Design And Analysis Of Algorithm

**Topic: Algorithmic Optimizations in E-Commerce**

**Submitted To: Mr Omer Ajmal**

**Submitted By: Adil Shakoor**

**Submitted On: 16-01-2024**

**Class: BS-CS-6-2M**

****

**The Islamia University of Bahawalpur**

# **Abstract:**

Algorithmic applications play a pivotal role in shaping the landscape of the e-commerce industry, influencing various aspects ranging from personalized user experiences to dynamic pricing strategies. This report delves into the realm of algorithmic applications in e-commerce, with a specific focus on recommendation systems. We explore the implementation of Matrix Factorization for recommendation systems, delve into the intricacies of various recommendation algorithms, and examine real-world applications that demonstrate the tangible impact of recommender systems in enhancing user engagement and driving business outcomes.

# Algorithmic applications in e-commerce

* recommendation systems
* dynamic pricing
* supply chain management

### Recommendation Systems Algorithms:

##### 1. Collaborative Filtering:

Description: Collaborative filtering algorithms recommend items based on the preferences and behaviors of similar users.

Example Algorithm: User-Based Collaborative Filtering, Item-Based Collaborative Filtering.

##### 2. Content-Based Filtering:

Description: Content-based filtering recommends items based on the characteristics and features of the items themselves, as well as the user's preferences.

Example Algorithm: TF-IDF (Term Frequency-Inverse Document Frequency), Cosine Similarity.

##### 3. Matrix Factorization:

Description: Matrix factorization algorithms decompose user-item interaction matrices into latent factors, uncovering hidden patterns.

Example Algorithm: Singular Value Decomposition (SVD), Alternating Least Squares (ALS).

##### 4. Deep Learning Models:

Description: Deep learning techniques, such as neural networks, are used to capture complex patterns in user behavior and item characteristics.

Example Algorithm: Neural Collaborative Filtering, Deep Autoencoders.

### Dynamic Pricing Algorithms:

##### 1. Rule-Based Pricing:

Description: Rule-based pricing involves setting prices based on predefined rules, such as competitor pricing, demand fluctuations, or time-based adjustments.

Example Algorithm: Time-of-Day Pricing, Competitor-Based Pricing.

##### 2. Reinforcement Learning:

Description: Reinforcement learning algorithms adjust prices dynamically based on continuous feedback from the market to maximize revenue or profit.

Example Algorithm: Q-Learning, Deep Reinforcement Learning for Pricing.

##### 3. Predictive Analytics:

Description: Predictive analytics models forecast demand and market conditions to optimize prices in real-time.

Example Algorithm: Time Series Analysis, Regression Models.

### Supply Chain Management Algorithms:

##### 1. Inventory Optimization:

Description: Inventory optimization algorithms aim to minimize costs and maximize service levels by determining optimal stock levels.

Example Algorithm: Economic Order Quantity (EOQ), Reorder Point (ROP).

##### 2. Routing Algorithms:

Description: Routing algorithms optimize the movement of goods through the supply chain, determining the most efficient transportation routes.

Example Algorithm: Dijkstra's Algorithm, Ant Colony Optimization.

##### 3. Demand Forecasting:

Description: Demand forecasting algorithms predict future demand based on historical data, enabling better inventory management.

Example Algorithm: Moving Averages, Exponential Smoothing.

# **Recommendation** **system**

A recommendation system is a tool that uses a series of algorithms, data analysis and artificial Intelligence (AI) to make recommendations online.

Recommendation Engines can be used to customize the experiences of your customer. You can quickly deploy this for any client since these are readily available in different languages such as R, Python, WEKA, etc. Companies like Netflix, YouTube, Amazon, etc use Recommendation Engines built using Python that readily interact with their microservices. Recommendation Engines also allow you to carry the experience of other customers and recommend products that people living in and around the customer are buying. The only caveat to Recommendation Engines is the highly researched ‘Cold Start’ problem. You need historic customer data to build a Recommendation Engine. Without that, a sophisticated Recommendation Engine cannot be built. There are a few workarounds and a lot of research going about to solve this problem.

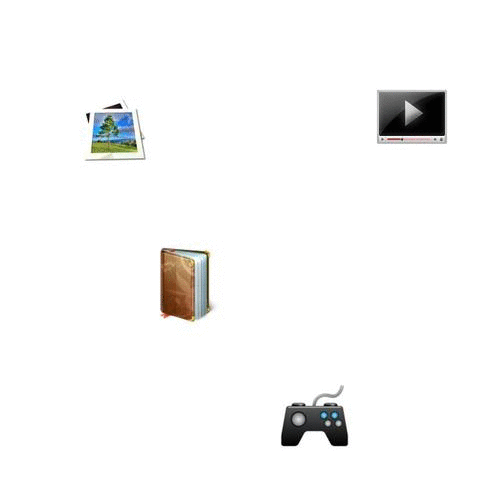
with the growing volume of online information, recommender systems have been an effective strategy to overcome information overload. The utility of recommender systems cannot be overstated, given their widespread adoption in many web applications, and their potential impact to ameliorate many problems related to over-choice. There are various types of Recommendation System Algorithms. They have been under development since the early 1990s. Since These systems play an important role in many e-commerce sites like Amazon, Uber-Eats, Netflix, and Youtube utilize these systems and they are also developing. In fact, even it is possible to see some competition to develop new algorithms for these companies. For example, In October 2006 Netflix released a dataset containing 100 million anonymous movie ratings and challenged the data mining, machine learning and computer science communities to develop systems that could beat the accuracy of its recommendation system, Cinematch. Recommender systems help users faced with an overwhelming selection of items by identifying particular items that are likely to match each user’s preferences or properties.

**There are fallowing common approaches to building a recommender system:**

## Collaborative Filtering

Collaborative filtering is a method of making predictions about the interests of a user by collecting preferences or taste information from many users. if you and I have liked many of the same movies, then I would probably like other movies that you like.

In figure 1, you see people rate different items. This is the key point of this approach. We always need a performance metric that’s called rating. This metric can either be exclusive for example after an item purchase the customer might rate the item or inclusive like viewing the page of the product. After that, the system makes predictions about the user’s rating for an item, that the user hasn’t rated yet considering similar items or users.



## Rating Matrix

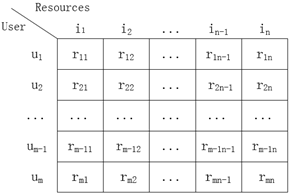
Applications of collaborative filtering typically involve very large data sets and these sets are sparse because we can’t expect users interacted with all the items that we have. Also, if they interacted with all of them, there would be nothing to recommend.

All collaborative filtering methods are ways to analyze this matrix from different perspectives.

*i = 1,2,3…N and N is the number of users*

*j = 1,2,3…M and M is the number of items*then

Let’s define the matrix *RNxM*. It is the user-item rating sparse matrix.



one specific row defines a user who rated items and one specific column defines an item that is rated by all users.

This matrix is the inspiration point of all collaborative methods. In this work, we are going to discuss Matrix Factorization specifically.

### Matrix Factorization Algorithm:

Matrix factorization is a dimensionality reduction technique that decomposes a matrix into the product of two or more lower-rank matrices. In the context of recommendation systems, matrix factorization is often applied to user-item interaction matrices to discover latent features representing user preferences and item characteristics. The goal is to approximate the original matrix by identifying these latent factors.

let's explore the Matrix Factorization Algorithm, focusing on the Singular Value Decomposition (SVD) approach as an example.

The primary goal of matrix factorization is to decompose a given matrix ( R ) into the product of two lower-rank matrices ( U ) and ( V ). This decomposition seeks to reveal latent factors that capture the underlying structure of the original matrix.

#### Problem Formulation:

- Given an (m\*n) matrix ( R ) representing user-item interactions (e.g., user ratings for movies), matrix factorization aims to find matrices ( U ) (an m\*k times matrix) and V an( k \*n times) matrix such that ( R approximatly equal to U \* V ).

#### Mathematical Representation:

- The matrix factorization can be represented as: (R approximtly equal to U \* V ).where ( U ) represents users and ( V ) represents items. The product (u\*v) approximates the original matrix ( R ).

#### Singular Value Decomposition (SVD:

- SVD is a key method for matrix factorization. It decomposes a matrix into three matrices, ( U ), Sigma , and ( V^T ), where ( Sigma ) is a diagonal matrix of singular values.

-Mathematically, for a matrix ( R ), SVD is represented as: ( R = U\* V^T \* sigma).

#### Algorithm Steps:

a. Initialization:

- Initialize matrices ( U ) and ( V ) with random values or predefined initializations.

b. Singular Value Decomposition (SVD):

- Use SVD to decompose the matrix ( R ) into ( U ), ( sigma ), and ( V^T ).

c. Rank Reduction:

- Choose a desired rank ( k ) and retain the top ( k ) singular values and corresponding columns of ( U ) and ( V^T ).

d. Approximation:

- Reconstruct the approximate matrix ( R ) using the reduced matrices ( U\_k ), ( Sigma\_k ), and ( V\_k^T ).

e. Optimization:

- Fine-tune the values in ( U\_k ) and ( V\_k^T ) using optimization techniques (e.g., gradient descent) to minimize the error between the original and reconstructed matrices.

#### Complexity Analysis:

- The time complexity of the matrix factorization algorithm depends on the dimensions of the matrices involved. SVD itself has a time complexity of approximately O(m \* n^2) , and the optimization step may add additional complexity.

#### Applications:

- Matrix factorization is widely used in recommendation systems, collaborative filtering, and data compression. It uncovers latent patterns in user-item interactions, enabling personalized recommendations.

#### Performance Metrics:

- Evaluate the performance of the algorithm using metrics such as Mean Squared Error (MSE) or Root Mean Squared Error (RMSE), comparing predicted and actual values in the user-item interaction matrix.

# **Recommender** **System Algorithmic Explanation**

## **Matrix Factorization:**

The algorithm of matrix factorization of the recommender system:

(1) Introduction to Matrix Factorization

(2) Mathematic concept of matrix factorization

(3) Hands-on experience of python code on matrix factorization

### Introduction to Matrix Factorization

Matrix factorization is a way to generate latent features when multiplying two different kinds of entities. Collaborative filtering is the application of matrix factorization to identify the relationship between items and users’ entities. With the input of users’ ratings on the shop items, we would like to predict how the users would rate the items so the users can get the recommendation based on the prediction.

Assume we have the customers’ ranking table of 5 users and 5 movies, and the ratings are integers ranging from 1 to 5, the matrix is provided by the table below.

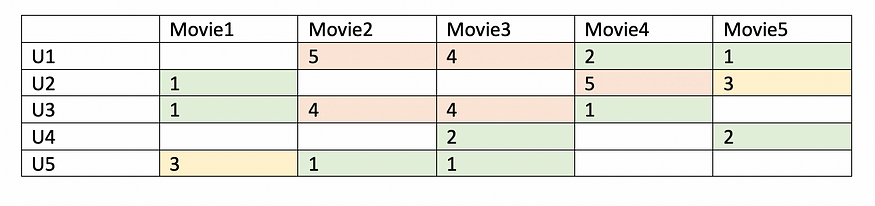


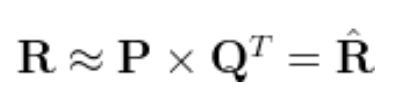
Table1-Users’ ratings table on movie

Since not every user gives ratings to all the movies, there are many missing values in the matrix and it results in a sparse matrix. Hence, the null values not given by the users would be filled with 0 such that the filled values are provided for the multiplication. For example, two users give high ratings to a certain move when the movie is acted by their favorite actor and actress or the movie genre is an action one, etc. From the table above, we can find that the user1 and user3 both give high ratings to move2 and movie3. Hence, from the matrix factorization, we are able to discover these latent features to give a prediction on a rating with respect to the similarity in user’s preferences and interactions.

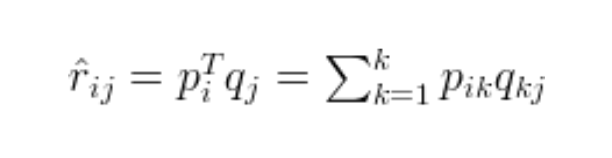
Given a scenario, user 4 didn’t give a rating to the movie 4. We’d like to know if user 4 would like movie 4. The method is to discover other users with similar preferences of user 4 by taking the ratings given by users of similar preferences to the movie 4 and predict whether the user 4 would like the movie 4 or not.

### Mathematic concept of matrix factorization

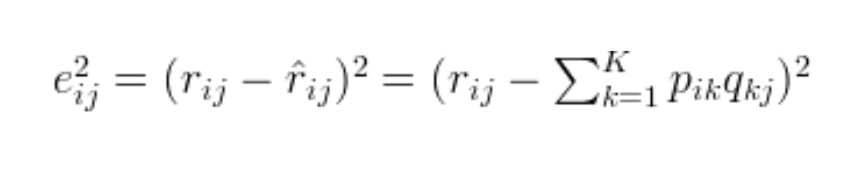
Define a set of Users (U), items (D), R size of |U|, and |D|. The matrix |U|\*|D| includes all the ratings given by users. The goal is to discover**K**latent features. Given with the input of two matrics matrices P (|U|\*k) and Q (|D|\*k), it would generate the product result R.



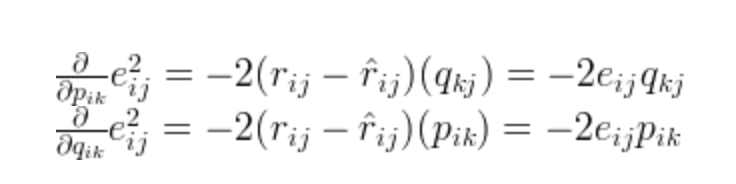
Matrix P represents the association between a user and the features while matrix Q represents the association between an item and the features. We can get the prediction of a rating of an item by the calculation of the dot product of the two vectors corresponding to u\_i and d\_j.



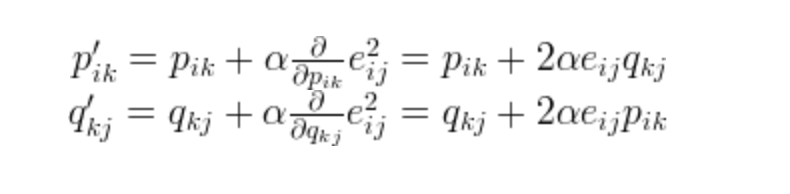
To get two entities of both P and Q, we need to initialize the two matrices and calculate the difference of the product named as matrix M. Next, we minimize the difference through the iterations. The method is called **gradient descent**, aiming at finding a local minimum of the difference.



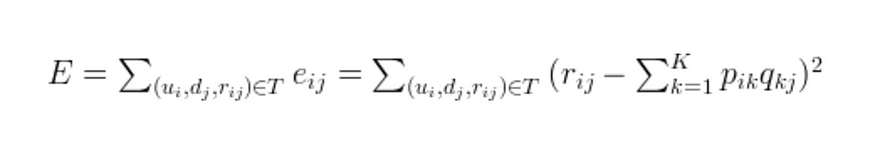
To minimize the error, the gradient is able to minimize the error, and therefore we differentiate the above equation with respect to these two variables separately.



From the gradient, the mathematic formula can be updated for both p\_ik and q\_kj. a is the step to reach the minimum while the gradient is calculated, and a is usually set with a small value.



From the above equation, p’\_ik and q’\_kj can both be updated through iterations until the error converges to its minimum.



**Example of matrix factorization**

The dot product of user and item matrix can generate the rating matrix, while the user matrix is the shape of k (users) \* f (features) and the item matrix is the shape of j(items) \* f (features). From user’s and item’s matrices, features of the movies can be its genre, actors, plot, etc. Having two features of the factored matrices, let’s assume F1 to be “If this movie is a comedy or not?” and F2 to be “if Robin Williams acts in the movie?”

**User Matrix:**According to User1, if its a comedy movie, he’ll give it 4 points and if Robin Williams is the actor in the movie, he’ll give it 3 more points.

**Item Matrix:**There are mainly binary values in the item matrix where the value is 1 when it meets the conditions of features above and 0 otherwise. By performing the dot product of the user matrix and item matrix,*the rating matrix would be generated.*

*The matrix factorization of user and item matrices can be generated when the math cost function****RMSE****is minimized through matrix factorization. Following the above mathematic concept, gradient descent is one of the methods to minimize RMSE through each iteration.*

### Practical Python Code for Matrix Factorization

Below is the python code snippet to conduct the gradient descent algorithm. We set a rating matrix with 4 movies given by 6 users. As you can see, some users didn’t watch some movies before, so the rating is given as 0 in the rating.

|  |  |
| --- | --- |
| import numpy | |
|  | |  | |
|  | | def matrix\_factorization(R, P, Q, K, steps=5000, alpha=0.0002, beta=0.02): | |
|  | | ''' | |
|  | | R: rating matrix | |
|  | | P: |U| \* K (User features matrix) | |
|  | | Q: |D| \* K (Item features matrix) | |
|  | | K: latent features | |
|  | | steps: iterations | |
|  | | alpha: learning rate | |
|  | | beta: regularization parameter''' | |
|  | | 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: | |
|  | | # calculate error | |
|  | | eij = R[i][j] - numpy.dot(P[i,:],Q[:,j]) | |
|  | |  | |
|  | | for k in range(K): | |
|  | | # calculate gradient with a and beta parameter | |
|  | | 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)) | |
|  | | # 0.001: local minimum | |
|  | | if e < 0.001: | |
|  | |  | |
|  | | break | |
|  | |  | |
|  | | return P, Q.T | |
|  |
| R = [ |
|  | [4,0,0,1], | |
|  |  | |
|  | [1,1,0,5], | |
|  |  | |
|  | [1,0,0,4], | |
|  |  | |
|  | [0,1,5,4], | |
|  |  | |
|  | [2,1,3,0], | |
|  |  | |
|  | ] | |
|  |  | |
|  | R = numpy.array(R) | |
|  | # N: num of User | |
|  | N = len(R) | |
|  | # M: num of Movie | |
|  | M = len(R[0]) | |
|  | # Num of Features | |
|  | K = 3 | |
|  |  | |

The predicted matrix is generated below. As you can see, the predicted matrix has similar output with the true values, and the 0 ratings are replaced with the prediction based on the similar users’ preferences on movies.

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. With the feature given as three, the algorithm is able to associate the users and items to three different features, and the predictions also follow these associations. We can find that U2, U5, and U6 give a low rating on M1 and M2, and give a high rating on M3. Even though U5 didn’t watch M1, we would further infer that U5 may not like M1.

In the real-world, the rating matrix is very sparse since every user watches movies at different frequencies. However, the error function RMSE is only calculated with the non-null rating. The missing entries in the rating matrix would be replaced by the dot product of the factor matrices. Therefore, we know what to recommend to the users with the unseen movies based on the prediction.

### In Conclusion:

* Matrix factorization is a collaborative filtering method to find the relationship between items’ and users’ entities. Latent features, the association between users and movies matrices, are determined to find similarity and make a prediction based on both item and user entities
* The matrix factorization of user and item matrices can be generated when the math cost function **RMSE** is minimized through matrix factorization. Gradient descent is a method to minimize the cost function.

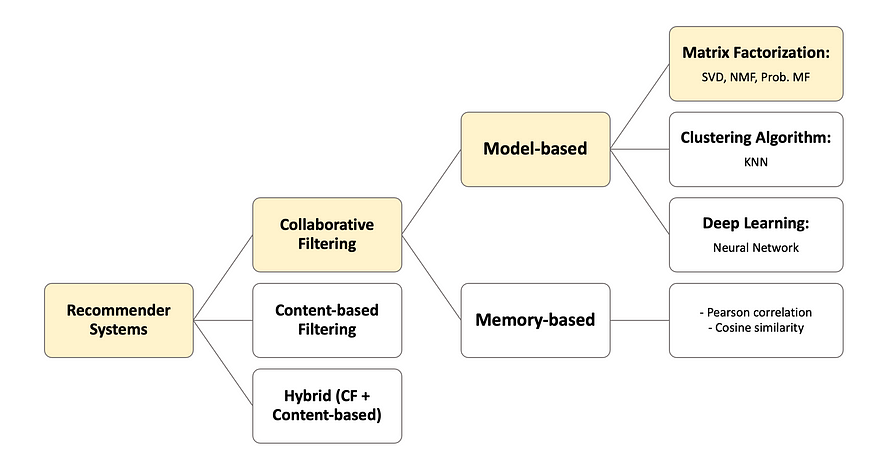
# **Implementation of Recommendation System Using Matrix Factorization**

## Collaborative Filtering with Matrix Factorization:

Netflix is ​​a popular online streaming platform that offers its subscribers a wide range of movies, documentaries, and TV shows. To improve users’ experience, Netflix has developed a sophisticated recommendation system that suggests movies based on your past viewing history, ratings, and preferences.

The recommender system uses complex algorithms that analyze vast amounts of data to predict what users will most likely enjoy. With over 200 million subscribers worldwide, Netflix’s recommendation system is a key factor in its success and sets the standard for the streaming industry.

### More on CollaborativeFiltering

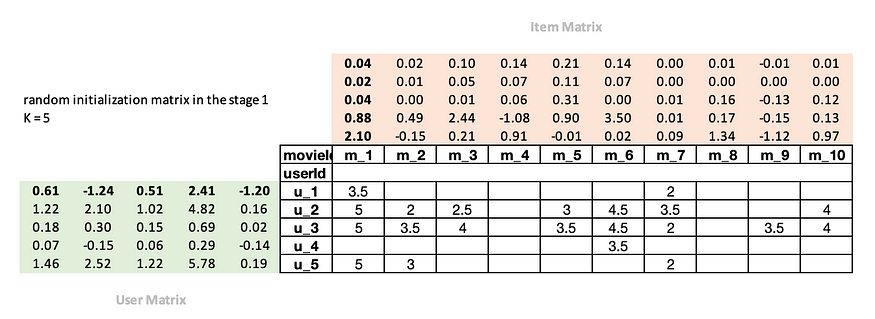
There is a wide scope of Recommender System model types as shown in the figure below, but today this article will focus on collaborative filtering (CF) with Matrix Factorization

**Type of Recommender System** -Image

**Collaborative Filtering with Matrix Factorization**

Matrix Factorization is a mathematical process that transforms a complicated matrix into a lower-dimensional space. One of the most popular matrix factorization techniques used in recommender systems is Singular Value Decomposition (SVD), Non-negative Matrix Factorization (NMF), and Probabilistic Matrix Factorization

Following is the illustration of how the matrix factorization concept is capable of predicting the user-movie rating



**Stage 1:** Matrix Factorization will randomly initialize the number, and the number of factors (K) is set. In this sample, we will set K = 5

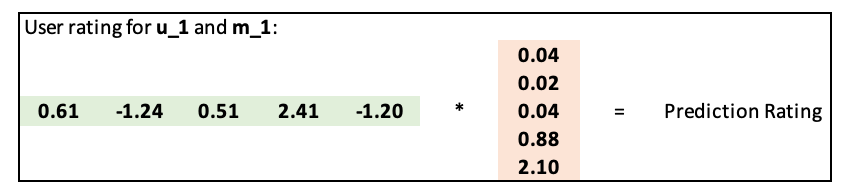
* User Matrix (green box) represents the association between each user and the features
* Item Matrix (orange box) represents the association between each item and the features

Here, for instance, we are creating 5 features (k=5) to represent the character of m\_1 movie: comedy as 2.10, horror as 0.88, action as 0.04, parent-guide as 0.02, and family-friendly as 0.04. And the reverse is true for user\_matrix. User\_matrix represents the character of user such as prefered actors or directors, favorite movie production and many more.

**Stage 2:** Rating Prediction is calculated from the dot product of *User Matrix* and *Item Matrix*



where R as true rating, P as User Matrix, Q as Item Matrix, resulted R’ as predicted rating.



In better mathematical notation, the **predicted rating R’** can be represented in the equation as follows:



**Stage 3:** The squared error is used to calculate the difference between true rating and prediction rating



Once we have these steps in place, we can optimize our parameters, using stochastic gradient descent. It will then compute the derivative of this value





At each iteration, the optimizer will compute the match between each movie and each user by multiplying them using the dot product, then compare it to the actual rating that the user gave the movie. It will then compute the derivative of this value and update the weights by multiplying it by the learning rate ⍺. As we repeat this process many times, the loss will improve, leading to better recommendations.





One of matrix factorization models that have been widely used in recommendation systems is known as [Singular Value Decomposition](https://en.wikipedia.org/wiki/Singular_value_decomposition) (SVD). SVD itself has broad applications, including image compression, and noise reduction in signal processing. Additionally, SVD is commonly employed in recommender systems, where it is adept at addressing the sparsity issue inherent in large user-item matrices.

This article will also provide **an overview** of SVD implementation using the Surprise Package.

So let’s get our hands dirty with **the implementation**!!

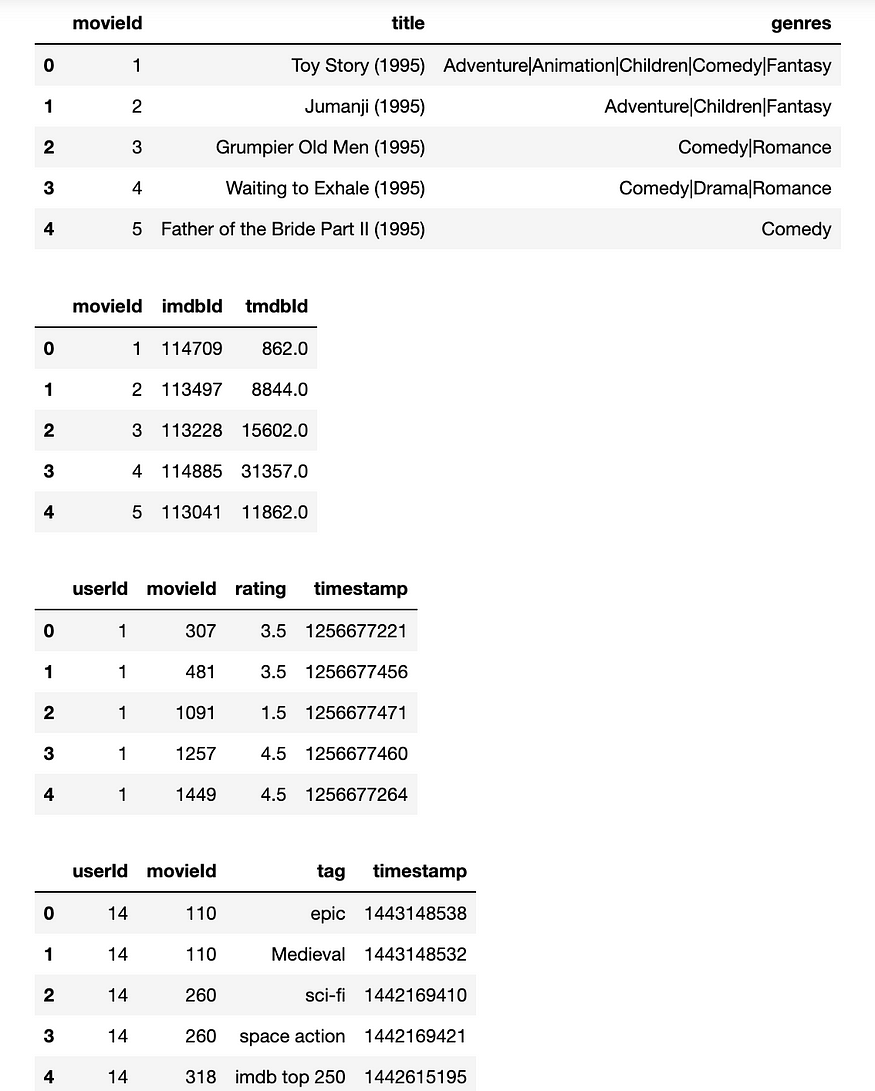
## Implementation Contents

* Data Import
* Data Pre-Processing
* Implementation #1: Matrix Factorization in Python from Scratch
* Implementation #2: Matrix Factorization with Surprise Package

### Data Import

Since we are developing a recommendation system like Netflix, but we may not have access to their big data, we are going to use a great dataset from [MovieLens](https://grouplens.org/datasets/movielens/" \t "_blank) for this practice  *with permission*. Besides, you can read and review their [README](https://files.grouplens.org/datasets/movielens/ml-latest-small-README.html) files for the usage licenses and other details. This dataset comprises millions of movies, users, and users’ past-interacting ranking.

After extracting the zip file, there will be 4 csv given as follows:



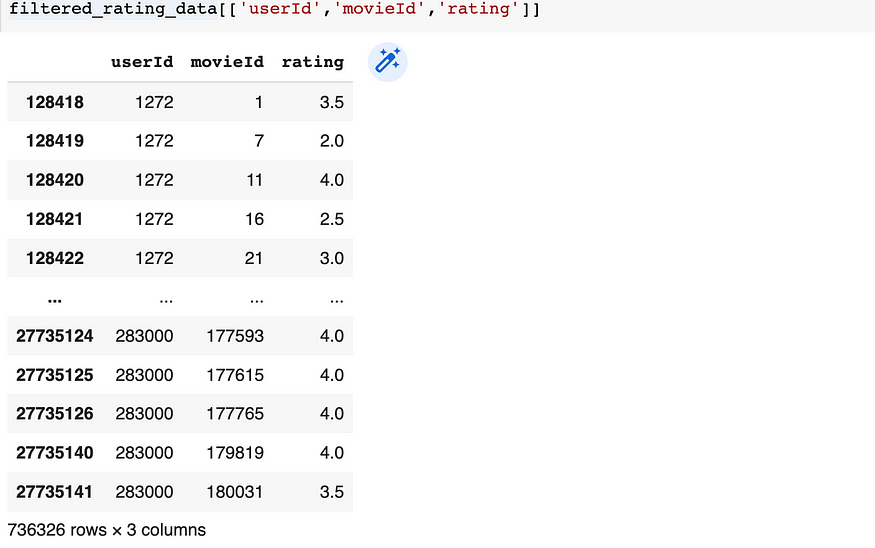
### Data Pre-Processing

Btw, Collaborative Filtering has a problem with *user cold-start.* The cold-start problem refers to a situation in which a system or algorithm could not make accurate predictions or recommendations for new users, items, or entities that has no prior information. This can happen when there is little or no historical data available for the new users or items, making it difficult for the system to understand their preferences or characteristics.

The cold-start problem is a common challenge in recommendation systems, where the system needs to provide personalized recommendations for users with limited or no interaction history.

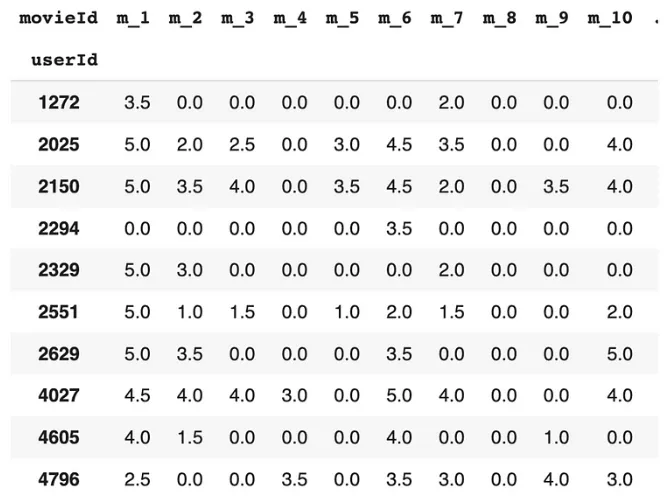
In this stage, we are going to select users who have at least interacted with 2000 movies and movies who have been rated by 1000 users.

|  |
| --- |
| n\_interacted = 2000 |
|  | user\_movie\_data\_temp = pd.pivot\_table(rating\_data, index = ['userId'], values='movieId', aggfunc='count') |
|  | user\_movie\_data\_temp[user\_movie\_data\_temp.movieId>=n\_interacted] |
|  | selected\_user\_ids = user\_movie\_data\_temp[user\_movie\_data\_temp.movieId>=n\_interacted].index |
|  | print('number of userIds: ', str(len(selected\_user\_ids))) |
|  |  |
|  | n\_rated = 1000 |
|  | get\_rated\_movie = pd.pivot\_table(rating\_data, index=['movieId'], values='userId', aggfunc='count') |
|  | get\_rated\_movie[get\_rated\_movie.userId>=n\_rated] |
|  | selected\_movie\_ids = get\_rated\_movie[get\_rated\_movie.userId>=n\_rated].index |
|  |  |
|  | print('numbser of movieIds: ', str(len(selected\_movie\_ids))) |
|  |  |
|  | filtered\_rating\_data = rating\_data[(rating\_data['userId'].isin(selected\_user\_ids)) &(rating\_data['movieId'].isin(selected\_movie\_ids))] |
|  | filtered\_rating\_data['movieId'] = filtered\_rating\_data['movieId'].apply(lambda x: 'm\_'+str(x)) |
|  | # filtered\_rating\_data['user\_movie'] = filtered\_rating\_data['userId'].astype(str) + '\_' + filtered\_rating\_data['movieId'].astype(str) |
|  |  |
|  | print('raw data shape. : ',str(filtered\_rating\_data.shape)) |



**Data output after data pre-processing**

While the information we require is present, it is not presented in a way that is beneficial for humans to comprehend. However, I have created a table that presents the same data in a format that is easier for humans to understand.



**Raw data** -Image

## Implementation #1: Matrix Factorization in Python from Scratch

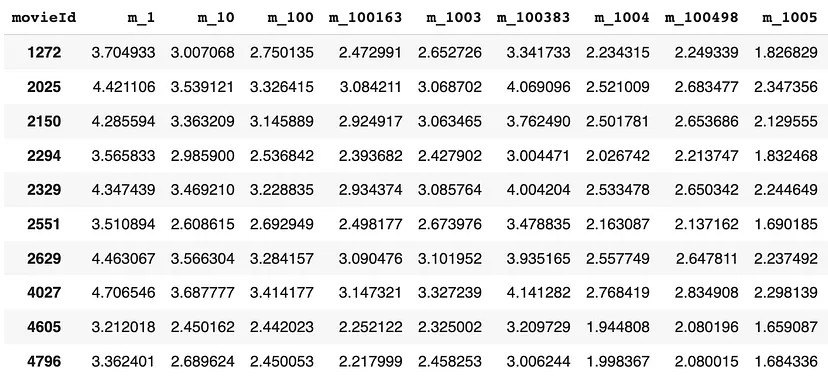
Here is the Python snippet for implementing Matrix Factorization with the gradient descent. The matrix\_factorization function returns 2 matrices: **nP (user matrix)** and **nQ (item matrix)**.

|  |
| --- |
| def matrix\_factorization(R, K, steps=5, alpha=0.002, beta=0.02): |
|  | ''' |
|  | R: rating matrix |
|  | P: |U| \* K (User features matrix) |
|  | Q: |D| \* K (Item features matrix) |
|  | K: latent features |
|  | steps: iterations |
|  | alpha: learning rate |
|  | beta: regularization parameter |
|  |  |
|  | ''' |
|  |  |
|  | P = np.random.rand(len(R),K) |
|  | Q = np.random.rand(len(R[0]),K) |
|  | Q = Q.T |
|  |  |
|  | for step in range(steps): |
|  | print('Processing epoch {}'.format(step)) |
|  |  |
|  | for i in range(len(R)): |
|  | for j in range(len(R[i])): |
|  | if R[i][j] > 0: |
|  | eij = R[i][j] - np.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 = np.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] - np.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)) |
|  | # 0.001: local minimum |
|  | if e < 0.001: |
|  |  |
|  | break |
|  |  |
|  | return P, Q.T |

Then, fit the training dataset to the model and here I set n\_factor K = 5. Following that, predictions can be computed by **multiplying nP and the transpose of nQ** using the dot product method, as illustrated in the code snippet below.

|  |
| --- |
| R=np.array(user\_movie\_data\_train) |
|  | nP, nQ = matrix\_factorization(R, K=10) |
|  |  |
|  |  |
|  | #------------------- |
|  | pred\_R = np.dot(nP, nQ.T) |
|  |  |
|  | # Transforming prediction to reconstructed matrix back into a Pandas dataframe in cross-tabural format |
|  | user\_movie\_pred = pd.DataFrame(pred\_R, columns=user\_movie\_data\_train.columns, index=list(user\_movie\_data\_train.index)) |
|  | print(user\_movie\_pred.shape) |
|  | user\_movie\_pred.head(10) |

As a result, here is the final prediction that the matrix\_factorization produce

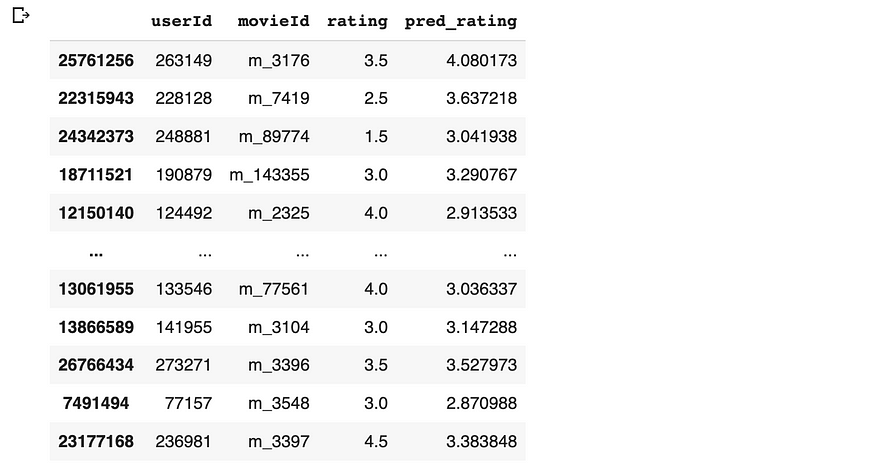


**New predicted rating in train set**-Image

**Prediction on the Test Set**

The following snippet leverages the given **nP (user matrix)** and **nQ (movie matrix)** to make a prediction on the test set

|  |
| --- |
| This matrix can be used independently to predict testing dataset |
|  | we will tranform it so each array row can be index to each userId or movieId |
|  | ''' |
|  |  |
|  | # User Matrix |
|  | Pu = pd.DataFrame(nP, index=list(user\_movie\_data\_train.index)) |
|  | # Movie Matrix |
|  | Qu = pd.DataFrame(nQ, index=user\_movie\_data\_train.columns) |
|  |  |
|  | # Create a simple function to predict test set |
|  | def predict\_rating(data): |
|  | try: |
|  | pred\_rating = np.dot(Pu.loc[data.userId], Qu.loc[data.movieId].T) |
|  | except Exception as e: |
|  | pred\_rating = np.nan |
|  | print('Unknown user: {} or movieId: {}'.format(data.userId,data.movieId)) |
|  |  |
|  | return pred\_rating |
|  |  |
|  | # Calculate predicted rating |
|  | test\_df['pred\_rating'] = test\_df.apply(predict\_rating, axis=1) |
|  | test\_df |



**Evaluating The Prediction Performance**

Although there are various evaluation metrics for Recommender Systems, such as Precision@K, Recall@K, MAP@K, and the list goes on. For this exercise, I will employ a basic accuracy metric namely RMSE. I probably will write other evaluation metrics in greater detail in the subsequent article.

|  |
| --- |
| rmse\_test = mean\_squared\_error(test\_df['rating'], test\_df['pred\_rating'], squared=False) |
|  | rmse\_test |

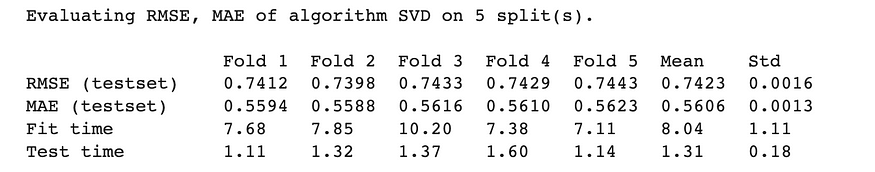
As the result, the RMSE on the test set is **0.829**, which is pretty decent even before the hyper-tuning is implemented. Definitely, we can tune several parameters like learning rate, n\_factor, epochs steps for better outcomes.

## Implementation #2: Matrix Factorization with Surprise Package

In this segment, we opted for the Python library namely the **surprise package.** A [surprise package](https://surprise.readthedocs.io/en/latest/getting_started.html) is a Python library for building and evaluating recommendation systems. It provides a simple and easy-to-use interface for loading and processing datasets, as well as implementing and evaluating different recommendation algorithms.

**Data Import and Model Training**

|  |
| --- |
| reader = Reader(rating\_scale=(0.5,5)) |
|  | data = Dataset.load\_from\_df(filtered\_rating\_data[['userId','movieId','rating']], reader) |
|  |  |
|  | trainset, testset = train\_test\_split(data, test\_size=0.25) |
|  |  |
|  | # We'll use the famous SVD (one of matrix factorization) algorithm. |
|  | algo = SVD() |
|  |  |
|  | # Train the algorithm on the trainset, and predict ratings for the testset |
|  | algo.fit(trainset) |
|  | predictions = algo.test(testset) |
|  |  |
|  | # Then compute RMSE |
|  | evaluation = cross\_validate(algo, data, measures=['RMSE','MAE'], cv= 5, verbose=True) |

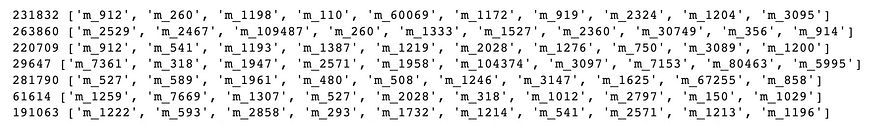


**Top-N recommendation generator**

|  |
| --- |
| def get\_top\_n(predictions, n=5): |
|  | # First map the predictions to each user. |
|  | top\_n = defaultdict(list) |
|  | for uid, iid, true\_r, est, \_ in predictions: |
|  | top\_n[uid].append((iid, est)) |
|  |  |
|  | # Then sort the predictions for each user and retrieve the k highest ones. |
|  | for uid, user\_ratings in top\_n.items(): |
|  | user\_ratings.sort(key=lambda x: x[1], reverse=True) |
|  | top\_n[uid] = user\_ratings[:n] |
|  |  |
|  | return top\_n |
|  |  |
|  | #get the top 10 recommended movie |
|  | top\_n = get\_top\_n(predictions, 10) |
|  |  |
|  | for uid, user\_ratings in top\_n.items(): |
|  | print(uid, [mid for (mid, \_) in user\_ratings]) |

for UserId: 231832 following is the top 10 movie recommendation list:

*m\_912, m\_260, m\_1198, m\_110, m\_60069, m\_1172, m\_919, m\_2324, m\_1204, m\_3095*

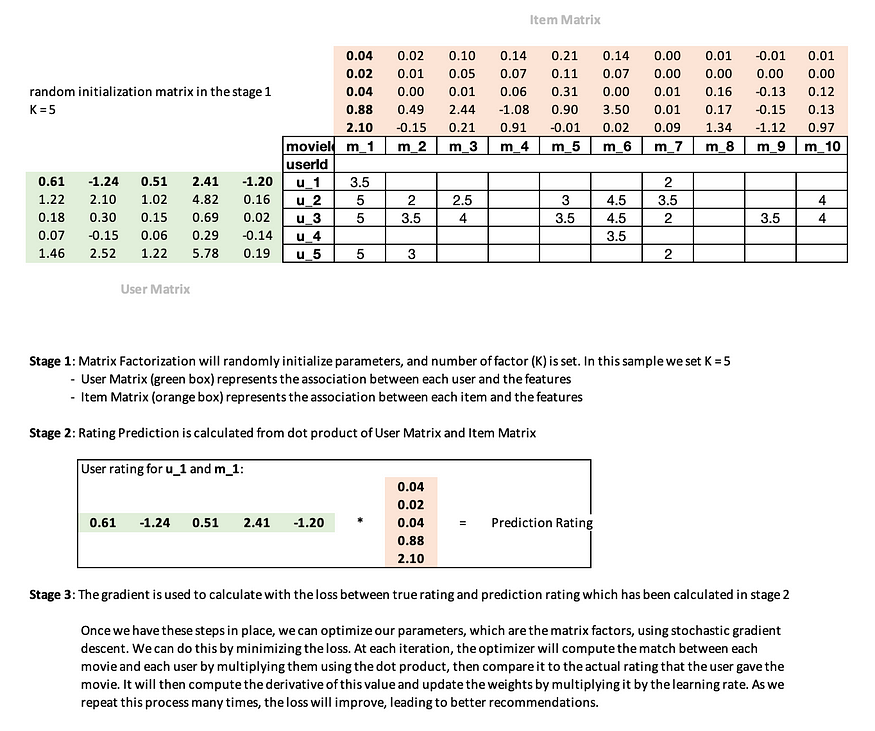


**Top 10 recommendation output**-Image

**Summary**

The utilization of Matrix Factorization in modern entertainment like Netflix helps to understand user preferences. This information is then used to recommend the most relevant item/product/movie to the end user.

Here is a summary of the Matrix Factorization , in case I need to explain it to my grandkids one day….



**Reference**

*Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) .*<https://doi.org/10.1145/2827872>

# Analysis of Matrix Factorization Algorithms in Industry:

## Singular Value Decomposition (SVD):

#### Strengths:

- Latent Factor Discovery: SVD effectively uncovers latent factors within data, allowing industries to discover hidden patterns and relationships.

- Global and Local Patterns: It captures both global patterns in the entire dataset and local patterns within specific subsets.

#### Limitations:

- Scalability: SVD may face scalability challenges with large datasets due to its computational complexity.

- Cold Start Problem: SVD may struggle with new items or users not present in the training data, leading to the "cold start" problem.

#### Suitability:

- Well-suited for collaborative filtering in recommendation systems where uncovering latent factors is crucial.

## Alternating Least Squares (ALS):

#### Strengths:

- Parallelization: ALS can be efficiently parallelized, making it suitable for distributed computing environments.

- Implicit Feedback Handling: It handles implicit feedback well, allowing industries to leverage implicit user interactions.

#### Limitations:

- Hyperparameter Tuning: ALS requires tuning of hyperparameters, and the choice of regularization terms can impact its performance.

- Convergence Sensitivity: Convergence can be sensitive to initialization and data sparsity.

#### Suitability:

- Particularly suitable for collaborative filtering with implicit feedback data.

## Stochastic Gradient Descent (SGD):

#### Strengths:

- Scalability: SGD is scalable and can handle large datasets efficiently.

- Real-Time Updates: It supports real-time updates, making it suitable for scenarios with frequently changing data.

#### Limitations:

- Sensitivity to Learning Rates: Performance can be sensitive to the choice of learning rates, requiring careful tuning.

- Convergence Speed: Convergence speed may vary depending on the dataset and initialization.

#### Suitability:

- Well-suited for scenarios where scalability and real-time updates are critical, such as dynamic recommendation systems.

## Evaluation of Impact on Industry Metrics:

Efficiency:

- Positive Impact: Matrix factorization algorithms contribute to efficiency by uncovering relevant patterns, leading to more accurate recommendations and streamlined decision-making.

- Challenge: The computational complexity of some algorithms, such as SVD, may impact real-time efficiency, necessitating optimized implementations or alternative approaches like ALS or SGD.

### Productivity:

- Positive Impact: Improved recommendations enhance user satisfaction and engagement, directly contributing to increased productivity and user loyalty.

- Challenge: The "cold start" problem, especially with SVD, can affect productivity when dealing with new items or users.

### Cost Savings:

- Positive Impact: Enhanced efficiency and productivity can lead to cost savings through optimized resource allocation and improved user retention.

- Challenge: Initial setup costs, including algorithm implementation and tuning, may be incurred.

### Other Relevant Metrics:

- Personalization: Matrix factorization algorithms excel in providing personalized recommendations, contributing significantly to user experience.

- Adaptability: Algorithms like ALS and SGD, with their parallelization and real-time capabilities, enable industries to adapt quickly to changing user preferences.

# **Real World Applications**

Matrix factorization is a popular technique in recommender systems that involves decomposing a user-item interaction matrix into latent factors. Here are some real-world applications where recommender systems using matrix factorization have been successfully employed:

## 1. Movie and TV Show Recommendations:

Platforms like Netflix and Hulu use matrix factorization to recommend movies and TV shows to users based on their viewing history and preferences. The system learns latent factors for users and items, improving the accuracy of recommendations.

## 2. Music Recommendations:

Music streaming services like Spotify and Pandora utilize matrix factorization to suggest songs and playlists to users. The system analyzes user listening history, preferences, and behaviors to provide personalized music recommendations.

## 3. E-commerce Product Recommendations:

Online retailers such as Amazon and eBay employ matrix factorization to recommend products to users based on their browsing history, purchase behavior, and preferences. This helps improve user engagement and sales.

## 4. News Article Recommendations:

News websites and aggregators use matrix factorization to personalize news article recommendations for users. The system considers a user's reading habits and interests to suggest relevant articles.

# **Conclusion**

In conclusion, algorithmic applications, particularly recommendation systems, are indispensable tools for e-commerce platforms seeking to optimize user experiences and drive business success. Matrix Factorization, with its ability to uncover latent features, stands out as a key technique within this domain. As e-commerce continues to evolve, the strategic implementation of recommendation algorithms proves to be a transformative force, shaping the way users discover products and interact with online platforms. The real-world applications presented underscore the practical significance of these algorithms, emphasizing their role in enhancing customer satisfaction, boosting sales, and ensuring a competitive edge in the dynamic e-commerce landscape.