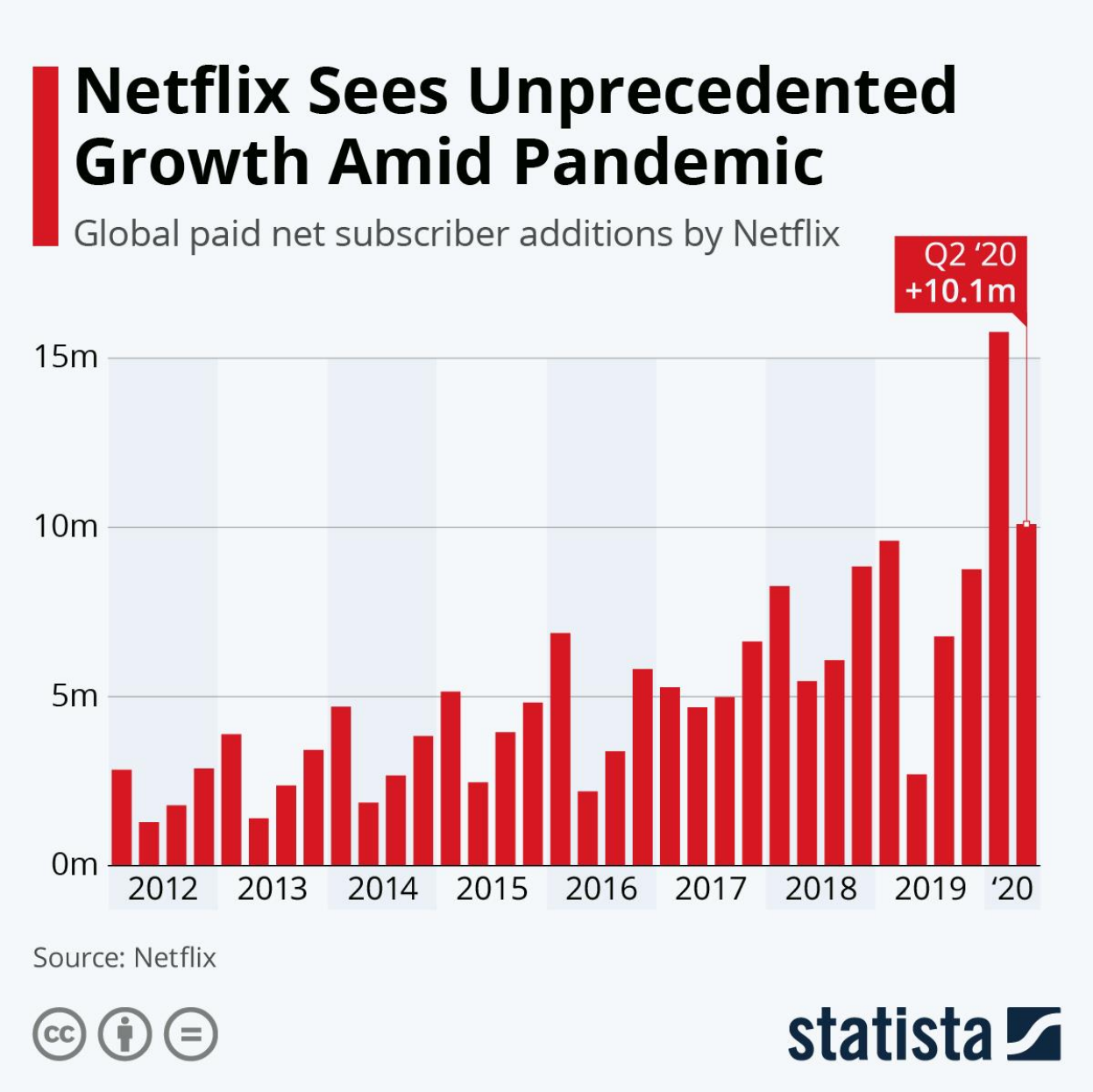
These methods were used to find the most convenient way to give a recommendation for a user.

In the content-based filtering, we used NLP (natural language processing) to help us use the information in our dataset in a faster, more precise and more effective manner. (disclaimer: some words such as “function”, “method”, etc. are used based on the context of the sentence rather than their usage in python, so they don’t necessarily have the same meaning.)

Introduction:

Today, the world is marching fast in the way of communication and technology. The idea of “everything by your side with just a click/tap” is the appealing motto of pioneer companies of technology and science. The goal of these companies is to attract more people to their platform and provide them with the content they gravitate to, so that they remain loyal to them and continue their subscription. For study case we have companies like Spotify, Netflix and Amazon (via amazon prime) that are the giants of online-streaming services. With the recent pandemic and forced quarantine, these companies (we are getting more specific with Netflix) seemed to triumph even more due to the lockdown of theaters and people’s constant boredom at home as a consequence to this new quarantine situation. Leading content streaming service Netflix has added 10.1 million new paid

subscribers as people stayed home, as the company reported net earnings of $720 million over $6.15 billion in revenue for its second quarter (April-June period).



As mentioned earlier, streaming services can continue having subscribers and growth in their stock if and only if they provide their subscribers with their desired material. Almost every streaming platform shares the same ace: a good recommendation system for their users. This feature is tempting and is the reason why you can't stop watching random videos that are suggested for you (and you seem to really enjoy them) on Youtube. With the growth of home-friendly streaming companies and the huge amount of money they make by making us

addicted to them, we can only imagine how important it must be to have a good recommendation system for our users.

Types of filtering:

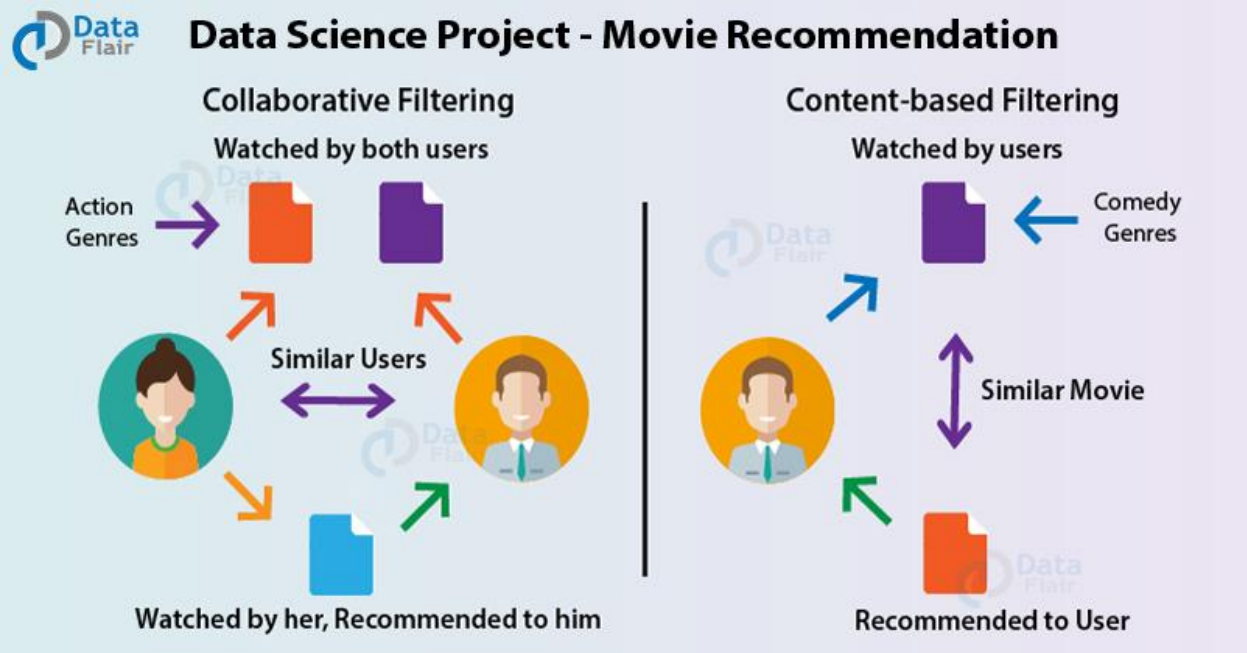
There are majorly six types of Recommender Systems which work primarily in the media and entertainment industry: Collaborative Recommender System, Content-based recommender system, demographic based recommender system, utility based recommender system, knowledge based recommender system and hybrid recommender system.

Component Procedures of a Recommender which are mostly 3:

1. **Candidate Generations:** This method is responsible for generating smaller subsets of candidates to recommend to a user, given a huge pool of thousands of items.
2. **Scoring Systems**: Candidate Generations can be done by different Generators, so, we need to standardize everything and try to assign a score to each of the items in the subsets. This is done by the Scoring system.
3. **Re-Ranking Systems:** After the scoring is done, along with it the system takes into account other additional constraints to produce the final rankings.

Types of Candidate Generation Systems:

1. Collaborative filtering System
2. Content-based filtering System



Collaborative filtering System:

Collaborative does not need the features of the items to be given. Every user and item is described by a feature vector or embedding.It creates embedding for both users and items on its own. It embeds both users and items in the same embedding space.It considers other users' reactions while recommending a particular user. It notes which items a particular user likes and also the items that the users with behavior and likings like him/her likes, to recommend items to that user.It collects user feedbacks on different items and uses them for recommendations.

Sources of user-item interactions:

Implicit Feedback: The user's likes and dislikes are noted and recorded on the basis of his/her actions like clicks, searches, and purchases. They are found in abundance but negative feedback is not found.

Explicit Feedback: The user specifies his/her likes or dislikes by actions like reacting to an item or rating it. It has both positive and negative feedback but less in number

Types of collaborative Recommender Systems:

I. Memory-based collaborative filtering:

Done mainly remembering the user-item interaction matrix, and how a user reacts to it, i.e, the rating that a user gives to an item. There is no dimensionality reduction or model fitting as such. Mainly two sections:

i. **User-User filtering:** In this kind, if a user A's characteristics are similar to some other user B then, the products that B liked are recommended to A. As a statement, we can say, "the users who like products similar to you also liked those products". So here we recommend using the similarities between two users. Now, if one user A behaves like other users B, C, and D, then for a product x, A's rating is given by: $R_{xu}=(\Sigma_{i=0}^{n}R_i) / n$

Where Rxu is the rating given to x by user u and i=0 to n are the users who have shown behavior similar to u. Now, all the n users are not an equal amount similar to the user u. So, we find a weighted sum to provide the rank.

$$R_{xu}=(\Sigma_{i=0}^{n}R_iW_i) / \Sigma_{i=0}^{n}W_i$$

The weights here are the similarity metrics used.

Now, users show some differences in behaviors while rating. Some are generous raters, others are not, i.e, maybe one user rates in range 3 to 5, while other user rates 1 to 3. So, we calculate the average of all the ratings that the user has provided, and subtract the value from Ri in order to normalize the ratings by each user.

ii. **Item-Item filtering:** Here, if user A likes an item x, then, the items y and z which are similar to x in property, then y and z are recommended to the user. As a statement, it can be said, "Because you liked this, you may also like those".

The same equations are used here also:

$$R_{xu}=(\Sigma_{i=0}^{n}R_i) / n$$

Where R is the rating user u gives to the product x, and it is the average of the ratings u gave to products like x. Here also, we take a weighted average:

$$R_{xu}=(\Sigma_{i=0}^{n}R_iW_i) / \Sigma_{i=0}^{n}W_i$$

Where the Weight is the similarity between the products.

Similarity Metrics:

They are mathematical measures which are used to determine how similar is a vector to a given vector. Similarity metrics used mostly:

1. Cosine Similarity: The Cosine angle between the vectors.
2. Dot Product: The cosine angle and magnitude of the vectors also matters.
3. Euclidian Distance: The elementwise squared distance between two vectors
4. Pearson Similarity: It is a coefficient given by:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

II. Model-based collaborative filtering:

Remembering the matrix is not required here. From the matrix, we try to learn how a specific user or an item behaves. We compress the large interaction matrix using dimensional Reduction or using clustering algorithms. In this type, We fit machine learning models and try to predict how many ratings will a user give a product. There are several methods:

1. **Clustering algorithms**
2. **Matrix Factorization based algorithm**
3. **Deep Learning methods**

Clustering Algorithms: They normally use simple clustering Algorithms like K-Nearest Neighbours to find the K closest neighbors or embeddings given a user or an item embedding based on the similarity metrics used.

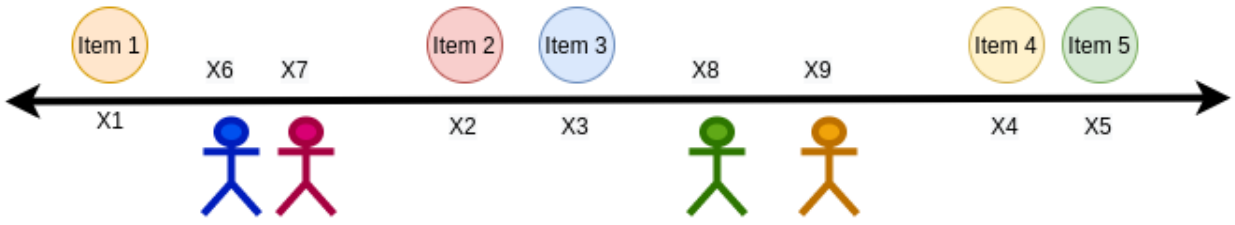
Matrix Factorization based algorithms:

Idea: Like any big number can be factorized into smaller numbers, the user-item interaction table or matrix can also be factorized into two smaller matrices, and these two matrices can also be used to generate back the interaction matrix.

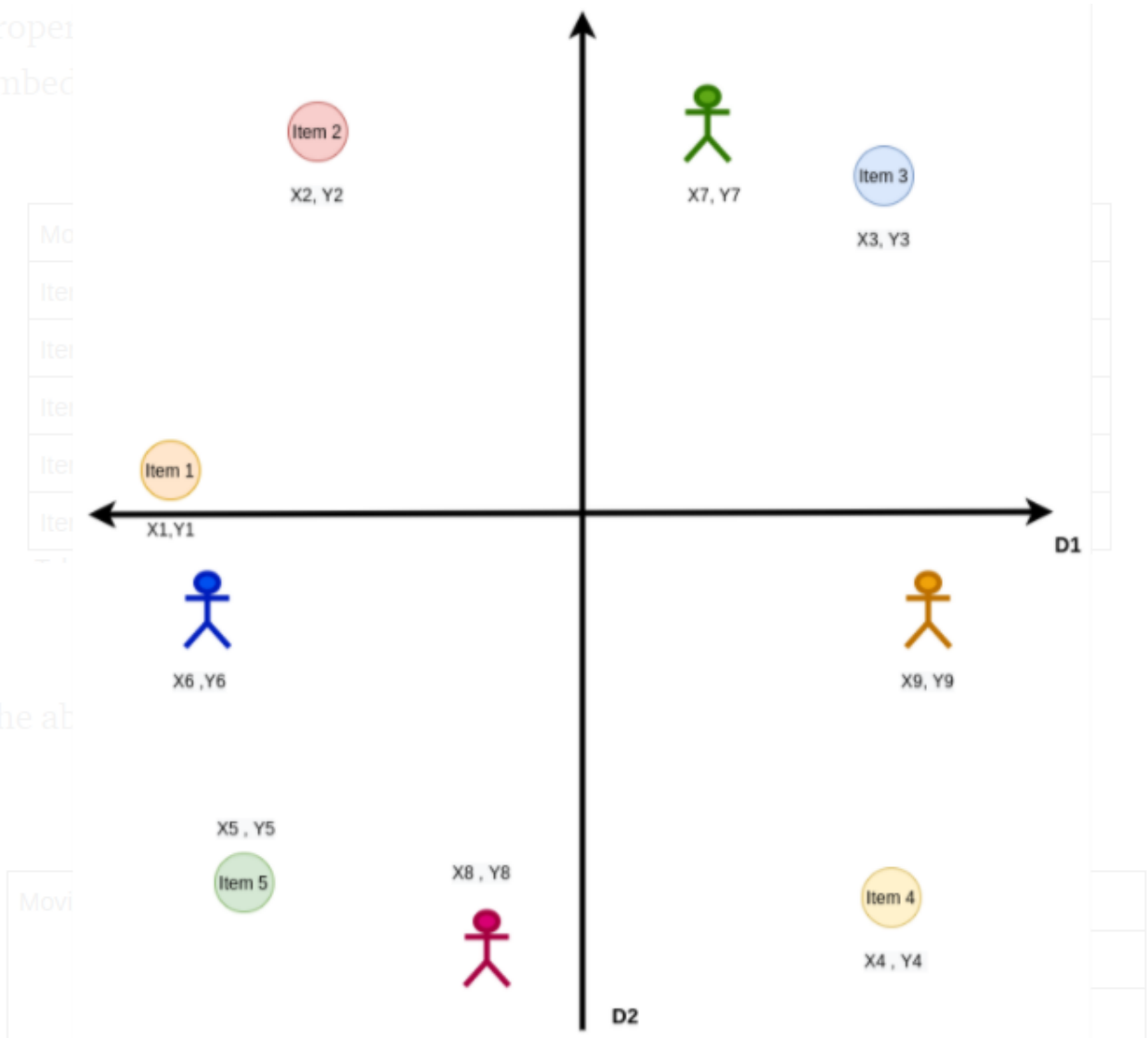
So, we generate the factor matrices as feature matrices for users and items. These feature matrices serve as embeddings for each user and item. To create the feature matrices we need dimensional reduction.

Say,

There are 4 users and 5 items, the users and items are placed according to a domain D1 say, genre if items are movies. We can say they are not very well separable and the whole thing looks very generalized.



So, we increase the number of domains or add a domain, on whose basis we can classify the users and the items.



Now using these two domains we can easily classify the items and users properly, so, the (x,y) pairs can be used as their feature vectors or embeddings. Thus, our matrix is factorized.

Movies	User 1	User 2	User 3	User 4
Item 1	1		4	5
Item 2	5	4	1	2
Item 3	4	4		3
Item 4	2	2	4	
Item 5		5	3	2

Table 1

The above interaction table is converted into:

Movies				User 1	User 2	User 3	User 4
				x6	x7	x8	x9
				y6	y7	y8	y9
Item 1	x1	y1		x1.x6+y1.y6	x1.x7+y1.y7	x1.x8+y1.y8	x1.x9+y1.y9
Item 2	x2	y2		x2.x6+y2.y6	x2.x7+y2.y7	x2.x8+y2.y9	x2.x9+y2.y9
Item 3	x3	y3		x3.x6+y3.y6			
Item 4	x4	y4		x4.x6+y4.y6			
Item 5	x5	y5		x5.x6+y5.y6			

Table 2

So, our task is to find the (x,y) values in such a way that the numbers generated in table 2 are as close as the actual interaction matrix. Once we find all the (x,y) values we can also, find the missing values. Now, the missing values are due to the fact that the user has not rated the item. So, if the generated values are good, we can recommend them to the user. Here 2 domains only x and y are shown actually there can be a very large number of domains. More the number of domain bigger the feature vector, bigger the embedding space.

Now, the number of features in the feature vectors depends on how many domains or features (a feature represented in a domain), we need to consider to distinctly represent the users and the items. So, we basically need to find the principal components of the user and items distributions. Finding principal components implies dimensionality reduction, i.e, representing a distribution distinctly using the least number of features possible.

The dimensionality reduction can be done by several methods:

1. **SVD: Singular Value Decomposition** (the only Model-based collaborative filtering method we used in our project, other methods are only mentioned and explained here for better clarification and understanding.)
2. **PMF: Probability Matrix Factorization**
3. **NMF: Non-Negative Matrix Factorization**

If we observe table 2, x1.x6+y1.y6, is the dot product of item 1 embedding vector multiplied by [x6, y6] i.e, transpose of the embedding vector of user 1.

So, each cell in table 2,

Rating to item u by user v= U.transpose(V)

Dimensionality Reduction(Creating Feature Vectors):

Movies	User 1	User 2	User 3	User 4
Item 1	1	0	1	1
Item 2	1	1	1	1
Item 3	1	1	0	1
Item 4	1	1	1	0
Item 5	0	1	1	1

Table 3

1 denotes the user reacted to the item or rated it and 0 means the user did not. Here the number of 1 much greater than the number of 0. In the real-world, actually the number of 0's much greater than the number of 1s. So, mostly they are sparse matrices.

Our objective function, i,e the one we need to minimize here is:

$$\text{Loss}= \Sigma_i\Sigma_j(A_{ij} - U_iV_j^T)^2 \quad \text{for } r(i,j)=1$$

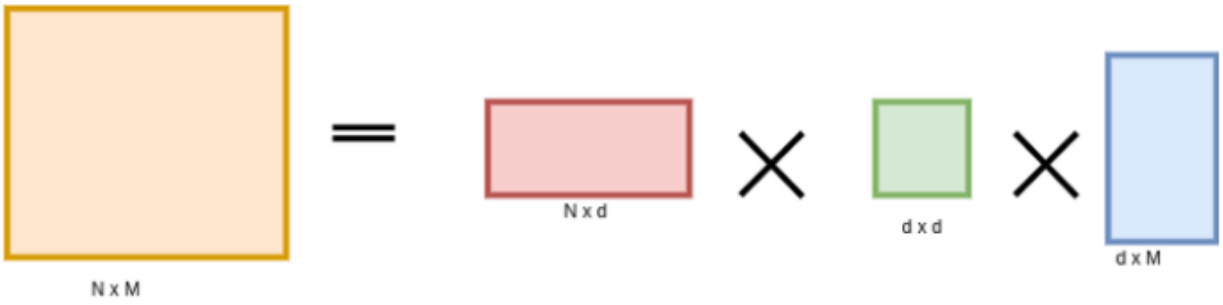
r(i,j)=1 can be found from table 3. It implies to the users j who have reacted or rated items i. If r(i,j)=0 user j have not rated item i.

So, it means we are trying to minimize the difference between the original rating given A(i,j) from table 1 and the values obtained from table 2, multiplying the user and item's feature vectors. This helps to optimize the feature vectors.

For SVD:

$$\text{Loss}= \Sigma_i\Sigma_j(A_{ij} - U_iV_j^T)^2$$

where Aij is the cell wise rated value in table 1 and u, v are the values generated by the algorithm. Now, due to the sparsity of the matrix SVD does not perform that well here. SVD is given by R=UΣV, where U is the item matrix of dimension n x d, i.e, total n items and V has dimension m x d i.e, total m users, Σ is d x d diagonal matrix for multiplication compatability. D is the size of the feature vector for each user and item also. In SVD, the matrix is decomposed as:



Where N is the number of items, M is the number of users and d is the dimension or size of the feature vector. The middle one is the diagonal vector.

NMF: Non-negative matrix factorization is so-called because the matrix here has no negative components, i.e, ratings can never be negative. The ones not rated is considered as 0's:

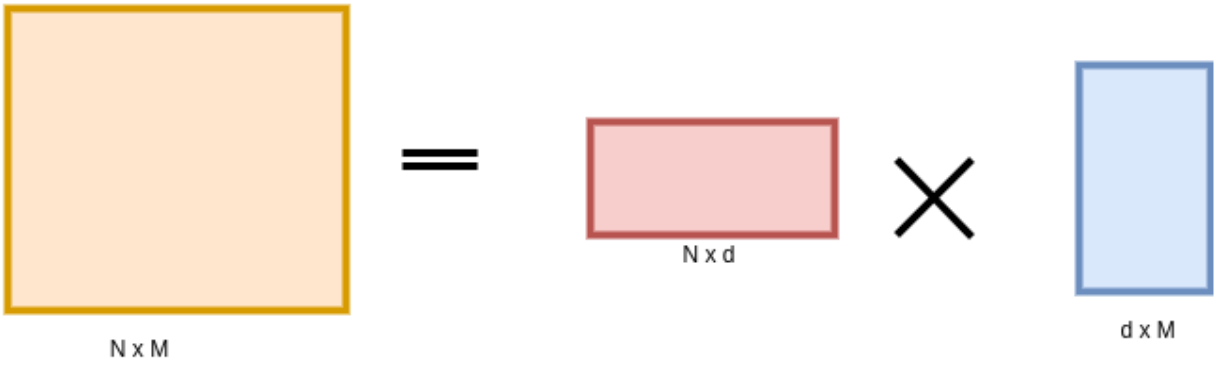
Movies	User 1	User 2	User 3	User 4
Item 1	1	0	4	5
Item 2	5	4	1	2
Item 3	4	4	0	3
Item 4	2	2	4	0
Item 5	0	5	3	2

Table 4

So, NMF uses only the observed or rated ones. So, it modifies the function as:

$$\text{Loss} = \Sigma_i \Sigma_j (A_{ij} - U_i V_j^T)^2 \quad \text{for } r(i,j)=1 \text{ or } (i,j) \in \text{(obs/rated)}$$

This performs better with sparse matrices. It breaks the vector as:



Where N is the number of users, M is the number of items and d is the dimension or size of the feature vector. There is another variation of Matrix factorization. It is called **Weighted matrix factorization.** It modifies the loss function as:

$$\text{Loss} = W_0 \Sigma_i \Sigma_j (A_{ij} - U_i V_j^T)^2 \quad \text{for } r(i,j)=1 \text{ or } (i,j) \in \text{(rated)} + W_1 \quad \Sigma_i \Sigma_j (U_i V_j^T)^2 \quad \text{for } r(i,j)=0 \text{ or } (i,j) \notin \text{(rated)}$$

Here, we include two rated terms w0 and w1 and try to optimize the rated as well as the non-rated ones. The non-rated ones are considered to be zero, so, ((Ui.Vj)-0)2 is used to optimize the non-rated or zero ones. W0 and W1 are hyperparameters, we need to choose carefully.

Minimizing the Objective function: The most common algorithm used to minimize the include:

Weighted Alternating Least Squares: If we concentrate on the problem, we can find two separate problems: Finding the optimal embeddings that best describe the items and also the optimal embeddings that best describe the users.

Now, let's decompose and see the problems individually,

1. If we have the feature matrices or vectors for the items as we did for the Content-Based systems, we can easily find the user embeddings or feature sets by noting how a user reacts or rates items having different feature vectors.

So, our target function becomes:

$$\Sigma_i^m \Sigma^n_j (A_{ij} - U_i V_j^T)^2 \text{ for } r(i,j)=1 \text{ or } (i,j) \in (\text{rated}) + \lambda/2 \, \Sigma^n_j (V_j)^2$$

Where Vj is the user's feature vector and the lambda term is the regularization term for optimizing the user embeddings, given the item embedding or feature vector Ui for each item i.

2. Now, if we have the user features or know how a user behaves or reacts, we can optimally find the item's feature or embedding vectors, from the way each user reacts or rates the item. Our target function for this:

$$\Sigma^n_j \Sigma_i^m (A_{ij} - U_i V_j^T)^2 \text{ for } r(i,j)=1 \text{ or } (i,j) \in (\text{rated}) + \lambda/2 \, \Sigma_i^m (U_i)^2$$

Where Ui is the item's feature vector. We need to find Ui given the user vector's Vj.

Now, In matrix factorization, we need to find both U and V. So, the algorithm WALS works by alternating between the above two equations.

- Fixing U and solving for V.
- Fixing V and solving for U.

Now, the problem is these two equations are not convex at the same time. Either equation 1 is convex or equation 2 but not combined. As a result, we can't reach a global minimum here, but it has been observed that reaching a local minimum close to the global minimum gives us a good approximation of the optimized results at the global minimum. So, this algorithm gives us an approximated result.

Challenges:

1. The prediction of the model for a given (user, item) pair is the dot product of the corresponding embeddings. So, if an item is not seen during training, the system can't create an embedding for it and can't query the model with this item. This issue is often called the **cold-start problem**.

This problem is generally solved using two methods:

1. Projection in WALS
2. Heuristics to generate embeddings for fresh items

The side features are hard to include. Side features are the ones that may affect the recommendations like for a movie, the U/PG ratings can be side features or the country.

Content-based filtering system:

Content-Based recommender system tries to guess the features or behavior of a user given the item's features, they reacts positively to:

Movies	User 1	User 2	User 3	User 4	Action	Comedy
Item 1	1		4	5	Yes	No
Item 2	5	4	1	2	No	Yes
Item 3	4	4		3	Yes	Yes
Item 4	2	2	4	4	No	Yes

The last two columns Action and Comedy, describe the Genres of the movies. Now, given these genres, we can know which users like which genre, as a result, we can obtain features corresponding to that particular user, depending on how he/she reacts to movies of that genre. Once we know the likings of the user, we can embed them in an embedding space using the feature vector generated and recommend them according their choice. During recommendation, the similarity metrics are calculated from the item's feature vectors and the user's preferred feature vectors from his/her previous records. Then, the top few are recommended. Content-based filtering does not require other users' data during recommendations to one user.

Our code:

1. **Using Collaborative Filtering:**

We will build a recommender system using the Movie Lens -100k dataset available here: https://grouplens.org/datasets/movielens/100k/. The dataset folder contains a number of files. we will be using the 'ua.base' file which contains 90,000 ratings and the 'ua.test' file which contains 10,000. These files contain the number of a user, number of a movie, user's rating for that movie and the unix timestamp of the rating. (the numbers that are mapped to the movies are shown in the 'u.item' file)

The recommendation system we will build will be user-user based collaborative filtering and item-item based collaborative filtering and later go onto try a model based collaborative filtering using Singular Value Decomposition.
Importing all the libraries:

```
import pandas as pd
import numpy as np
from sklearn.metrics.pairwise import pairwise_distances
from sklearn.metrics import mean_squared_error
import math
```

Pandas library: is a library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

NumPy library: is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. (we import numPy and pandas as "np" and "pd" for our convenience.)

sklearn.metrics.pairwise_distances(): Compute the distance matrix from a vector array X and optional Y. This method takes either a vector array or a distance matrix, and returns a distance matrix. If the input is a vector array, the distances are

computed. If the input is a distances matrix, it is returned instead. This method provides a safe way to take a distance matrix as input, while preserving compatibility with many other algorithms that take a vector array.

sklearn.metrics. mean_squared_error(): yields Mean squared error regression loss.

Math module: is a standard module in Python and to use mathematical functions under this module, we need to import the module.

Now, Reading both the datasets and setting the column names(and then showing the column):

We use **.read_csv()** to read the data from the files and **.head()** which is a function used to return the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it. If n is not specified, 5 rows will be shown by default. Both are features from pandas.

```
r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
ratings_base = pd.read_csv('ml-100k/ua.base', sep='\t', names=r_cols, encoding='latin-1')
ratings_test = pd.read_csv('ml-100k/ua.test', sep='\t', names=r_cols, encoding='latin-1')
```

```
ratings_base.head()
```

	user_id	movie_id	rating	unix_timestamp
0	1	1	5	874965758
1	1	2	3	876893171
2	1	3	4	878542960
3	1	4	3	876893119
4	1	5	3	889751712

The column user_id contains ids' of users starting from 1, the column movie_id contains ids' of users starting from 1 and the 'rating' column contains the corresponding ratings. Let us see how many unique users and how many unique movies(items) are there. We use **.max()** and **.unique()** (from numPy) to do so.

```
n_users_base = ratings_base['user_id'].unique().max()
n_items_base = ratings_base['movie_id'].unique().max()
n_users_base,n_items_base
```

```
(943, 1682)
```

There are 943 users and 1682 movies in the training set.

```
n_users_test = ratings_test['user_id'].unique().max()
n_items_test = ratings_test['movie_id'].unique().max()
n_users_test,n_items_test
```

```
(943, 1664)
```

creating user-item matrix:

There are 943 users and 1664 movies in the training set. Now let us go ahead and create our user-item matrices, test_matrix and train_matrix which contain number of rows equal to the number of unique users and number of columns equal to the number of unique movies. The cells of this matrix are filled with the corresponding rating a user has given to a movie. If a user has not rated a movie then the cell is filled with 0.

np.zerros()return a new array of given shape and type, filled with zeros.
itertuples() from pandas, iterate over DataFrame rows as namedtuples.

```
train_matrix = np.zeros((n_users_base, n_items_base))
for line in ratings_base.itertuples():
    train_matrix[line[1]-1,line[2]-1] = line[3]

test_matrix = np.zeros((n_users_test, n_items_test))
for line in ratings_test.itertuples():
    test_matrix[line[1]-1,line[2]-1] = line[3]
```

Trying user-user based collaborative filtering:

The first approach we try is user-user based collaborative filtering. In this method, we first create a similarity matrix which specifies the similarity between two users based on the ratinngs they have given to different movies. We use the cosine similarity metric which computers the dot product between the two vectors made up of the ratings of the movies they have rated.

```
user_similarity = pairwise_distances(train_matrix, metric='cosine')
print('shape: ',user_similarity.shape)
user_similarity
```

```
shape:  (943, 943)

array([[ 0.        ,  0.85324924,  0.9493235 , ...,  0.96129522,
         0.8272823 ,  0.61960392],
       [ 0.85324924,  0.        ,  0.87419215, ...,  0.82629308,
         0.82681535,  0.91905667],
       [ 0.9493235 ,  0.87419215,  0.        , ...,  0.97201154,
         0.87518372,  0.97030738],
       ...,
       [ 0.96129522,  0.82629308,  0.97201154, ...,  0.        ,
         0.96004871,  0.98085615],
       [ 0.8272823 ,  0.82681535,  0.87518372, ...,  0.96004871,
         0.        ,  0.85528944],
       [ 0.61960392,  0.91905667,  0.97030738, ...,  0.98085615,
         0.85528944,  0.        ]])
```

The similarity matrix has a shape of 943 x 943 as expected with each cell corresponding to the similarity between two users. Now we will write a prediction function which will predict the values in the user-item matrix. We will only consider the top n users which are similar to a user to make predictions for that user. In the formula we normalise the ratings of users by subtracting the mean rating of a user from every rating given by the user:

$$\hat{x}_{k,m} = \bar{x}_k + \frac{\sum_{u_a} sim_u(u_k, u_a)(x_{a,m} - \bar{x}_{u_a})}{\sum_{u_a} |sim_u(u_k, u_a)|}$$

.argsort() function (from numPy) is used to perform an indirect sort along the given axis using the algorithm specified by the kind keyword. It returns an array of indices of the same shape as that array that would sort the array.

enumerate() lets us write Pythonic for loops when we need a count and the value from an iterable. The big advantage of **enumerate**() is that it returns a tuple with the counter and value, so we don't have to increment the counter ourself.

.mean() gives us the average and **.sum()** gives us the summation. **np.newaxis()** is used to increase the dimension of the existing array by *one more dimension*, when used *once*. The **.T** accesses the attribute T of the object, which happens to be a NumPy array. The T attribute is the transpose of the array. **np.dot()** returns the dot product of vectors a and b, and also 2D arrays as matrix with performing matrix multiplication.

```
def predict_user_user(train_matrix, user_similarity, n_similar=30):
    similar_n = user_similarity.argsort()[:,-n_similar:][:,::-1]
    pred = np.zeros((n_users_base,n_items_base))
    for i,users in enumerate(similar_n):
        similar_users_indexes = users
        similarity_n = user_similarity[i,similar_users_indexes]
        matrix_n = train_matrix[similar_users_indexes,:]
        rated_items = similarity_n[:,np.newaxis].T.dot(matrix_n - matrix_n.mean(axis=1)[:,np.newaxis])/ similarity_n.sum()
        pred[i,:]  = rated_items
    return pred
```

We will use this function to find the predicted ratings and add the average rating of every use to give back the final predicted ratings. Here, we are considering the top 50 users which are similar to our user and using their ratings to predict our user's ratings:

```
predictions = predict_user_user(train_matrix,user_similarity, 50) + train_matrix.mean(axis=1)[:, np.newaxis]
print('predictions shape ',predictions.shape)
predictions
```

```
predictions shape  (943, 1682)

array([[ 0.53079191,  0.53079191,  0.53079191, ...,  0.53079191,
         0.53079191,  0.53079191],
       [ 0.27556554,  0.17581381, -0.00189689, ..., -0.00189689,
        -0.00189689, -0.00189689],
       [ 1.17064209,  0.07064209,  0.01064209, ...,  0.01064209,
         0.01064209,  0.01064209],
       ...,
       [-0.0479786 , -0.0479786 , -0.0479786 , ..., -0.0479786 ,
        -0.0479786 , -0.0479786 ],
       [ 0.8909642 ,  0.12995357,  0.12995357, ...,  0.12995357,
         0.12995357,  0.12995357],
       [ 0.27315101,  0.27315101,  0.27315101, ...,  0.31315101,
         0.27315101,  0.27315101]])
```

The .**shape** attribute for numPy arrays returns the dimensions of the array.

Let us consider only those ratings which are not zero(rated) in the test matrix and use them to find the error in our model: (**np.nonzero()** function is used to Compute the indices of the elements that are non-zero.)

```
predicted_ratings = predictions[test_matrix.nonzero()]
test_truth = test_matrix[test_matrix.nonzero()]
```

```
math.sqrt(mean_squared_error(predicted_ratings,test_truth))
```

```
3.507744099069281
```

Trying item-item based collaborative filtering:

Now, we will go on and try item-item based collaborative filtering. This method finds the similarity between items instead of users, exactly like the previous method using 'cosine similarity'. Using the similarity between items and the users rating for similar items, we find the predicted ratings for un-rated items. Let us make the item similarity matrix:

```
item_similarity = pairwise_distances(train_matrix.T, metric = 'cosine')
item_similarity.shape
```

```
(1682, 1682)
```

```
item_similarity
```

```
array([[ 0.        ,  0.59704074,  0.66673863, ...,  1.        ,
         0.94919585,  0.94919585],
       [ 0.59704074,  0.        ,  0.7308149 , ...,  1.        ,
         0.91844091,  0.91844091],
       [ 0.66673863,  0.7308149 ,  0.        , ...,  1.        ,
         1.        ,  0.90098525],
       ...,
       [ 1.        ,  1.        ,  1.        , ...,  0.        ,
         1.        ,  1.        ],
       [ 0.94919585,  0.91844091,  1.        , ...,  1.        ,
         0.        ,  1.        ],
       [ 0.94919585,  0.91844091,  0.90098525, ...,  1.        ,
         1.        ,  0.        ]])
```

The similarity matrix has a shape of 1682 x 1682 as expected with each cell corresponding to the similarity between two users. Now we will write a prediction function which will predict the values in the user-item matrix. We will only consider the top n items which are similar to a item to make predictions.. In this formula we don't need normalise the ratings of users as we are using items to make predictions instead of users.

$$\hat{x}_{k,m} = \frac{\sum_{i_b} sim_i(i_m, i_b)(x_{k,b})}{\sum_{i_b} |sim_i(i_m, i_b)|}$$

This function is similar to the function we previously made in the user-user based filtering:

```
def predict_item_item(train_matrix, item_similarity, n_similar=30):
    similar_n = item_similarity.argsort()[:,-n_similar:][:,::-1]
    print('similar_n shape: ', similar_n.shape)
    pred = np.zeros((n_users_base,n_items_base))

    for i,items in enumerate(similar_n):
        similar_items_indexes = items
        similarity_n = item_similarity[i,similar_items_indexes]
        matrix_n = train_matrix[:,similar_items_indexes]
        rated_items = matrix_n.dot(similarity_n)/similarity_n.sum()
        pred[:,i]  = rated_items
    return pred
```

We will use this function to find the predicted ratings. Here, we are considering the top 50 users which are similar to our user and using their ratings to predict our user's ratings.

```
predictions = predict_item_item(train_matrix,item_similarity,50)
print('predictions shape ',predictions.shape)
predictions
```

```
similar_n shape:  (1682, 50)
predictions shape  (943, 1682)

array([[ 0.  ,  0.  ,  0.  , ...,  0.  ,  0.  ,  0.66],
       [ 0.  ,  0.  ,  0.  , ...,  0.  ,  0.  ,  0.  ],
       [ 0.  ,  0.  ,  0.  , ...,  0.  ,  0.  ,  0.  ],
       ...,
       [ 0.  ,  0.  ,  0.  , ...,  0.  ,  0.  ,  0.  ],
       [ 0.  ,  0.  ,  0.  , ...,  0.1 ,  0.08,  0.08],
       [ 0.  ,  0.  ,  0.  , ...,  0.44,  0.2 ,  0.06]])
```

Let us consider only those ratings which are not zero in the test matrix and use them to find the error in our model:

```
predicted_ratings = predictions[test_matrix.nonzero()]
test_truth = test_matrix[test_matrix.nonzero()]
math.sqrt(mean_squared_error(predicted_ratings,test_truth))
```

```
3.749688827167227
```

Getting recommendations for a user:

In the next part we get recommendations for a user based on the highest predicted ratings for a particular user. Let us get predctions for the user with user id 77. we will be using the predictions from the item-item collaborative filtering model for this:

```
user_id = 77 #or #int(input("enter a user ID:"))
user_ratings = predictions[user_id-1,:]
```

We should subtract 1 from user_id as it is used as an index here(starting from 0)

We extract the indices of the movies in the matrix which have not been rated by the user i.e. value is 0 and get their predticted ratings: (**np.where()** function returns the indices of elements in an input array where the given condition is satisfied.)

```
train_unkown_indices = np.where(train_matrix[user_id-1,:] == 0)[0]
train_unkown_indices
```

```
array([   1,    2,    4, ..., 1679, 1680, 1681], dtype=int64)
```

```
user_recommendations = user_ratings[train_unkown_indices]
```

```
user_recommendations.shape
```

```
(1620,)
```

We go on and print the top 5 recommendations:

```
print('\nRecommendations for user {} are the movies: \n'.format(user_id))
for movie_id in user_recommendations.argsort()[-5:][: : -1]:
    print(movie_id +1)
```

```
Recommendations for user 77 are the movies:

1426
1356
1316
1100
1374
```

Trying singular value decomposition:

After we have tried out both the memory based methods i.e user-user and item-item collaborative filtering, in this method we will try a model-based method. (as mentioned before)Singular value decomposition is a mathematical techinique used to find the missing values in a matrix. It decomposes a matrix into three matrices two of which are rectangular and the middle one is a diagonal matrix:

$$X = U \times S \times V^T$$

Importing the required libraries:

```
import scipy.sparse as sp
from scipy.sparse.linalg import svds
```

Fortunately python has a vast plethora of built in features that make this method easy to implement:

Scipy.sparse: SciPy 2-D sparse matrix package for numeric data.

scipy.sparse.linalg.svds(): Compute the largest or smallest k singular values/vectors for a sparse matrix.

Making the matrices u, s and vt: (the diagonal matrix s is completed later)

```
u, s, vt = svds(train_matrix, k = 20)
```

```
u.shape, s.shape, vt.shape
```

```
((943, 20), (20,), (20, 1682))
```

Turning the matrix “s” into a diagonal matrix using np.diag():

```
s_diag_matrix = np.diag(s)
```

We get the predictions by finding the dot product of the three matrices:

```
predictions_svd = np.dot(np.dot(u,s_diag_matrix),vt)
```

```
predictions_svd.shape
```

```
(943, 1682)
```

```
predicted_ratings_svd = predictions_svd[test_matrix.nonzero()]
test_truth = test_matrix[test_matrix.nonzero()]
math.sqrt(mean_squared_error(predicted_ratings_svd,test_truth))
```

```
2.8258075694458307
```

As you can see the root mean square error is the least using this method.

Let us now get the recommendations for user 33:

```
user_id = 33 #or #int(input("enter a user ID:"))
user_ratings = predictions_svd[user_id-1,:]
train_unkown_indices = np.where(train_matrix[user_id-1,:] == 0)[0]
user_recommendations = user_ratings[train_unkown_indices]
user_recommendations.shape
```

```
(1668,)
```

```
print('\nRecommendations for user {} are the movies: \n'.format(user_id))
for movie_id in user_recommendations.argsort()[-5:][: : -1]:
    print(movie_id +1)
```

```
Recommendations for user 33 are the movies:

257
321
736
325
319
```

2. Using content-based filtering:

for this method, we used the tmdb 5000 credits and tmdb 5000 movies database.

Importing all the libraries and reading the dataset and setting the column names.

```
import pandas as pd
import numpy as np
df1=pd.read_csv('tmdb_5000_credits.csv')
df2=pd.read_csv('tmdb_5000_movies.csv')
df1.columns = ['id','tittle','cast','crew']
df2= df2.merge(df1,on='id')
from sklearn.feature_extraction.text import TfidfVectorizer
```

sklearn.feature_extraction.text.TfidfVectorizer: Convert a collection of raw documents to a matrix of TF-IDF features. **TF-IDF** stands for "Term Frequency — Inverse Document Frequency". This is a technique to quantify a word in documents, we generally compute a weight to each word which signifies the importance of the word in the document and corpus. This method is a widely used technique in Information Retrieval, Text Mining and NLP in general. It is a statistical measure that, as we said, evaluates how relevant a word is to a document in a collection of documents. This is done by multiplying two metrics: how many times a word appears in a document, and the inverse document frequency of the word across a set of documents.

Define a TF-IDF Vectorizer Object. Remove all english stop words such as 'the', 'a' so that we can just have the meaningful words to work with.

```
tfidf = TfidfVectorizer(stop_words='english')
```

Replace NaN in the overview section of our read database with an empty string: (**nan** is a single object that always has the same id, no matter which variable you assign it to.)

```
df2['overview'] = df2['overview'].fillna('')
```

Constructing the required TF-IDF matrix by fitting and transforming the data:

```
tfidf_matrix = tfidf.fit_transform(df2['overview'])
```

Outputing the shape of tfidf_matrix:

```
from sklearn.metrics.pairwise import linear_kernel
```

sklearn.metrics.pairwise.linear_kernel: Compute the linear kernel between X and Y.

Computing the cosine similarity matrix:

```
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
indices = pd.Series(df2.index, index=df2['title']).drop_duplicates()
```

pd.series(): One-dimensional ndarray with axis labels (including time series)

Pandas **drop_duplicates()** method helps in removing duplicates from the data frame.

Function that takes in movie title as input and outputs most similar movies:

```
def get_recommendations(title, cosine_sim=cosine_sim):

    idx = indices[title]

    sim_scores = list(enumerate(cosine_sim[idx]))

    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)

    sim_scores = sim_scores[1:11]

    movie_indices = [i[0] for i in sim_scores]

    return df2['title'].iloc[movie_indices]
```

A **lambda** function can take any number of arguments, but can only have one expression. The **sorted()** function returns a sorted list of the specified iterable object. You can specify ascending or descending order. Strings are sorted alphabetically, and numbers are sorted numerically. method **list()** takes sequence types and converts them to lists. This is used to convert, let's say, a given tuple into list.

In the function made above, first we get the index of the movie that matches the title. Then we get the pairwsie similarity scores of all movies with that movie. Afterwards, we sort the movies based on the similarity scores. We get the scores of the 10 most similar movies, we get the movie indices and then the function returns the top 10 most similar movies.

Now we get a recommendation based on a movie title:

```
#get_movie = input("enter the name of a movie you enjoyed watching:")
get_recommendations('The Godfather'#or get_movie)
```

References:

https://www.news18.com/news/tech/netflix-managed-to-add-10-million-paid-subscribers-during-covid-19-lockdown-2720159.html

https://www.kaggle.com

https://www.github.com

https://towardsdatascience.com/

https://Wikipedia.com

https://www.geeksforgeeks.org

https://docs.scipy.org