Motivation

The reason that there should be data science on this project is because I have wondered how YouTube recommends content, or how Facebook recommends new friends? This particular project is going to be recommending different movies. All of these recommendations are made possible by the implementation of recommender systems.

Recommender systems encompass a class of techniques and algorithms that can suggest “relevant” items to users. They predict future behavior based on past data through a multitude of techniques including matrix factorization.

In this article, I’ll look at why we need recommender systems and the different types of users online. Then, I’ll show you how to build your own movie recommendation system using an open-source dataset.

Why Do We Need Recommender Systems?

For any given product, there are sometimes thousands of options to choose from. Think of the examples above: streaming videos, social networking, online shopping; the list goes on. Recommender systems help to personalize a platform and help the user find something they like.

The easiest and simplest way to do this is to recommend the most popular items. However, to really enhance the user experience through personalized recommendations, we need dedicated recommender systems.

Now that we understand the importance of recommender systems, let’s have a look at types of recommendation systems.

Understanding

Types of Recommender Systems

Machine learning algorithms in recommender systems typically fit into two categories: content-based and also collaborative filtering systems, but modern systems combine both approaches.

A) Content-Based Movie Recommendation Systems

Content-based methods are based on the similarity of movie attributes. Using this type of recommender system, if a user watches one movie, similar movies are recommended. For example, if a user watches a comedy movie starring Adam Sandler, the system will recommend them movies in the same genre or starring the same actor, or both. With this in mind, the input for building a content-based recommender system is movie attributes.

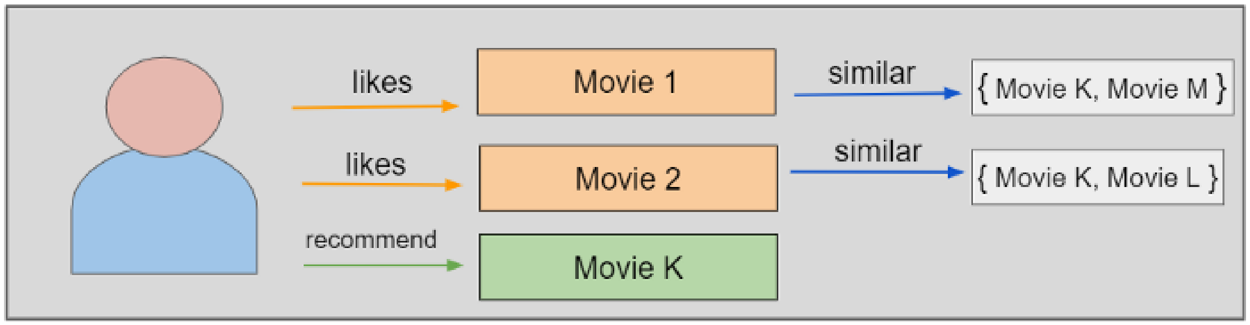
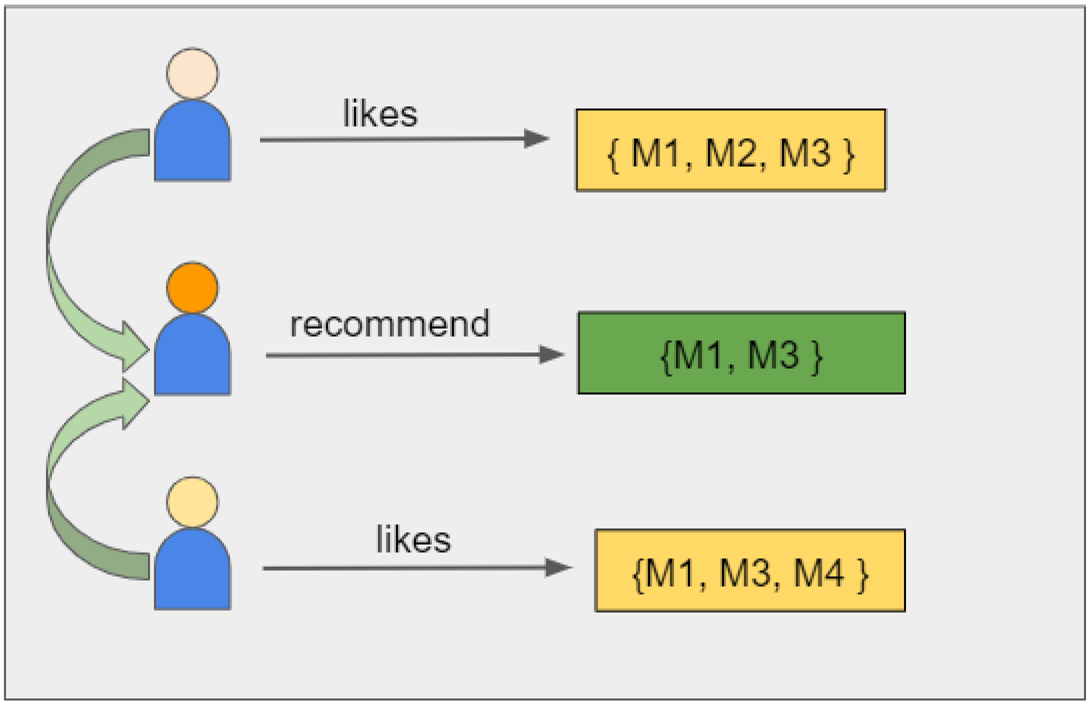


Figure 1: Overview of content-based recommendation system (Image created by author)

B) Collaborative Filtering Movie Recommendation Systems

With collaborative filtering, the system is based on past interactions between users and movies. With this in mind, the input for a collaborative filtering system is made up of past data of user interactions with the movies they watch.

For example, if user A watches M1, M2, and M3, and user B watches M1, M3, M4, we recommend M1 and M3 to a similar user C. You can see how this looks in the figure below for clearer reference.



This data is stored in a matrix called the user-movie interactions matrix, where the rows are the users and the columns are the movies.

Now, let's implement our own movie recommendation system using the concepts discussed above.

Pipeline

Data Collection

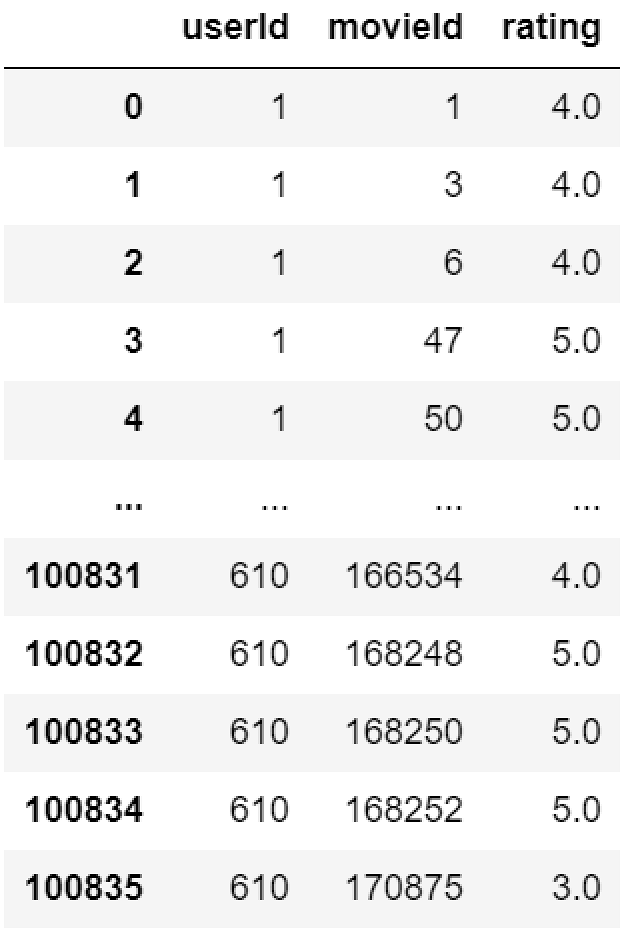
The Dataset

For our own system, we'll use the open-source MovieLens dataset from GroupLens. This dataset contains 100K data points of various movies and users.

We will use three columns from the data:

- userId
- movieId
- rating

You can see a snapshot of the data in figure 3, below:



	userId	movieId	rating
0	1	1	4.0
1	1	3	4.0
2	1	6	4.0
3	1	47	5.0
4	1	50	5.0
...	...	...	...
100831	610	166534	4.0
100832	610	168248	5.0
100833	610	168250	5.0
100834	610	168252	5.0
100835	610	170875	3.0

Data Management/Representation

Designing our Movie Recommendation System

To obtain recommendations for our users, we will predict their ratings for movies they haven't watched yet. Movies are then indexed and suggested to users based on these predicted ratings.

To do this, I will use past records of movies and user ratings to predict their future ratings. At this point, it's worth mentioning that in the real world, we will likely encounter new users or movies without a history.

Implementation

For my recommender system, I will use both of the techniques mentioned above: content-based and collaborative filtering. To find the similarity between movies for my content based method, I'll use a cosine similarity function. For my collaborative filtering method, I'll use a matrix factorization technique.

The first step is creating a matrix factorization based model. I'll use the output of this model and a few handcrafted features to provide inputs to the final model. The basic process will look like this:

- Step 1: Build a matrix factorization-based model
- Step 2: Create handcrafted features
- Step 3: Implement the final model

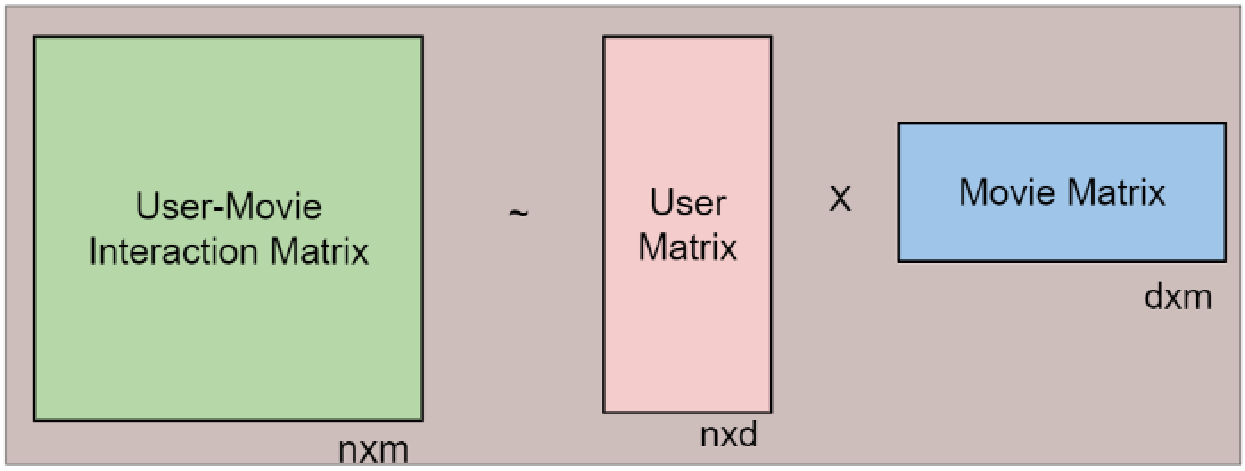
We'll look at these steps in greater detail below.

Step 1: Matrix Factorization-based Algorithm

Matrix factorization is a class of collaborative filtering algorithms used in recommender systems. This family of methods became widely known during the Netflix prize challenge due to how effective it was.

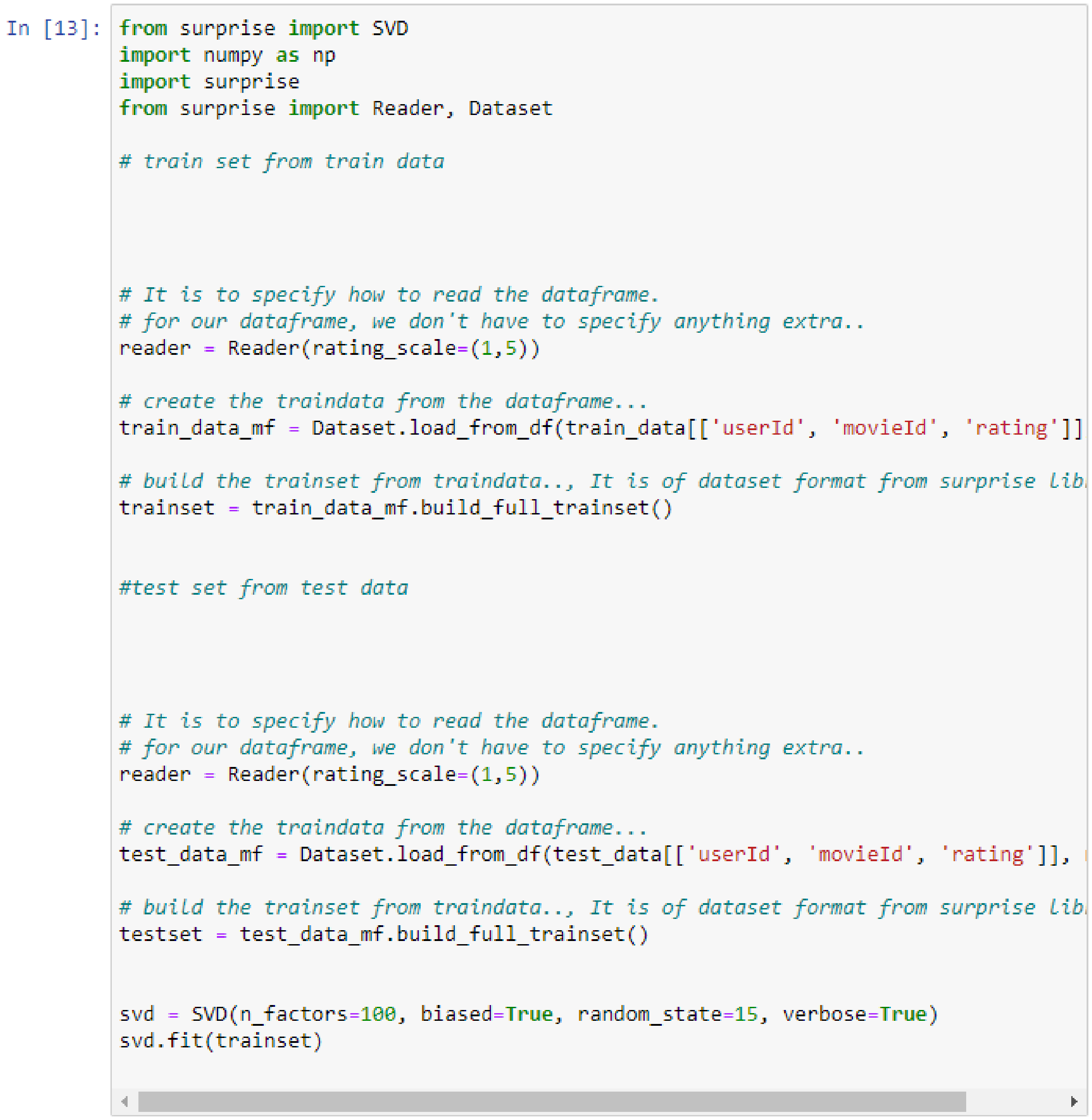
Matrix factorization algorithms work by decomposing the user-movie interaction matrix into the product of two lower dimensionality rectangular matrices, say U and M. The decomposition is done in such a way that the product results in almost similar values to the user-movie interaction matrix. Here, U represents the user matrix, M represents the movie matrix, n is the number of users, and m is the number of movies.

Each row of the user matrix represents a user and each column of the movie matrix represents a movie.



Once we obtain the U and M matrices, based on the non-empty cells in the user-movie interaction matrix, we perform the product of U and M and predict the values of non-empty cells in the user-movie interaction matrix.

To implement matrix factorization, we use a simple Python library named Surprise, which is for building and testing recommender systems. The data frame is converted into a train set, a format of data set to be accepted by the Surprise library.



```
In [13]: from surprise import SVD
         import numpy as np
         import surprise
         from surprise import Reader, Dataset

         # train set from train data




         # It is to specify how to read the dataframe.
         # for our dataframe, we don't have to specify anything extra..
         reader = Reader(rating_scale=(1,5))

         # create the traindata from the dataframe...
         train_data_mf = Dataset.load_from_df(train_data[['userId', 'movieId', 'rating']]

         # build the trainset from traindata.., It is of dataset format from surprise lib
         trainset = train_data_mf.build_full_trainset()


         #test set from test data




         # It is to specify how to read the dataframe.
         # for our dataframe, we don't have to specify anything extra..
         reader = Reader(rating_scale=(1,5))

         # create the traindata from the dataframe...
         test_data_mf = Dataset.load_from_df(test_data[['userId', 'movieId', 'rating']],

         # build the trainset from traindata.., It is of dataset format from surprise lib
         testset = test_data_mf.build_full_trainset()


         svd = SVD(n_factors=100, biased=True, random_state=15, verbose=True)
         svd.fit(trainset)
```

Now the model is ready. We'll store these predictions to pass to the final model as an additional feature. This will help us incorporate collaborative filtering into our system.

```
In [14]: # storing train predictions



         #getting predictions of trainset
         train_preds = svd.test(trainset.build_testset())

         train_pred_mf = np.array([pred.est for pred in train_preds])



         # storing test predictions



         #getting predictions of trainset
         test_preds = svd.test(testset.build_testset())

         test_pred_mf = np.array([pred.est for pred in test_preds])
```

```
In [15]: test_pred_mf

Out[15]: array([3.42586544, 3.22903645, 3.02600702, ..., 3.56157074, 3.92178227,
                3.32054949])
```

Note that we have to perform the above steps for test data also.

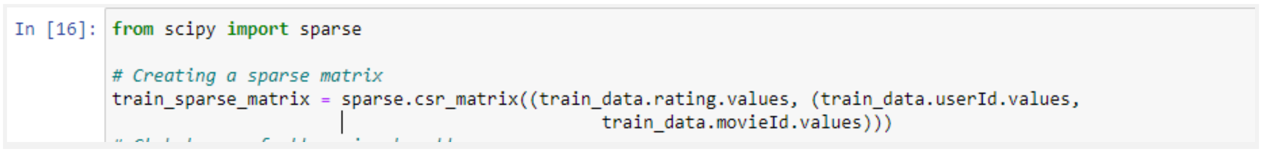
Step 2: Creating Handcrafted Features

Let’s convert the data in the data frame format into a user-movie interaction matrix. Matrices used in this type of problem are generally sparse because there’s a high chance users may only rate a few movies.

The advantages of the sparse matrix format of data, also called CSR format, are as follows:

- efficient arithmetic operations: CSR + CSR, CSR * CSR, etc.
- efficient row slicing
- fast matrix-vector products

scipy.sparse.csr_matrix is a utility function that efficiently converts the data frame into a sparse matrix.



```
In [16]: from scipy import sparse

         # Creating a sparse matrix
         train_sparse_matrix = sparse.csr_matrix((train_data.rating.values, (train_data.userId.values,
                                                  train_data.movieId.values)))
```

'train_sparse_matrix' is the sparse matrix representation of the train_data data frame.

We'll create 3 sets of features using this sparse matrix:

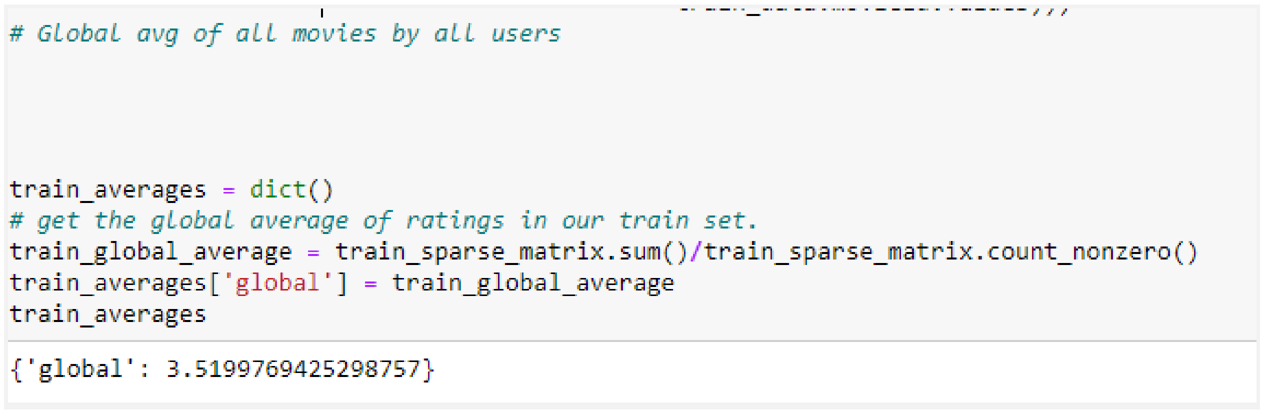
1. Features which represent global averages
2. Features which represent the top five similar users
3. Features which represent the top five similar movies

Let's take a look at how to prepare each in more detail.

1. Features which represent the global averages

The three global averages we'll employ are:

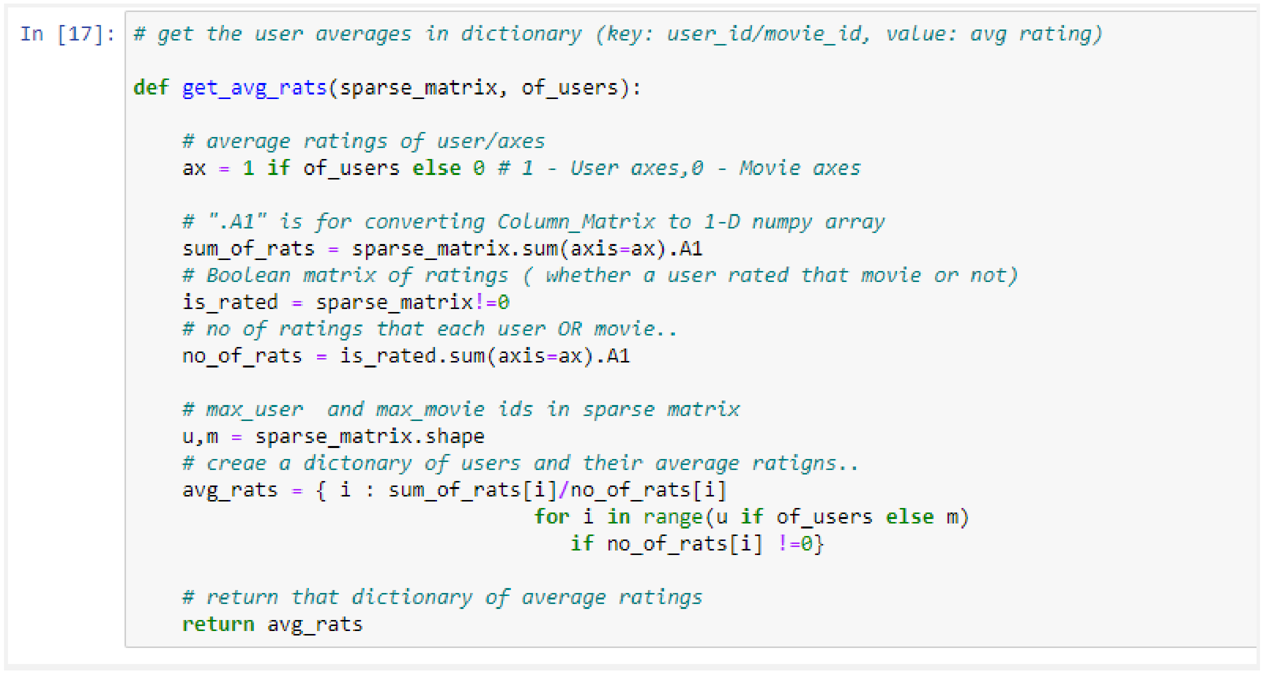
1. The average ratings of all movies given by all users
2. The average ratings of a particular movie given by all users
3. The average ratings of all movies given by a particular user



```
# Global avg of all movies by all users



train_averages = dict()
# get the global average of ratings in our train set.
train_global_average = train_sparse_matrix.sum()/train_sparse_matrix.count_nonzero()
train_averages['global'] = train_global_average
train_averages
```

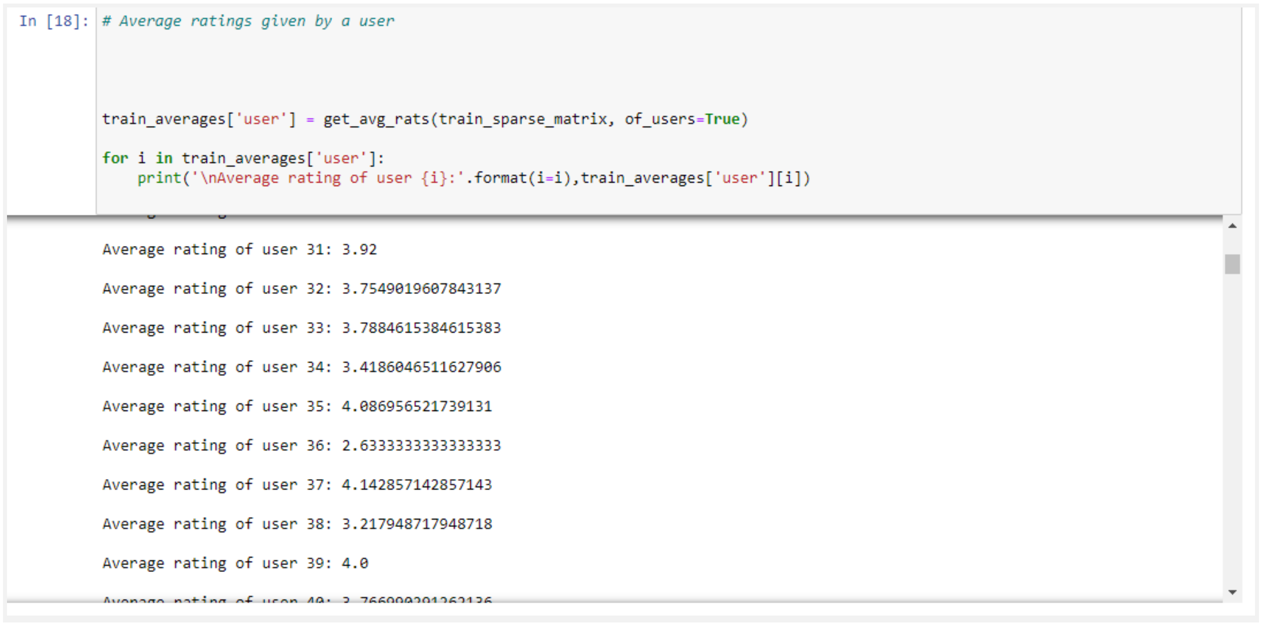
```
{'global': 3.5199769425298757}
```

Next, let's create a function which takes the sparse matrix as input and gives the average ratings of a movie given by all users, and the average rating of all movies given by a single user.



```
In [17]: # get the user averages in dictionary (key: user_id/movie_id, value: avg rating)

         def get_avg_rats(sparse_matrix, of_users):

             # average ratings of user/axes
             ax = 1 if of_users else 0 # 1 - User axes,0 - Movie axes

             # ".A1" is for converting Column_Matrix to 1-D numpy array
             sum_of_rats = sparse_matrix.sum(axis=ax).A1
             # Boolean matrix of ratings ( whether a user rated that movie or not)
             is_rated = sparse_matrix!=0
             # no of ratings that each user OR movie..
             no_of_rats = is_rated.sum(axis=ax).A1

             # max_user  and max_movie ids in sparse matrix
             u,m = sparse_matrix.shape
             # creae a dictonary of users and their average ratigns..
             avg_rats = { i : sum_of_rats[i]/no_of_rats[i]
                                      for i in range(u if of_users else m)
                                         if no_of_rats[i] !=0}

             # return that dictionary of average ratings
             return avg_rats
```

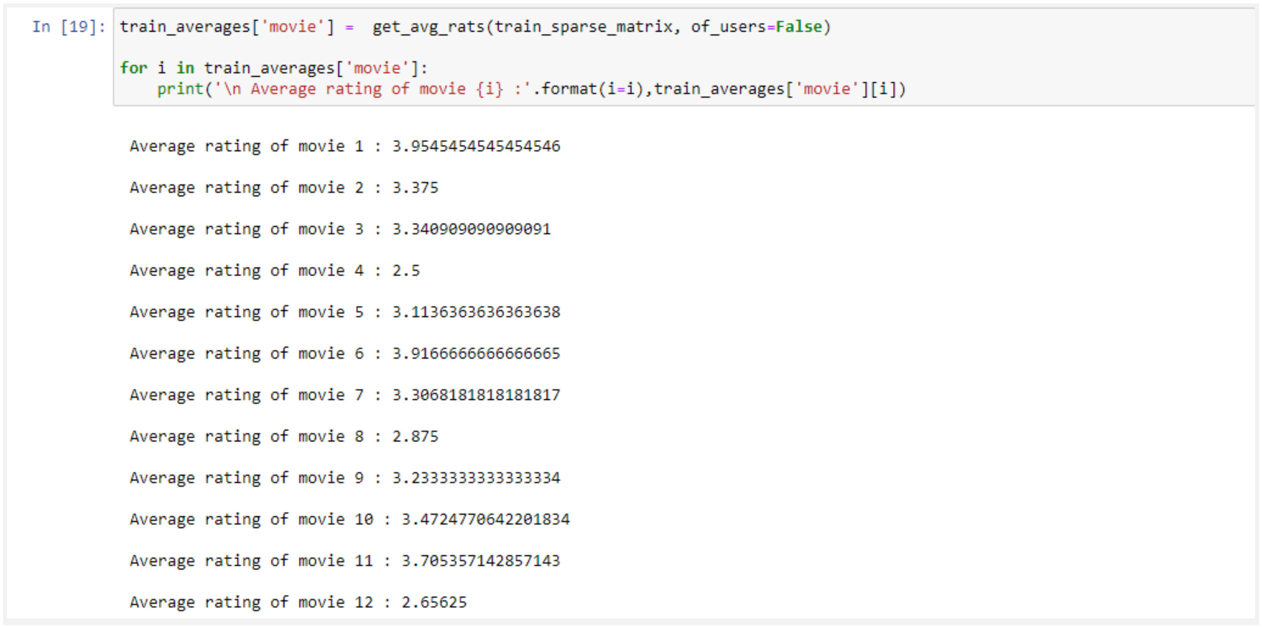
The average rating is given by a user:



```
In [18]: # Average ratings given by a user



         train_averages['user'] = get_avg_rats(train_sparse_matrix, of_users=True)

         for i in train_averages['user']:
             print('\nAverage rating of user {i}:'.format(i=i),train_averages['user'][i])
```

```
Average rating of user 31: 3.92

Average rating of user 32: 3.7549019607843137

Average rating of user 33: 3.7884615384615383

Average rating of user 34: 3.4186046511627906

Average rating of user 35: 4.086956521739131

Average rating of user 36: 2.6333333333333333

Average rating of user 37: 4.142857142857143

Average rating of user 38: 3.217948717948718

Average rating of user 39: 4.0

Average rating of user 40: 3.766990291262136
```

Average ratings are given for a movie:

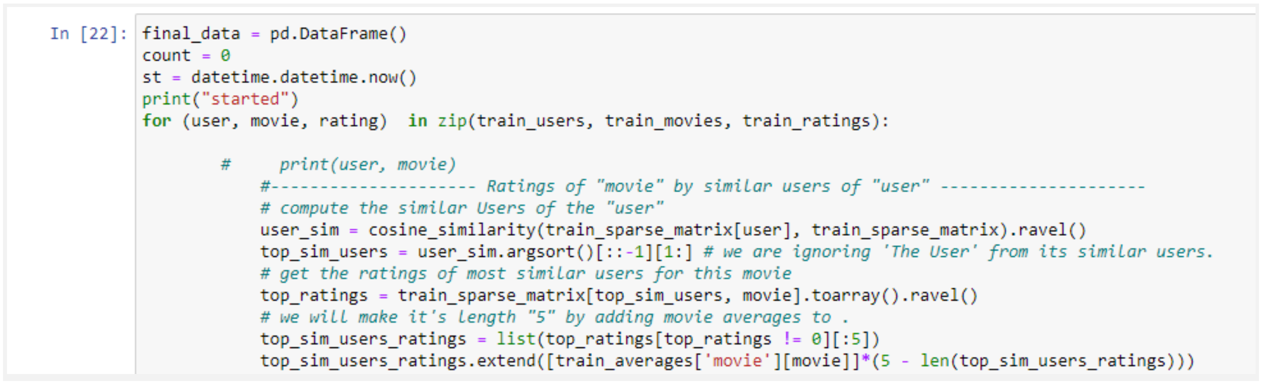


```
In [19]: train_averages['movie'] =  get_avg_rats(train_sparse_matrix, of_users=False)

         for i in train_averages['movie']:
             print('\n Average rating of movie {i} :'.format(i=i),train_averages['movie'][i])
```

```
 Average rating of movie 1 : 3.9545454545454546

 Average rating of movie 2 : 3.375

 Average rating of movie 3 : 3.340909090909091

 Average rating of movie 4 : 2.5

 Average rating of movie 5 : 3.1136363636363638

 Average rating of movie 6 : 3.9166666666666665

 Average rating of movie 7 : 3.3068181818181817

 Average rating of movie 8 : 2.875

 Average rating of movie 9 : 3.2333333333333334

 Average rating of movie 10 : 3.4724770642201834

 Average rating of movie 11 : 3.705357142857143

 Average rating of movie 12 : 2.65625
```

2. Features which represent the top 5 similar users

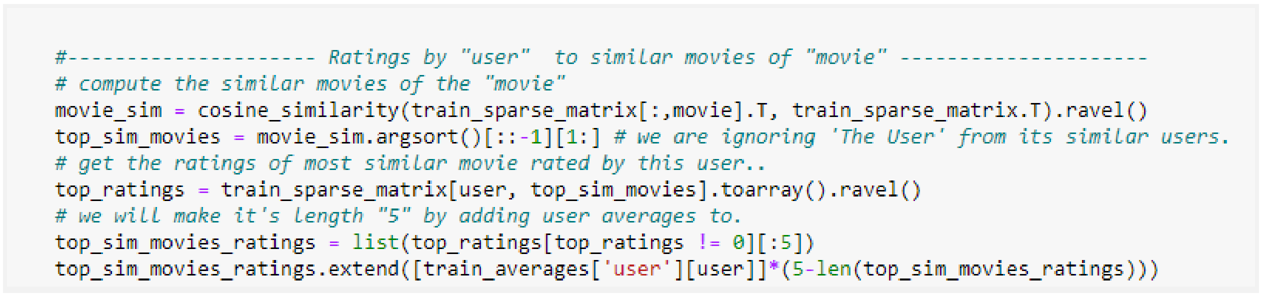
In this set of features, we will create the top 5 similar users who rated a particular movie. The similarity is calculated using the cosine similarity between the users.



```
In [22]: final_data = pd.DataFrame()
         count = 0
         st = datetime.datetime.now()
         print("started")
         for (user, movie, rating)  in zip(train_users, train_movies, train_ratings):

                 #     print(user, movie)
                     #-------------------- Ratings of "movie" by similar users of "user" --------------------
                     # compute the similar Users of the "user"
                     user_sim = cosine_similarity(train_sparse_matrix[user], train_sparse_matrix).ravel()
                     top_sim_users = user_sim.argsort()[::-1][1:] # we are ignoring 'The User' from its similar users.
                     # get the ratings of most similar users for this movie
                     top_ratings = train_sparse_matrix[top_sim_users, movie].toarray().ravel()
                     # we will make it's length "5" by adding movie averages to .
                     top_sim_users_ratings = list(top_ratings[top_ratings != 0][:5])
                     top_sim_users_ratings.extend([train_averages['movie'][movie]]*(5 - len(top_sim_users_ratings)))
```

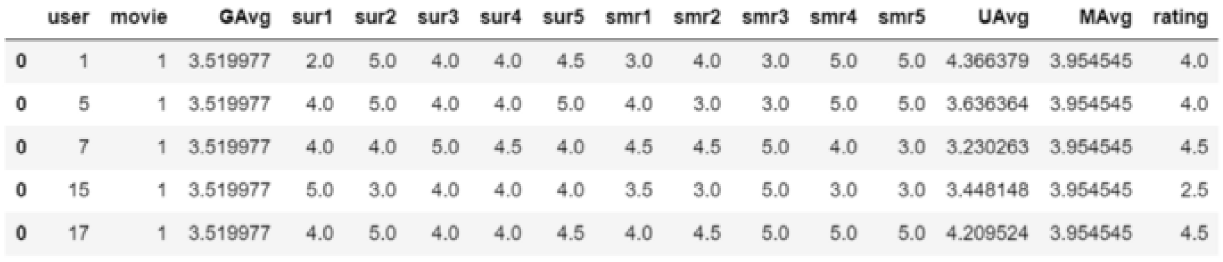
3. Features which represent the top 5 similar movies

In this set of features, we obtain the top 5 similar movies rated by a particular user. This similarity is calculated using the cosine similarity between the movies.



```
#--------------------- Ratings by "user"  to similar movies of "movie" ---------------------
# compute the similar movies of the "movie"
movie_sim = cosine_similarity(train_sparse_matrix[:,movie].T, train_sparse_matrix.T).ravel()
top_sim_movies = movie_sim.argsort()[::-1][1:] # we are ignoring 'The User' from its similar users.
# get the ratings of most similar movie rated by this user..
top_ratings = train_sparse_matrix[user, top_sim_movies].toarray().ravel()
# we will make it's length "5" by adding user averages to.
top_sim_movies_ratings = list(top_ratings[top_ratings != 0][:5])
top_sim_movies_ratings.extend([train_averages['user'][user]]*(5-len(top_sim_movies_ratings)))
```

We append all these features for each movie-user pair and create a data frame. Figure 5 is a snapshot of our data frame.



	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UAvg	MAvg	rating
0	1	1	3.519977	2.0	5.0	4.0	4.0	4.5	3.0	4.0	3.0	5.0	5.0	4.366379	3.954545	4.0
0	5	1	3.519977	4.0	5.0	4.0	4.0	5.0	4.0	3.0	3.0	5.0	5.0	3.636364	3.954545	4.0
0	7	1	3.519977	4.0	4.0	5.0	4.5	4.0	4.5	4.5	5.0	4.0	3.0	3.230263	3.954545	4.5
0	15	1	3.519977	5.0	3.0	4.0	4.0	4.0	3.5	3.0	5.0	3.0	3.0	3.448148	3.954545	2.5
0	17	1	3.519977	4.0	5.0	4.0	4.0	4.5	4.0	4.5	5.0	5.0	5.0	4.209524	3.954545	4.5

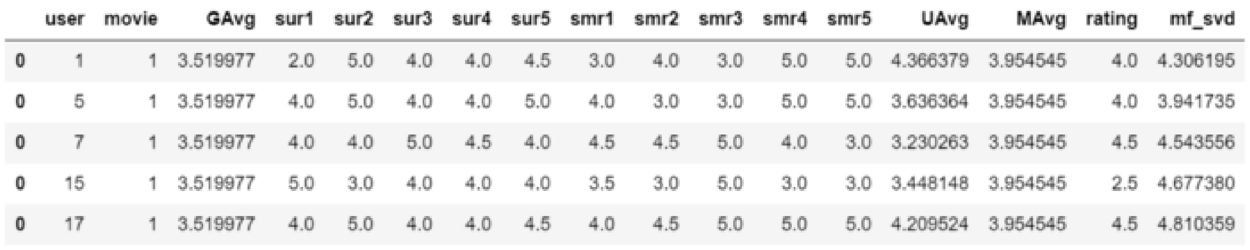
Figure 5: Overview of data with 13 features

Exploratory Data analysis

Here's a more detailed breakdown of its contents:

- GAvg: Average rating of all ratings
- Similar users rating of this movie: sur1, sur2, sur3, sur4, sur5 (top 5 similar users who rated that movie)
- Similar movies rated by this user: smr1, smr2, smr3, smr4, smr5 (top 5 similar movies rated by user)
- UAvg: User AVerage rating
- MAvg: Average rating of this movie
- rating: Rating of this movie by this user.

Once we have these 13 features ready, we'll add the Matrix Factorization output as the 14th feature. In Figure 6 you can see a snapshot of our data after adding the output from Step 1.



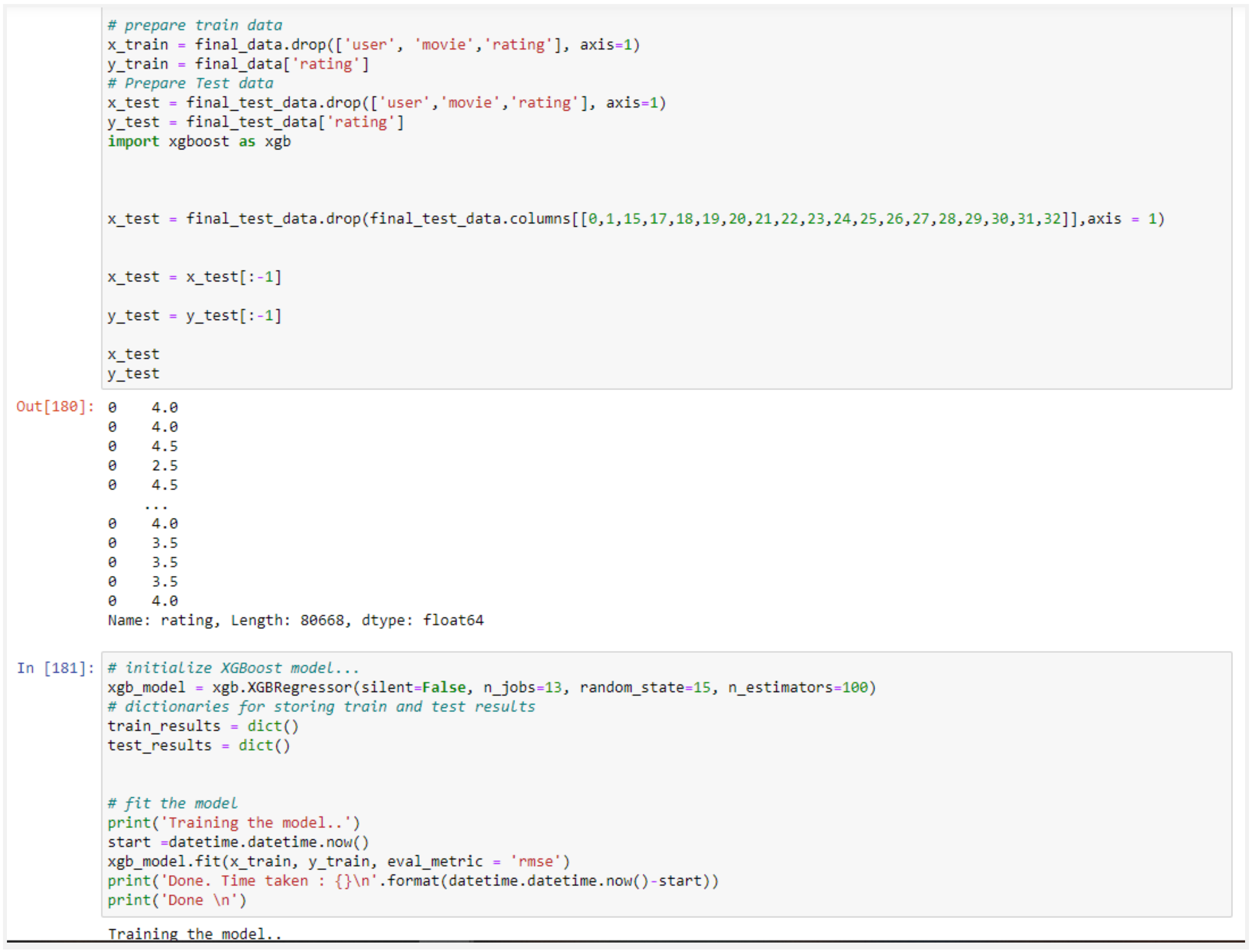
	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UAvg	MAvg	rating	mf_svd
0	1	1	3.519977	2.0	5.0	4.0	4.0	4.5	3.0	4.0	3.0	5.0	5.0	4.366379	3.954545	4.0	4.306195
0	5	1	3.519977	4.0	5.0	4.0	4.0	5.0	4.0	3.0	3.0	5.0	5.0	3.636364	3.954545	4.0	3.941735
0	7	1	3.519977	4.0	4.0	5.0	4.5	4.0	4.5	4.5	5.0	4.0	3.0	3.230263	3.954545	4.5	4.543556
0	15	1	3.519977	5.0	3.0	4.0	4.0	4.0	3.5	3.0	5.0	3.0	3.0	3.448148	3.954545	2.5	4.677380
0	17	1	3.519977	4.0	5.0	4.0	4.0	4.5	4.0	4.5	5.0	5.0	5.0	4.209524	3.954545	4.5	4.810359

Figure 6: Overview of data with 13 features and matrix factorization output(Image by author)

The last column, named, mf_svd, is the additional column that contains the output of the model performed in Step 1.

Step 3: Creating a final model for our movie recommendation system

To create our final model, let’s use XGBoost, an optimized distributed gradient boosting library.



```
# prepare train data
x_train = final_data.drop(['user', 'movie','rating'], axis=1)
y_train = final_data['rating']
# Prepare Test data
x_test = final_test_data.drop(['user','movie','rating'], axis=1)
y_test = final_test_data['rating']
import xgboost as xgb


x_test = final_test_data.drop(final_test_data.columns[[0,1,15,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32]],axis = 1)


x_test = x_test[:-1]

y_test = y_test[:-1]

x_test
y_test
```

```
Out[180]: 0    4.0
          0    4.0
          0    4.5
          0    2.5
          0    4.5
              ...
          0    4.0
          0    3.5
          0    3.5
          0    3.5
          0    4.0
          Name: rating, Length: 80668, dtype: float64
```

```
In [181]: # initialize XGBoost model...
xgb_model = xgb.XGBRegressor(silent=False, n_jobs=13, random_state=15, n_estimators=100)
# dictionaries for storing train and test results
train_results = dict()
test_results = dict()


# fit the model
print('Training the model..')
start =datetime.datetime.now()
xgb_model.fit(x_train, y_train, eval_metric = 'rmse')
print('Done. Time taken : {}\n'.format(datetime.datetime.now()-start))
print('Done \n')
```

```
Training the model..
```

Hypothesis Testing

Performance Metrics

There are two main ways to evaluate a recommender system's performance: Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE). RMSE measures the squared loss, while MAPE measures the absolute loss. Lower values mean lower error rates and thus better performance.

Both are good as they allow for easy interpretation. Let's take a look at what each of them is:

Root Mean Squared Error (RMSE)

RMSE is the square root of the average of squared errors and is given by the below formula.

$$\text{RMSE} = \sqrt{\frac{(r-r\hat{})^2}{N}}$$

Where:

r is the actual rating,

r^ is the predicted ratings and

N is the total number of predictions

Mean Absolute Percentage Error (MAPE)

MAPE measures the error in percentage terms. It is given by the formula below:

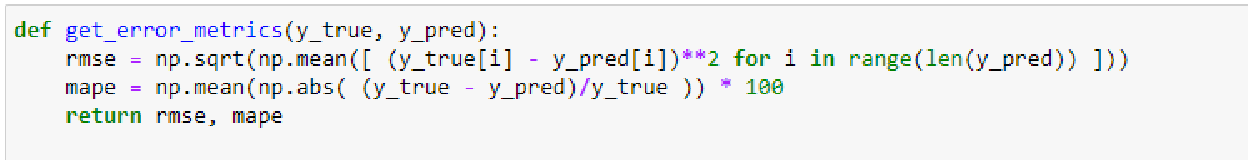
$$MAPE = \frac{1}{N} \sum \frac{|r - r^\wedge|}{r} \text{ X } 100$$

Where:

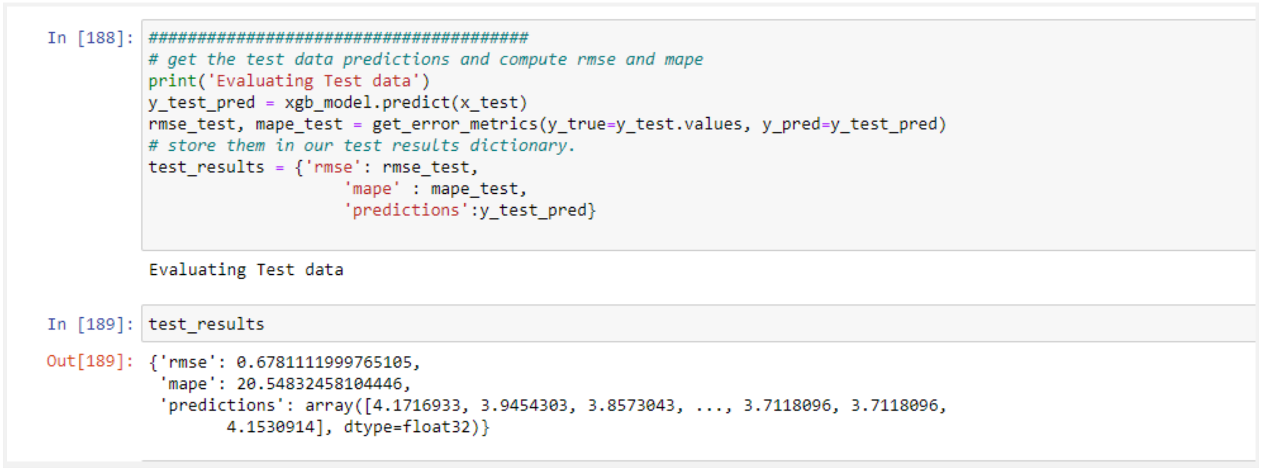
r is the actual rating,

r^ is the predicted ratings and

N is the total number of predictions



```
def get_error_metrics(y_true, y_pred):
    rmse = np.sqrt(np.mean([ (y_true[i] - y_pred[i])**2 for i in range(len(y_pred)) ]))
    mape = np.mean(np.abs( (y_true - y_pred)/y_true )) * 100
    return rmse, mape
```



```
In [188]: ##########################################
          # get the test data predictions and compute rmse and mape
          print('Evaluating Test data')
          y_test_pred = xgb_model.predict(x_test)
          rmse_test, mape_test = get_error_metrics(y_true=y_test.values, y_pred=y_test_pred)
          # store them in our test results dictionary.
          test_results = {'rmse': rmse_test,
                              'mape' : mape_test,
                              'predictions':y_test_pred}

          Evaluating Test data

In [189]: test_results

Out[189]: {'rmse': 0.6781111999765105,
           'mape': 20.54832458104446,
           'predictions': array([4.1716933, 3.9454303, 3.8573043, ..., 3.7118096, 3.7118096,
                  4.1530914], dtype=float32)}
```

Communication of Insights Attained

Our model resulted in 0.68 RMSE, and 20.55 MAPE on the unseen test data, which is a good and usable model. An RMSE value of less than 2 is considered good, and a MAPE less than 25 is excellent.

Communication of Approach

That said, this model can be further enhanced by adding features that would be recommended based on the top picks dependent on location or genre. We could also test the efficacy of our various models in real-time through A/B testing.

Summary

In this article, we learned the importance of recommender systems, the types of recommender systems being implemented, and how to use matrix factorization to enhance a system. We then built a movie recommendation system that considers user-user similarity, movie-movie similarity, global averages, and matrix factorization. These concepts can be applied to any other user-item interactions systems.