

# Matrix Factorization: Designing a Netflix recommendation system from scratch



Rohan Thoma



## 1. Business Problem

### 1.1 Problem Description

Netflix is all about connecting people to the movies they love. To help customers find those movies, they developed world-class movie recommendation system: CinematchSM. Its job is to predict whether someone will enjoy a movie based on how much they liked or disliked other movies. Netflix use those predictions to make personal movie recommendations based on each customer's unique tastes. And while **Cinematch** is doing pretty well, it can always be made better.

Now there are a lot of interesting alternative approaches to how Cinematch works that netflix haven't tried. Some are described in the literature, some aren't. We're curious whether any of these can beat Cinematch by making better predictions. Because, frankly, if there is a much better approach it could make a big difference to our customers and our business.

Credits: <https://www.netflixprize.com/rules.html>

### 1.2 Problem Statement

Netflix provided a lot of anonymous rating data, and a prediction accuracy bar that is 10% better than what Cinematch can do on the same training data set. (Accuracy is a measurement of how closely predicted ratings of movies match subsequent actual ratings.)

### 1.3 Sources

- <https://www.netflixprize.com/rules.html>
- <https://www.kaggle.com/netflix-inc/netflix-prize-data>
- Netflix blog: <https://medium.com/netflix-techblog/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429> (very nice blog)
- surprise library: <http://surpriselib.com/> (we use many models from this library)
- surprise library doc: [http://surprise.readthedocs.io/en/stable/getting\\_started.html](http://surprise.readthedocs.io/en/stable/getting_started.html) (we use many models from this library)
- installing surprise: <https://github.com/NicolasHug/Surprise#installation>
- Research paper: <http://courses.ischool.berkeley.edu/i290-dm/s11/SECURE/a1-koren.pdf> (most of our work was inspired by this paper)
- SVD Decomposition : <https://www.youtube.com/watch?v=P5mlg91as1c>

## 1.4 Real world/Business Objectives and constraints

Objectives:

1. Predict the rating that a user would give to a movie that he has not yet rated.
2. Minimize the difference between predicted and actual rating (RMSE and MAPE)

Constraints:

1. Some form of interpretability.

## 2. Machine Learning Problem

### 2.1 Data

#### 2.1.1 Data Overview

Get the data from : <https://www.kaggle.com/netflix-inc/netflix-prize-data/data>

Data files :

- combined\_data\_1.txt
- combined\_data\_2.txt
- combined\_data\_3.txt
- combined\_data\_4.txt
- movie\_titles.csv

The first line of each file [combined\_data\_1.txt, combined\_data\_2.txt, combined\_data\_3.txt, combined\_data\_4.txt] contains the movie id followed by a colon. Each subsequent line in the file corresponds to a rating from a customer and its date in the following format:

CustomerID,Rating,Date

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

#### 2.1.2 Example Data point

```
1:  
1488844,3,2005-09-06  
822109,5,2005-05-13
```

885013,4,2005-10-19  
30878,4,2005-12-26  
823519,3,2004-05-03  
893988,3,2005-11-17  
124105,4,2004-08-05  
1248029,3,2004-04-22  
1842128,4,2004-05-09  
2238063,3,2005-05-11  
1503895,4,2005-05-19  
2207774,5,2005-06-06  
2590061,3,2004-08-12  
2442,3,2004-04-14  
543865,4,2004-05-28  
1209119,4,2004-03-23  
804919,4,2004-06-10  
1086807,3,2004-12-28  
1711859,4,2005-05-08  
372233,5,2005-11-23  
1080361,3,2005-03-28  
1245640,3,2005-12-19  
558634,4,2004-12-14  
2165002,4,2004-04-06  
1181550,3,2004-02-01  
1227322,4,2004-02-06  
427928,4,2004-02-26  
814701,5,2005-09-29  
808731,4,2005-10-31  
662870,5,2005-08-24  
337541,5,2005-03-23  
786312,3,2004-11-16  
1133214,4,2004-03-07  
1537427,4,2004-03-29  
1209954,5,2005-05-09  
2381599,3,2005-09-12  
525356,2,2004-07-11  
1910569,4,2004-04-12  
2263586,4,2004-08-20  
2421815,2,2004-02-26  
1009622,1,2005-01-19  
1481961,2,2005-05-24  
401047,4,2005-06-03  
2179073,3,2004-08-29  
1434636,3,2004-05-01  
93986,5,2005-10-06  
1308744,5,2005-10-29  
2647871,4,2005-12-30  
1905581,5,2005-08-16  
2508819,3,2004-05-18  
1578279,1,2005-05-19  
1159695,4,2005-02-15  
2588432,3,2005-03-31  
2423091,3,2005-09-12  
470232,4,2004-04-08  
2148699,2,2004-06-05  
1342007,3,2004-07-16  
466135,4,2004-07-13  
2472440,3,2005-08-13  
1283744,3,2004-04-17  
1927580,4,2004-11-08  
716874,5,2005-05-06  
4326,4,2005-10-29

## 2.2 Mapping the real world problem to a Machine Learning Problem

### 2.2.1 Type of Machine Learning Problem

For a given movie and user we need to predict the rating would be given by him/her to the movie.  
The given problem is a Recommendation problem  
It can also seen as a Regression problem

## 2.2.2 Performance metric

- Mean Absolute Percentage Error: [https://en.wikipedia.org/wiki/Mean\\_absolute\\_percentage\\_error](https://en.wikipedia.org/wiki/Mean_absolute_percentage_error)
- Root Mean Square Error: [https://en.wikipedia.org/wiki/Root-mean-square\\_deviation](https://en.wikipedia.org/wiki/Root-mean-square_deviation)

## 2.2.3 Machine Learning Objective and Constraints

1. Minimize RMSE.
2. Try to provide some interpretability.

```
In [1]: # this is just to know how much time will it take to run this entire ipython notebook
from datetime import datetime
# globalstart = datetime.now()
import pandas as pd
import numpy as np
import matplotlib
matplotlib.use('nbagg')

import matplotlib.pyplot as plt
plt.rcParams.update({'figure.max_open_warning': 0})

import seaborn as sns
sns.set_style('whitegrid')
import os
from scipy import sparse
from scipy.sparse import csr_matrix

from sklearn.decomposition import TruncatedSVD
from sklearn.metrics.pairwise import cosine_similarity
import random
```

# 3. Exploratory Data Analysis

## 3.1 Preprocessing

### 3.1.1 Converting / Merging whole data to required format: u\_i, m\_j, r\_ij

```
In [ ]: start = datetime.now()
if not os.path.isfile('data.csv'):
    # Create a file 'data.csv' before reading it
    # Read all the files in netflix and store them in one big file('data.csv')
    # We re reading from each of the four files and appendig each rating to a global file 'train.csv'
    data = open('data.csv', mode='w')

    row = list()
    files=['data_folder/combined_data_1.txt','data_folder/combined_data_2.txt',
           'data_folder/combined_data_3.txt', 'data_folder/combined_data_4.txt']
    for file in files:
        print("Reading ratings from {}...".format(file))
        with open(file) as f:
            for line in f:
                del row[:] # you don't have to do this.
                line = line.strip()
                if line.endswith(':'):
                    # All below are ratings for this movie, until another movie appears.
                    movie_id = line.replace(':', '')
                else:
                    row = [x for x in line.split(',')]
```

```

        row.insert(0, movie_id)
        data.write(','.join(row))
        data.write('\n')
    print("Done.\n")
    data.close()
print('Time taken :', datetime.now() - start)

```

Reading ratings from data\_folder/combined\_data\_1.txt...  
Done.

Reading ratings from data\_folder/combined\_data\_2.txt...  
Done.

Reading ratings from data\_folder/combined\_data\_3.txt...  
Done.

Reading ratings from data\_folder/combined\_data\_4.txt...  
Done.

Time taken : 0:05:03.705966

```
In [1]: import pandas as pd

print("creating the dataframe from data.csv file..")
df = pd.read_csv('train.csv', sep=',',
                  names=['movie', 'user', 'rating', 'date'], low_memory=False)
df.date = pd.to_datetime(df.date, errors='coerce')
print('Done.\n')

# we are arranging the ratings according to time.
print('Sorting the dataframe by date..')
df.sort_values(by='date', inplace=True)
print('Done..')
```

creating the dataframe from data.csv file..  
Done.

Sorting the dataframe by date..  
Done..

```
In [2]: df.head()
```

```
Out[2]:   movie  user  rating      date
1  10341  510180      4  1999-11-11
28  17764  510180      5  1999-11-11
26  1367   510180      5  1999-11-11
25   8079  510180      2  1999-11-11
24  15105  510180      4  1999-11-11
```

```
In [3]: df['rating'].describe()
```

```
Out[3]: count    80384406
unique         6
top            4
freq    27161596
Name: rating, dtype: object
```

### 3.1.2 Checking for NaN values

```
In [4]: # just to make sure that all Nan containing rows are deleted..
print("No of Nan values in our dataframe : ", sum(df.isnull().any()))
```

No of Nan values in our dataframe : 1

```
In [5]: df.dropna(inplace=True, axis=0)

print("No of Nan values in our dataframe : ", sum(df.isnull().any()))
```

No of Nan values in our dataframe : 0

```
In [6]: df['rating'].describe()
```

```
Out[6]: count    80384405
unique      5
top         4
freq    27161596
Name: rating, dtype: object
```

### 3.1.3 Removing Duplicates

```
In [13]: dup_bool = df.duplicated(['movie','user','rating'])
```

```
In [12]: print(dup_bool.any())
```

```
False
```

### 3.1.4 Basic Statistics (#Ratings, #Users, and #Movies)

```
In [14]: import numpy as np

print("Total data ")
print("-"*50)
print("\nTotal no of ratings : ",df.shape[0])
print("Total No of Users   : ", len(np.unique(df.user)))
print("Total No of movies  : ", len(np.unique(df.movie)))
```

```
Total data
-----
```

```
Total no of ratings : 80384405
Total No of Users   : 405041
Total No of movies  : 17424
```

## 3.2 Splitting data into Train and Test(80:20)

```
In [15]: if not os.path.isfile('train.csv'):
    # create the dataframe and store it in the disk for offline purposes..
    df.iloc[:int(df.shape[0]*0.80)].to_csv("train.csv", index=False)

if not os.path.isfile('test.csv'):
    # create the dataframe and store it in the disk for offline purposes..
    df.iloc[int(df.shape[0]*0.80):].to_csv("test.csv", index=False)

train_df = pd.read_csv("train.csv", parse_dates=['date'])
test_df = pd.read_csv("test.csv")
```

### 3.2.1 Basic Statistics in Train data (#Ratings, #Users, and #Movies)

```
In [16]: # movies = train_df.movie.value_counts()
# users = train_df.user.value_counts()
print("Training data ")
print("-"*50)
print("\nTotal no of ratings : ",train_df.shape[0])
print("Total No of Users   : ", len(np.unique(train_df.user)))
print("Total No of movies  : ", len(np.unique(train_df.movie)))
```

```
Training data
-----
```

```
Total no of ratings : 64307524
Total No of Users   : 344884
Total No of movies  : 16727
```

### 3.2.2 Basic Statistics in Test data (#Ratings, #Users, and #Movies)

```
In [17]: print("Test data ")
print("-"*50)
print("\nTotal no of ratings : ",test_df.shape[0])
print("Total No of Users   : ", len(np.unique(test_df.user)))
print("Total No of movies  : ", len(np.unique(test_df.movie)))
```

```
Test data
```

```
Total no of ratings : 16076881
Total No of Users   : 307632
Total No of movies  : 17377
```

### 3.3 Exploratory Data Analysis on Train data

```
In [18]: # method to make y-axis more readable
def human(num, units = 'M'):
    units = units.lower()
    num = float(num)
    if units == 'k':
        return str(num/10**3) + " K"
    elif units == 'm':
        return str(num/10**6) + " M"
    elif units == 'b':
        return str(num/10**9) + " B"
```

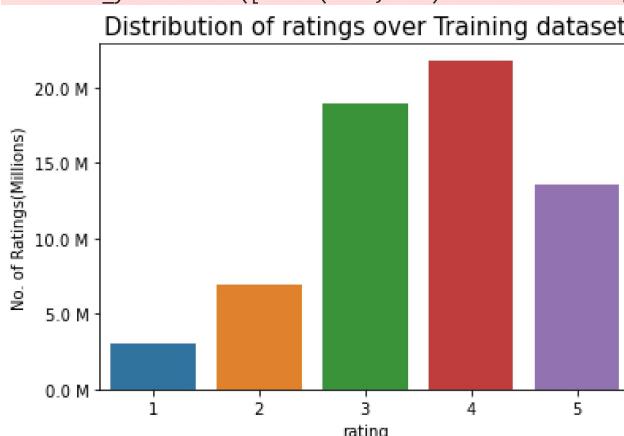
#### 3.3.1 Distribution of ratings

```
In [21]: import matplotlib.pyplot as plt
import seaborn as sns

fig, ax = plt.subplots()
plt.title('Distribution of ratings over Training dataset', fontsize=15)
sns.countplot(train_df.rating)
ax.set_yticklabels([human(item, 'M') for item in ax.get_yticks()])
ax.set_ylabel('No. of Ratings(Millions)')

plt.show()
```

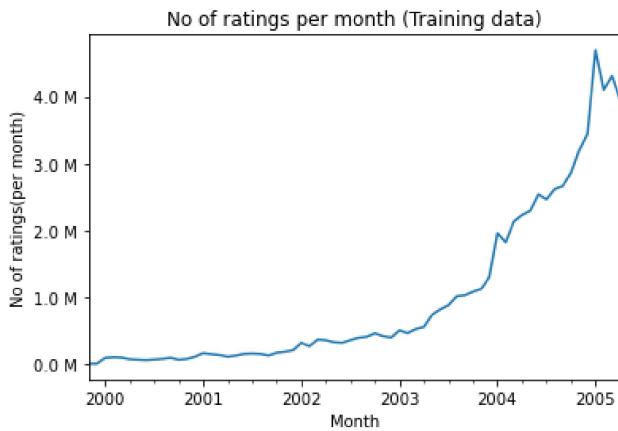
```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass the following variable as a keyword arg: x. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an explicit keyword will result in an error or misinterpretation.
  warnings.warn(
C:\Users\OVER_L~1\AppData\Local\Temp\ipykernel_13528/2082559964.py:7: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax.set_yticklabels([human(item, 'M') for item in ax.get_yticks()])
```



#### 3.3.2 Number of Ratings per a month

```
In [22]: ax = train_df.resample('m', on='date')['rating'].count().plot()
ax.set_title('No of ratings per month (Training data)')
plt.xlabel('Month')
plt.ylabel('No of ratings(per month)')
ax.set_yticklabels([human(item, 'M') for item in ax.get_yticks()])
plt.show()
```

```
C:\Users\OVER_L~1\AppData\Local\Temp\ipykernel_13528/4031331980.py:5: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax.set_yticklabels([human(item, 'M') for item in ax.get_yticks()])
```



### 3.3.3 Analysis on the Ratings given by user

```
In [23]: no_of_rated_movies_per_user = train_df.groupby(by='user')[['rating']].count().sort_values(ascending=False)
no_of_rated_movies_per_user.head()
```

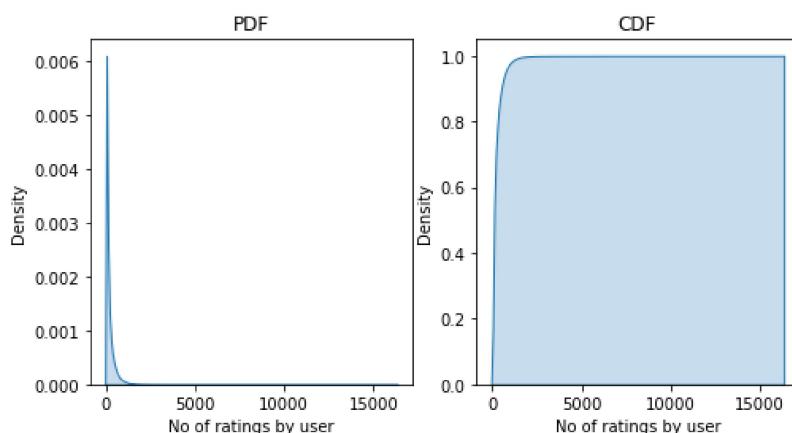
```
Out[23]: user
305344    16294
2439493   14915
387418    14437
1639792   9761
1461435   8379
Name: rating, dtype: int64
```

```
In [24]: fig = plt.figure(figsize=plt.figaspect(.5))

ax1 = plt.subplot(121)
sns.kdeplot(no_of_rated_movies_per_user, shade=True, ax=ax1)
plt.xlabel('No of ratings by user')
plt.title("PDF")

ax2 = plt.subplot(122)
sns.kdeplot(no_of_rated_movies_per_user, shade=True, cumulative=True, ax=ax2)
plt.xlabel('No of ratings by user')
plt.title('CDF')

plt.show()
```



```
In [25]: no_of_rated_movies_per_user.describe()
```

```
Out[25]: count    344884.000000
mean     186.461315
std      280.486668
min      1.000000
25%     26.000000
50%     82.000000
75%    231.000000
max    16294.000000
Name: rating, dtype: float64
```

There, is something interesting going on with the quantiles..

```
In [26]: quantiles = no_of_rated_movies_per_user.quantile(np.arange(0,1.01,0.01), interpolation='higher')
```

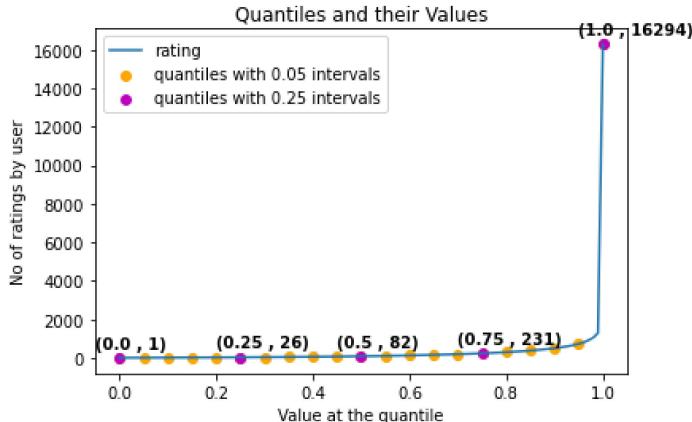
```
In [27]: plt.title("Quantiles and their Values")
quantiles.plot()
# quantiles with 0.05 difference
plt.scatter(x=quantiles.index[::5], y=quantiles.values[::5], c='orange', label="quantiles with 0.05 interv
# quantiles with 0.25 difference
plt.scatter(x=quantiles.index[::25], y=quantiles.values[::25], c='m', label = "quantiles with 0.25 interv
plt.ylabel('No of ratings by user')
plt.xlabel('Value at the quantile')
plt.legend(loc='best')

# annotate the 25th, 50th, 75th and 100th percentile values....
for x,y in zip(quantiles.index[::25], quantiles[::25]):
    plt.annotate(s="({} , {})".format(x,y), xy=(x,y), xytext=(x-0.05, y+500)
                ,fontweight='bold')

plt.show()
```

C:\Users\OVER\_L~1\AppData\Local\Temp\ipykernel\_13528/4210568691.py:13: MatplotlibDeprecationWarning: The 's' parameter of annotate() has been renamed 'text' since Matplotlib 3.3; support for the old name will be dropped two minor releases later.

```
plt.annotate(s="({} , {})".format(x,y), xy=(x,y), xytext=(x-0.05, y+500)
```



```
In [28]: quantiles[::5]
```

```
Out[28]: 0.00      1
0.05      4
0.10     10
0.15     15
0.20     20
0.25     26
0.30     34
0.35     43
0.40     54
0.45     67
0.50     82
0.55    101
0.60    124
0.65    152
0.70    186
0.75    231
0.80    290
0.85    371
0.90    494
0.95    717
1.00   16294
Name: rating, dtype: int64
```

how many ratings at the last 5% of all ratings??

```
In [29]: print('\n No of ratings at last 5 percentile : {} \n'.format(sum(no_of_rated_movies_per_user>= 717)))
```

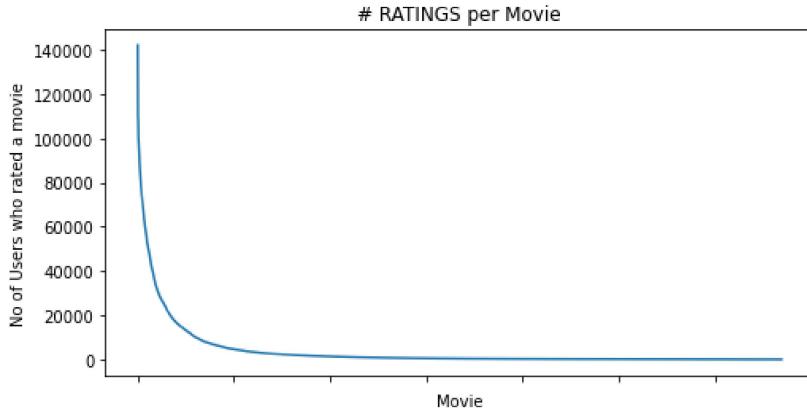
No of ratings at last 5 percentile : 17294

### 3.3.4 Analysis of ratings of a movie given by a user

```
In [30]: no_of_ratings_per_movie = train_df.groupby(by='movie')['rating'].count().sort_values(ascending=False)

fig = plt.figure(figsize=plt.figaspect(.5))
ax = plt.gca()
plt.plot(no_of_ratings_per_movie.values)
plt.title('# RATINGS per Movie')
plt.xlabel('Movie')
plt.ylabel('No of Users who rated a movie')
ax.set_xticklabels([])

plt.show()
```



- It is very skewed.. just like number of ratings given per user.
  - There are some movies (which are very popular) which are rated by huge number of users.
  - But most of the movies (like 90%) got some hundreds of ratings.

### 3.3.5 Number of ratings on each day of the week

Add new column (week day) to the data set for analysis.

```
In [35]: # It is used to skip the warning ''SettingWithCopyWarning''..
pd.options.mode.chained_assignment = None # default='warn'

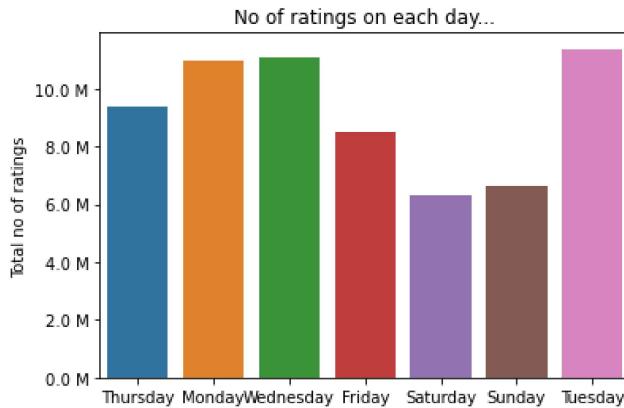
train_df['day_of_week'] = train_df.date.dt.day_name()

train_df.tail()
```

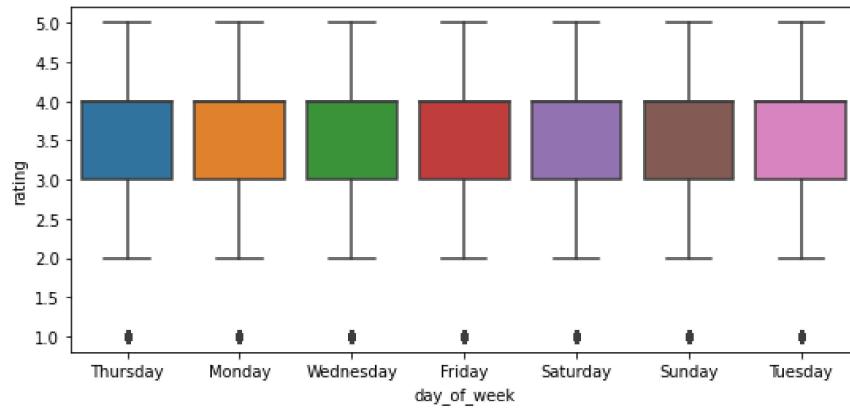
```
Out[35]:      movie    user  rating      date day_of_week
64307519    13392   75681      4  2005-04-29     Friday
64307520     6972  1442500      3  2005-04-29     Friday
64307521     6972  1574895      5  2005-04-29     Friday
64307522    10947   946655      5  2005-04-29     Friday
64307523    10947  2251285      5  2005-04-29     Friday
```

```
In [36]: fig, ax = plt.subplots()
sns.countplot(x='day_of_week', data=train_df, ax=ax)
plt.title('No of ratings on each day... ')
plt.ylabel('Total no of ratings')
plt.xlabel('')
ax.set_yticklabels([human(item, 'M') for item in ax.get_yticks()])
plt.show()
```

```
C:\Users\OVER_L~1\AppData\Local\Temp\ipykernel_13528\941500076.py:6: UserWarning: FixedFormatter should only  
be used together with FixedLocator  
ax.set_yticklabels([human(item, 'M') for item in ax.get_yticks()])
```



```
In [38]: from datetime import datetime  
  
start = datetime.now()  
fig = plt.figure(figsize=plt.figaspect(.45))  
sns.boxplot(y='rating', x='day_of_week', data=train_df)  
plt.show()  
print(datetime.now() - start)
```



0:00:21.769509

```
In [39]: avg_week_df = train_df.groupby(by=['day_of_week'])['rating'].mean()  
print(" Average ratings")  
print("-"*30)  
print(avg_week_df)  
print("\n")
```

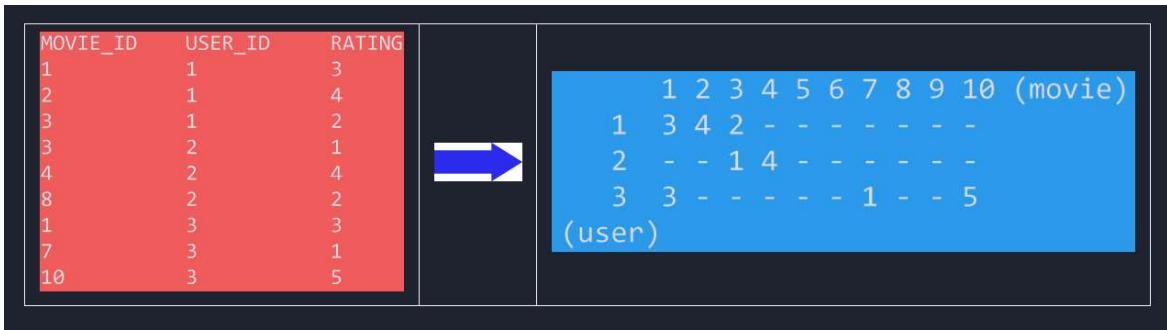
Average ratings

---

day_of_week	rating
Friday	3.557612
Monday	3.553024
Saturday	3.564901
Sunday	3.568389
Thursday	3.558272
Tuesday	3.549099
Wednesday	3.561520

Name: rating, dtype: float64

### 3.3.6 Creating sparse matrix from data frame



### 3.3.6.1 Creating sparse matrix from train data frame

```
In [26]: from scipy import sparse
from scipy.sparse import csr_matrix

start = datetime.now()
if os.path.isfile('train_sparse_matrix.npz'):
    print("It is present in your pwd, getting it from disk....")
    # just get it from the disk instead of computing it
    train_sparse_matrix = sparse.load_npz('train_sparse_matrix.npz')
    print("DONE..")
else:
    print("We are creating sparse_matrix from the dataframe..")
    # create sparse_matrix and store it for after usage.
    # csr_matrix(data_values, (row_index, col_index), shape_of_matrix)
    # It should be in such a way that, MATRIX[row, col] = data
    train_sparse_matrix = sparse.csr_matrix((train_df.rating.values, (train_df.user.values, train_df.movie

    print('Done. It\'s shape is : (user, movie) : ',train_sparse_matrix.shape)
    print('Saving it into disk for furthur usage..')
    # save it into disk
    sparse.save_npz("train_sparse_matrix.npz", train_sparse_matrix)
    print('Done..\n')

print(datetime.now() - start)
```

It is present in your pwd, getting it from disk....  
DONE..  
0:00:01.735961

### The Sparsity of Train Sparse Matrix

```
In [42]: us,mv = train_sparse_matrix.shape
elem = train_sparse_matrix.count_nonzero()

print("Sparsity Of Train matrix : {} % ".format( (1-(elem/(us*mv))) * 100 ) )
```

Sparsity Of Train matrix : 99.8634167407356 %

### 3.3.6.2 Creating sparse matrix from test data frame

```
In [25]: start = datetime.now()
if os.path.isfile('test_sparse_matrix.npz'):
    print("It is present in your pwd, getting it from disk....")
    # just get it from the disk instead of computing it
    test_sparse_matrix = sparse.load_npz('test_sparse_matrix.npz')
    print("DONE..")
else:
    print("We are creating sparse_matrix from the dataframe..")
    # create sparse_matrix and store it for after usage.
    # csr_matrix(data_values, (row_index, col_index), shape_of_matrix)
    # It should be in such a way that, MATRIX[row, col] = data
    test_sparse_matrix = sparse.csr_matrix((test_df.rating.values, (test_df.user.values,
                                                                    test_df.movie.values)))

    print('Done. It\'s shape is : (user, movie) : ',test_sparse_matrix.shape)
    print('Saving it into disk for furthur usage..')
    # save it into disk
    sparse.save_npz("test_sparse_matrix.npz", test_sparse_matrix)
    print('Done..\n')
```

```

print(datetime.now() - start)

It is present in your pwd, getting it from disk....
DONE..
0:00:00.503082

```

### The Sparsity of Test data Matrix

```

In [45]: us,mv = test_sparse_matrix.shape
elem = test_sparse_matrix.count_nonzero()

print("Sparsity Of Test matrix : {} %".format( (1-(elem/(us*mv))) * 100 ) )

Sparsity Of Test matrix : 99.9658541851839 %

```

## 3.3.7 Finding Global average of all movie ratings, Average rating per user, and Average rating per movie

```

In [46]: # get the user averages in dictionary (key: user_id/movie_id, value: avg rating)

def get_average_ratings(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_ratings = 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_ratings = is_rated.sum(axis=ax).A1

    # max_user and max_movie ids in sparse matrix
    u,m = sparse_matrix.shape
    # create a dictionary of users and their average ratings..
    average_ratings = { i : sum_of_ratings[i]/no_of_ratings[i]
                        for i in range(u if of_users else m)
                        if no_of_ratings[i] !=0}

    # return that dictionary of average ratings
    return average_ratings

```

```

In [59]: print((train_sparse_matrix!=0)[2,0])
False

```

### 3.3.7.1 finding global average of all movie ratings

```

In [60]: 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

```

```

Out[60]: {'global': 3.55792343987618}

```

### 3.3.7.2 finding average rating per user

```

In [61]: train_averages['user'] = get_average_ratings(train_sparse_matrix, of_users=True)
print('\nAverage rating of user 10 :',train_averages['user'][10])

Average rating of user 10 : 3.3846153846153846

```

### 3.3.7.3 finding average rating per movie

```

In [62]: train_averages['movie'] = get_average_ratings(train_sparse_matrix, of_users=False)
print('\n Average rating of movie 15 :',train_averages['movie'][15])

Average rating of movie 15 : 3.2844444444444445

```

```

In [69]: kill=list(train_averages['user'].values())

```

```
print(type(kill))
```

```
<class 'list'>
```

### 3.3.7.4 PDF's & CDF's of Avg.Ratings of Users & Movies (In Train Data)

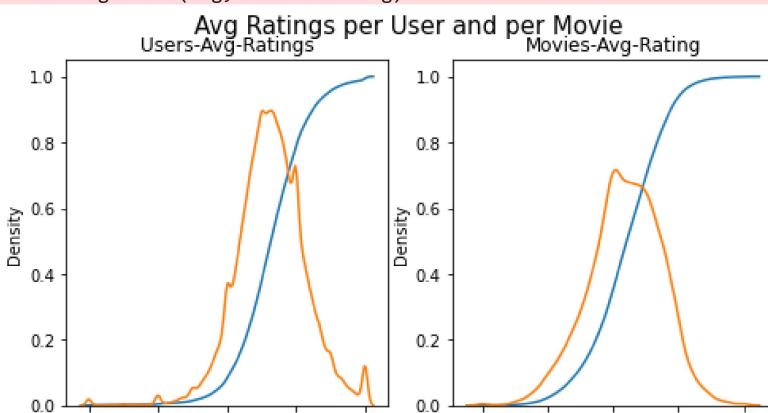
```
In [70]: start = datetime.now()
# draw pdfs for average rating per user and average
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=plt.figaspect(.5))
fig.suptitle('Avg Ratings per User and per Movie', fontsize=15)

ax1.set_title('Users-Avg-Ratings')
# get the list of average user ratings from the averages dictionary..
user_averages = list(train_averages['user'].values())
sns.distplot(user_averages, ax=ax1, hist=False,
             kde_kws=dict(cumulative=True), label='Cdf')
sns.distplot(user_averages, ax=ax1, hist=False, label='Pdf')

ax2.set_title('Movies-Avg-Rating')
# get the list of movie_average_ratings from the dictionary..
movie_averages = list(train_averages['movie'].values())
sns.distplot(movie_averages, ax=ax2, hist=False,
             kde_kws=dict(cumulative=True), label='Cdf')
sns.distplot(movie_averages, ax=ax2, hist=False, label='Pdf')

plt.show()
print(datetime.now() - start)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).
    warnings.warn(msg, FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).
    warnings.warn(msg, FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).
    warnings.warn(msg, FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).
    warnings.warn(msg, FutureWarning)
```



```
0:00:05.145068
```

### 3.3.8 Cold Start problem

#### 3.3.8.1 Cold Start problem with Users

```
In [71]: total_users = len(np.unique(df.user))
users_train = len(train_averages['user'])
new_users = total_users - users_train
```

```

print("\nTotal number of Users : ", total_users)
print("\nNumber of Users in Train data : ", users_train)
print("\nNo of Users that didn't appear in train data: {}({} %) \n ".format(new_users,
                                                                           np.round((new_users/total_users)*1

```

Total number of Users : 405041

Number of Users in Train data : 344884

No of Users that didn't appear in train data: 60157(14.85 %)

We might have to handle **new users ( 75148 )** who didn't appear in train data.

### 3.3.8.2 Cold Start problem with Movies

```

In [72]: total_movies = len(np.unique(df.movie))
movies_train = len(train_averages['movie'])
new_movies = total_movies - movies_train

print("\nTotal number of Movies : ", total_movies)
print("\nNumber of Users in Train data : ", movies_train)
print("\nNo of Movies that didn't appear in train data: {}({} %) \n ".format(new_movies,
                                                                           np.round((new_movies/total_movies)*1

```

Total number of Movies : 17424

Number of Users in Train data : 16727

No of Movies that didn't appear in train data: 697(4.0 %)

We might have to handle **346 movies** (small comparatively) in test data

## 3.4 Computing Similarity matrices

### 3.4.1 Computing User-User Similarity matrix

1. Calculating User User Similarity\_Matrix is **not very easy**(unless you have huge Computing Power and lots of time) because of number of. usersbeing lare.

- You can try if you want to. Your system could crash or the program stops with **Memory Error**

#### 3.4.1.1 Trying with all dimensions (17k dimensions per user)

```

In [73]: from sklearn.metrics.pairwise import cosine_similarity

def compute_user_similarity(sparse_matrix, compute_for_few=False, top = 100, verbose=False, verb_for_n_row
                           draw_time_taken=True):
    no_of_users, _ = sparse_matrix.shape
    # get the indices of non zero rows(users) from our sparse matrix
    row_ind, col_ind = sparse_matrix.nonzero()
    row_ind = sorted(set(row_ind)) # we don't have to
    time_taken = list() # time taken for finding similar users for an user..

    # we create rows, cols, and data lists.., which can be used to create sparse matrices
    rows, cols, data = list(), list(), list()
    if verbose: print("Computing top",top,"similarities for each user..")

    start = datetime.now()
    temp = 0

    for row in row_ind[:top] if compute_for_few else row_ind:
        temp = temp+1
        prev = datetime.now()

        # get the similarity row for this user with all other users
        sim = cosine_similarity(sparse_matrix.getrow(row), sparse_matrix).ravel()

```

```

# We will get only the top 'top' most similar users and ignore rest of them..
top_sim_ind = sim.argsort()[-top:]
top_sim_val = sim[top_sim_ind]

# add them to our rows, cols and data
rows.extend([row]*top)
cols.extend(top_sim_ind)
data.extend(top_sim_val)
time_taken.append(datetime.now().timestamp() - prev.timestamp())
if verbose:
    if temp%verb_for_n_rows == 0:
        print("computing done for {} users [ time elapsed : {} ]"
              .format(temp, datetime.now()-start))

# Lets create sparse matrix out of these and return it
if verbose: print('Creating Sparse matrix from the computed similarities')
#return rows, cols, data

if draw_time_taken:
    plt.plot(time_taken, label = 'time taken for each user')
    plt.plot(np.cumsum(time_taken), label='Total time')
    plt.legend(loc='best')
    plt.xlabel('User')
    plt.ylabel('Time (seconds)')
    plt.show()

return sparse.csr_matrix((data, (rows, cols)), shape=(no_of_users, no_of_users)), time_taken

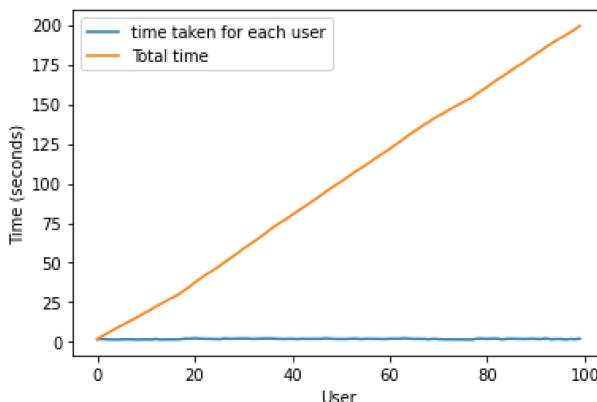
```

```

In [74]: start = datetime.now()
u_u_sim_sparse, _ = compute_user_similarity(train_sparse_matrix, compute_for_few=True, top = 100,
                                             verbose=True)
print("-"*100)
print("Time taken :",datetime.now()-start)

```

Computing top 100 similarities for each user..  
computing done for 20 users [ time elapsed : 0:00:35.119788 ]  
computing done for 40 users [ time elapsed : 0:01:18.191597 ]  
computing done for 60 users [ time elapsed : 0:01:59.628400 ]  
computing done for 80 users [ time elapsed : 0:02:38.888576 ]  
computing done for 100 users [ time elapsed : 0:03:19.346976 ]  
Creating Sparse matrix from the computed similarities



Time taken : 0:03:23.880970

### 3.4.1.2 Trying with reduced dimensions (Using TruncatedSVD for dimensionality reduction of user vector)

- We have **405,041 users** in our training set and computing similarities between them..( **17K dimensional vector.**) is time consuming..
- From above plot, It took roughly **8.88 sec** for computing similar users for **one user**
- We have **405,041 users** with us in training set.
- $405041 \times 8.88 = 3596764.08 \text{ sec} = 59946.068 \text{ min} = 999.101133333 \text{ hours} = 41.629213889 \text{ days...}$

- Even if we run on 4 cores parallelly (a typical system now a days), It will still take almost **10 and 1/2** days.

IDEA: Instead, we will try to reduce the dimentions using SVD, so that **it might** speed up the process...

```
In [ ]: from datetime import datetime
from sklearn.decomposition import TruncatedSVD

start = datetime.now()

# initilaize the algorithm with some parameters..
# All of them are default except n_components. n_itr is for Randomized SVD solver.
netflix_svd = TruncatedSVD(n_components=500, algorithm='randomized', random_state=15)
trunc_svd = netflix_svd.fit_transform(train_sparse_matrix)

print(datetime.now()-start)
```

0:29:07.069783

Here,

- $\Sigma \leftarrow (\text{netflix\_svd}.\text{singular\_values}_\text{})$
- $V^T \leftarrow (\text{netflix\_svd}.\text{components}_\text{})$
- $U$  is not returned. instead **Projection\_of\_X** onto the new vectorspace is returned.
- It uses **randomized svd** internally, which returns **All 3 of them saperately**. Use that instead..

```
In [ ]: expl_var = np.cumsum(netflix_svd.explained_variance_ratio_)

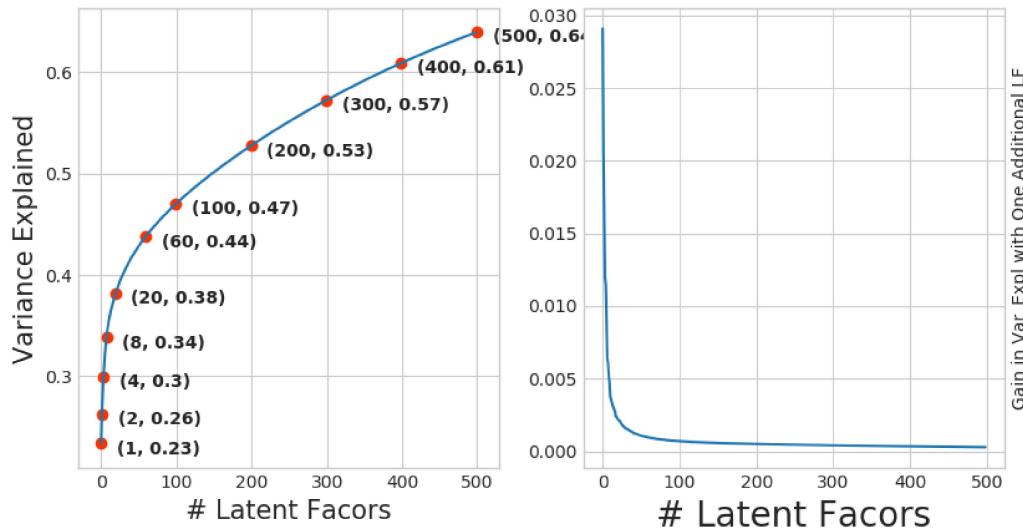
In [ ]: fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=plt.figaspect(.5))

ax1.set_ylabel("Variance Explained", fontsize=15)
ax1.set_xlabel("# Latent Facors", fontsize=15)
ax1.plot(expl_var)
# annotate some (Latentfactors, expl_var) to make it clear
ind = [1, 2, 4, 8, 20, 60, 100, 200, 300, 400, 500]
ax1.scatter(x = [i-1 for i in ind], y = expl_var[[i-1 for i in ind]], c='#ff3300')
for i in ind:
    ax1.annotate(s ="({}, {})".format(i, np.round(expl_var[i-1], 2)), xy=(i-1, expl_var[i-1]),
                 xytext = (i+20, expl_var[i-1] - 0.01), fontweight='bold')

change_in_expl_var = [expl_var[i+1] - expl_var[i] for i in range(len(expl_var)-1)]
ax2.plot(change_in_expl_var)

ax2.set_ylabel("Gain in Var_Expl with One Additional LF", fontsize=10)
ax2.yaxis.set_label_position("right")
ax2.set_xlabel("# Latent Facors", fontsize=20)

plt.show()
```



```
In [ ]: for i in ind:
    print("({}, {})".format(i, np.round(expl_var[i-1], 2)))
```

```
(1, 0.23)
(2, 0.26)
(4, 0.3)
(8, 0.34)
(20, 0.38)
(60, 0.44)
(100, 0.47)
(200, 0.53)
(300, 0.57)
(400, 0.61)
(500, 0.64)
```

I think 500 dimensions is good enough

- By just taking **(20 to 30)** latent factors, explained variance that we could get is **20 %**.
- To take it to **60%**, we have to take **almost 400 latent factors**. It is not fare.
- It basically is the **gain of variance explained**, if we **add one additional latent factor to it**.
- By adding one by one latent factor to it, the gain in explained variance with that addition is decreasing. (Obviously, because they are sorted that way).
- **LHS Graph:**
  - **x** --- ( No of latent factors ),
  - **y** --- ( The variance explained by taking x latent factors)
- More decrease in the line (RHS graph) :
  - We are getting more explained variance than before.
- **Less decrease in that line (RHS graph) :**
  - We are not getting benefitted from adding latent factor further. This is what is shown in the plots.
- **RHS Graph:**
  - **x** --- ( No of latent factors ),
  - **y** --- ( Gain in Expl\_Var by taking one additional latent factor)

```
In [ ]: # Let's project our Original U_M matrix into 500 Dimensional space...
start = datetime.now()
trunc_matrix = train_sparse_matrix.dot(netflix_svd.components_.T)
print(datetime.now() - start)
```

0:00:45.670265

```
In [ ]: type(trunc_matrix), trunc_matrix.shape
```

```
Out[ ]: (numpy.ndarray, (2649430, 500))
```

- Let's convert this to actual sparse matrix and store it for future purposes

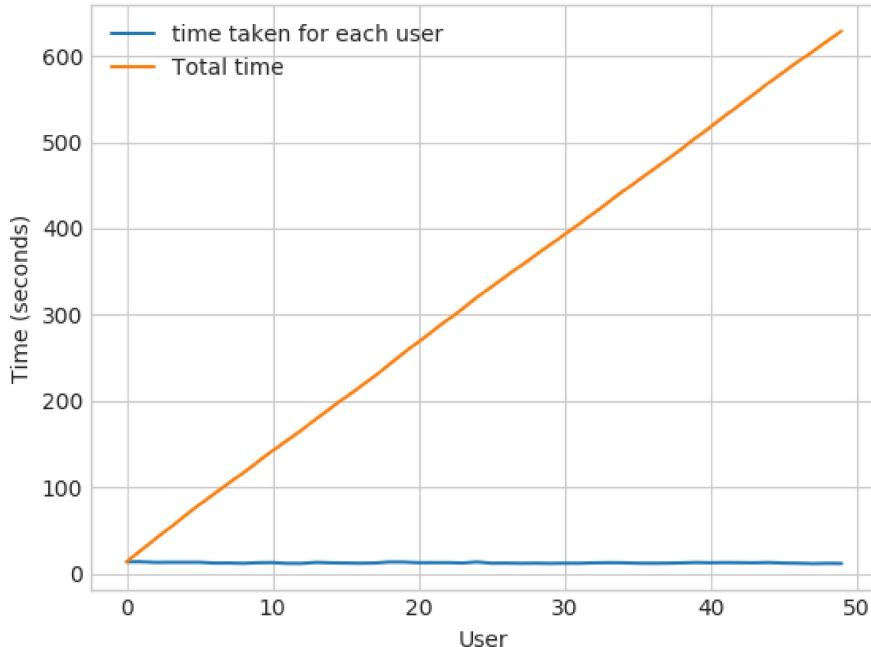
```
In [ ]: if not os.path.isfile('trunc_sparse_matrix.npz'):  
    # create that sparse sparse matrix  
    trunc_sparse_matrix = sparse.csr_matrix(trunc_matrix)  
    # Save this truncated sparse matrix for later usage..  
    sparse.save_npz('trunc_sparse_matrix', trunc_sparse_matrix)  
else:  
    trunc_sparse_matrix = sparse.load_npz('trunc_sparse_matrix.npz')
```

```
In [ ]: trunc_sparse_matrix.shape
```

```
Out[ ]: (2649430, 500)
```

```
In [ ]: start = datetime.now()  
trunc_u_u_sim_matrix, _ = compute_user_similarity(trunc_sparse_matrix, compute_for_few=True, top=50, verbose_for_n_rows=10)  
print("-"*50)  
print("time:", datetime.now()-start)
```

```
Computing top 50 similarities for each user..  
computing done for 10 users [ time elapsed : 0:02:09.746324 ]  
computing done for 20 users [ time elapsed : 0:04:16.017768 ]  
computing done for 30 users [ time elapsed : 0:06:20.861163 ]  
computing done for 40 users [ time elapsed : 0:08:24.933316 ]  
computing done for 50 users [ time elapsed : 0:10:28.861485 ]  
Creating Sparse matrix from the computed similarities
```



```
-----  
time: 0:10:52.658092
```

: This is taking more time for each user than Original one.

- from above plot, It took almost **12.18** for computing similar users for **one user**
- We have **405041 users** with us in training set.
- $405041 \times 12.18 = 4933399.38 \text{ sec} = 82223.323 \text{ min} = 1370.388716667 \text{ hours} = 57.0995$

- Even we run on 4 cores parallelly (a typical system now a days), It will still take almost (14 - 15) days.

- Why did this happen...??**

- Just think about it. It's not that difficult.

-----*( sparse & dense.....get it ?? )-----*

### Is there any other way to compute user user similarity..??

-An alternative is to compute similar users for a particular user, whenever required (**ie., Run time**) - We maintain a binary Vector for users, which tells us whether we already computed or not.. - **If not** : - Compute top (let's just say, 1000) most similar users for this given user, and add this to our datastructure, so that we can just access it(similar users) without recomputing it again. - **If It is already Computed**: - Just get it directly from our datastructure, which has that information. - In production time, We might have to recompute similarities, if it is computed a long time ago. Because user preferences changes over time. If we could maintain some kind of Timer, which when expires, we have to update it (recompute it). - **Which datastructure to use**: - It is purely implementation dependant. - One simple method is to maintain a **Dictionary Of Dictionaries**. - **key : userid** - **value**: Again a dictionary - **key : Similar User** - **value**: *Similarity Value*

### 3.4.2 Computing Movie-Movie Similarity matrix

```
In [5]: from datetime import datetime
from scipy import sparse
from sklearn.metrics.pairwise import cosine_similarity

start = datetime.now()
if not os.path.isfile('m_m_sim_sparse.npz'):
    print("It seems you don't have that file. Computing movie_movie similarity...")
    start = datetime.now()
    m_m_sim_sparse = cosine_similarity(X=train_sparse_matrix.T, dense_output=False)
    print("Done...")
    # store this sparse matrix in disk before using it. For future purposes.
    print("Saving it to disk without the need of re-computing it again.. ")
    sparse.save_npz("m_m_sim_sparse.npz", m_m_sim_sparse)
    print("Done...")
else:
    print("It is there, We will get it.")
    m_m_sim_sparse = sparse.load_npz("m_m_sim_sparse.npz")
    print("Done ...")

print("It's a ",m_m_sim_sparse.shape," dimensional matrix")

print(datetime.now() - start)
```

It is there, We will get it.  
 Done ...  
 It's a (17771, 17771) dimensional matrix  
 0:00:13.562191

```
In [6]: m_m_sim_sparse.shape
```

```
Out[6]: (17771, 17771)
```

- Even though we have similarity measure of each movie, with all other movies, We generally don't care much about least similar movies.
- Most of the times, only top\_xxx similar items matters. It may be 10 or 100.
- We take only those top similar movie ratings and store them in a saperate dictionary.

```
In [8]: movie_ids = np.unique(m_m_sim_sparse.nonzero()[1])
```

```
In [9]: start = datetime.now()
similar_movies = dict()
for movie in movie_ids:
```

```

# get the top similar movies and store them in the dictionary
sim_movies = m_m_sim_sparse[movie].toarray().ravel().argsort()[:-1][1:]
similar_movies[movie] = sim_movies[:100]
print(datetime.now() - start)

# just testing similar movies for movie_15
similar_movies[15]

```

0:00:22.054960

```

Out[9]: array([ 8013,  6124,  8279, 16528, 12049,  3755,  5927, 13105,  4424,
    15144,  590,  9071, 10193,  3973, 15571, 15054,  4549,  4818,
    7372,  7950,  9584,  6116,  6349,  5370, 16402, 14059, 16455,
    9376,  4706,  9802, 17139,  6292,  8934,  1596,  7428, 14920,
   17183,  1253,  9566,  9460, 10597,  2543,  8003,  2187, 11730,
   15390,  1720, 17590,  8988, 11981,  1942,  2818, 17584,  565,
  10788, 17115,  598,  5865, 10199, 13931,  1690,  3338,  8858,
  9688,  9166,  9558, 13013,  4513,  376,  1510,  3706, 12954,
 12762,  9840,  7845,  6410, 11867, 11175,  8873,  7481, 15360,
 2716,  9427,  9969,  9488,  4649,  4376, 16334, 16309,  5871,
 5921,  847,  7859,  5282, 1030, 14696,  8875,  6944,  4486,
 6488], dtype=int64)

```

```

In [21]: jack=m_m_sim_sparse[2].toarray().ravel().argsort()[:-1];
print(jack)

[ 2 11212 4733 ... 8081 8059      0]

```

### 3.4.3 Finding most similar movies using similarity matrix

— Does Similarity really works as the way we expected...? —

*Let's pick some random movie and check for its similar movies....*

```

In [23]: # First Let's load the movie details into soe dataframe..
# movie details are in 'netflix/movie_titles.csv'

movie_titles = pd.read_csv("movie_titles.csv", sep=',', header = None,
                           names=['movie_id', 'year_of_release', 'title'],
                           verbose=True,
                           index_col = 'movie_id', encoding = "ISO-8859-1")

movie_titles.head()

Tokenization took: 3.00 ms
Type conversion took: 10.00 ms
Parser memory cleanup took: 0.00 ms

```

```

Out[23]:      year_of_release          title
movie_id
1           2003.0        Dinosaur Planet
2           2004.0  Isle of Man TT 2004 Review
3           1997.0       Character
4           1994.0  Paula Abdul's Get Up & Dance
5           2004.0  The Rise and Fall of ECW

```

#### Similar Movies for 'Bruce Lee: A Warrior's Journey'

```

In [52]: mv_id = 113

print("\nMovie ---->",movie_titles.loc[mv_id].values[1])

print("\nIt has {} Ratings from users.".format(train_sparse_matrix[:,mv_id].getnnz()))

print("\nWe have {} movies which are similarto this and we will get only top most..".format(m_m_sim_spars

```

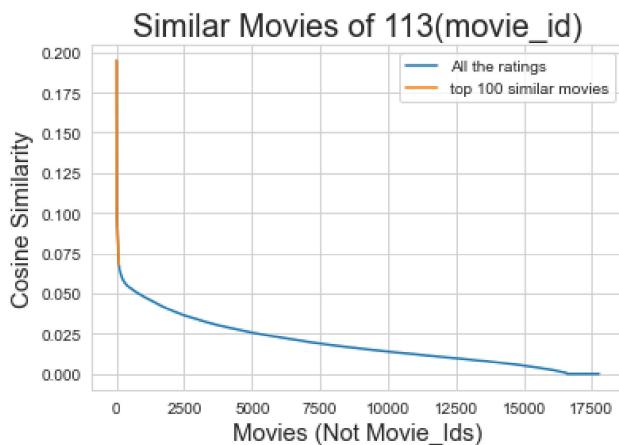
Movie ----> Bruce Lee: A Warrior's Journey

It has 861 Ratings from users.

We have 16575 movies which are similar to this and we will get only top most..

```
In [53]: similarities = m_m_sim_sparse[mv_id].toarray().ravel()
sim_indices = similarities.argsort()[:-1][1:] # It will sort and reverse the array and ignore its similar
# and return its indices(movie_ids)
```

```
In [54]: %matplotlib inline
plt.plot(similarities[sim_indices], label='All the ratings')
plt.plot(similarities[sim_indices[:100]], label='top 100 similar movies')
plt.title("Similar Movies of {}(movie_id)".format(mv_id), fontsize=20)
plt.xlabel("Movies (Not Movie_Ids)", fontsize=15)
plt.ylabel("Cosine Similarity", fontsize=15)
plt.legend()
plt.show()
```



### Top 10 similar movies

```
In [48]: movie_titles.loc[sim_indices[:10]]
```

```
Out[48]:      year_of_release          title
movie_id
11860        1976.0  Bruce Lee: Fists of Fury / Chinese Connection
1300         1973.0           Return of the Dragon
10636        1972.0       The Chinese Connection
14861        2001.0  Bruce Lee's Jeet Kune Do
16994        1973.0 Enter the Dragon: Special Edition
9318         1979.0           The Real Bruce Lee
1123         1993.0  Dragon: The Bruce Lee Story
17520        1974.0  Bruce Lee: A Dragon Story
17091        2002.0 Modern Warriors: The Martial Way
13634        1994.0           Fist of Legend
```

```
In [125... mv_id=2000
```

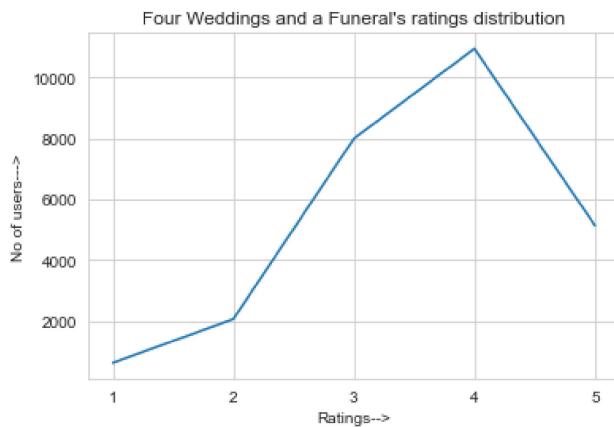
```
dog=np.array(train_sparse_matrix[:,mv_id].todense())
ser=pd.Series(dog.ravel())
list_p=[ser.value_counts()[s] for s in range(1,6)]

plt.plot(pd.Series(list_p,index=[1,2,3,4,5]))
plt.xticks([1,2,3,4,5])
plt.title(str(movie_titles.loc[mv_id].values[1])+'\'s ratings distribution')
```

```

plt.xlabel('Ratings-->')
plt.ylabel('No of users--->')
plt.show()

```



```

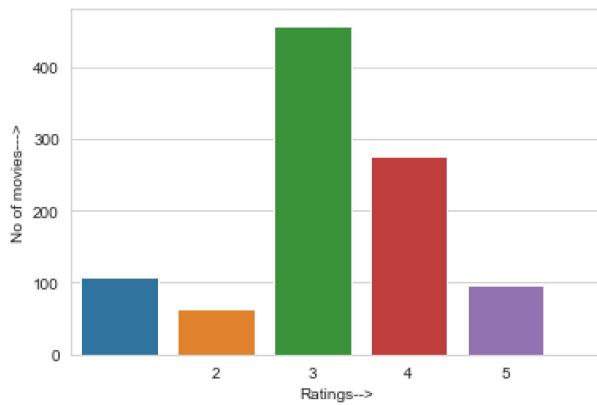
In [156...]
max_ratings=[]
for mv_id in range(1,1000):
    dog=np.array(train_sparse_matrix[:,mv_id].todense())
    ser=pd.Series(dog.ravel())
    list_p=[]
    for s in range(1,6):
        try:
            list_p.append(ser.value_counts()[s])
        except:
            list_p.append(0)
    max_ratings.append(np.argmax(list_p)+1)

```

```

In [165...]
list2=[pd.Series(max_ratings).value_counts()[s] for s in range(1,6)]
sns.barplot(x=[1,2,3,4,5],y=list2)
plt.xticks([1,2,3,4,5])
plt.xlabel('Ratings-->')
plt.ylabel('No of movies--->')
plt.show()

```



Similarly, we can **find similar users** and compare how similar they are.

## 4. Machine Learning Models

```
In [2]: def get_sample_sparse_matrix(sparse_matrix, no_users, no_movies, path, verbose = True):
    """
        It will get it from the ''path'' if it is present or It will create
        and store the sampled sparse matrix in the path specified.
    """

    # get (row, col) and (rating) tuple from sparse_matrix...
    row_ind, col_ind, ratings = sparse.find(sparse_matrix)
    users = np.unique(row_ind)
    movies = np.unique(col_ind)

    print("Original Matrix : (users, movies) -- ({}, {})".format(len(users), len(movies)))
    print("Original Matrix : Ratings -- {}".format(len(ratings)))

    # It just to make sure to get same sample everytime we run this program..
    # and pick without replacement....
    np.random.seed(15)
    sample_users = np.random.choice(users, no_users, replace=False)
    sample_movies = np.random.choice(movies, no_movies, replace=False)
    # get the boolean mask or these sampled_items in originl row/col_inds..
    mask = np.logical_and(np.isin(sample_users, row_ind), np.isin(sample_movies, col_ind))

    sample_sparse_matrix = sparse.csr_matrix((ratings[mask], (row_ind[mask], col_ind[mask])),
                                              shape=(max(sample_users)+1, max(sample_movies)+1))

    if verbose:
        print("Sampled Matrix : (users, movies) -- ({}, {})".format(len(sample_users), len(sample_movies)))
        print("Sampled Matrix : Ratings -- ", format(ratings[mask].shape[0]))

    print('Saving it into disk for furthur usage..')
    # save it into disk
    sparse.save_npz(path, sample_sparse_matrix)
    if verbose:
        print('Done..\n')

    return sample_sparse_matrix
```

### 4.1 Sampling Data

#### 4.1.1 Build sample train data from the train data

```
In [4]: start = datetime.now()
path = "sample_train_sparse_matrix.npz"
if os.path.isfile(path):
    print("It is present in your pwd, getting it from disk....")
    # just get it from the disk instead of computing it
    sample_train_sparse_matrix = sparse.load_npz(path)
    print("DONE..")
else:
    # get 10k users and 1k movies from available data
    train_sparse_matrix = sparse.load_npz('train_sparse_matrix.npz')
    sample_train_sparse_matrix = get_sample_sparse_matrix(train_sparse_matrix, no_users=10000, no_movies=1)

print(datetime.now() - start)
```

```
Original Matrix : (users, movies) -- (344884 16727)
Original Matrix : Ratings -- 64307524
```

```
Sampled Matrix : (users, movies) -- (10000 1000)
Sampled Matrix : Ratings -- 126757
Saving it into disk for furthur usage..
Done..
```

```
0:00:32.463067
```

#### 4.1.2 Build sample test data from the test data

```
In [5]: start = datetime.now()

path = "sample_test_sparse_matrix.npz"
if os.path.isfile(path):
    print("It is present in your pwd, getting it from disk....")
    # just get it from the disk instead of computing it
    sample_test_sparse_matrix = sparse.load_npz(path)
    print("DONE..")
else:
    # get 5k users and 500 movies from available data
    test_sparse_matrix = sparse.load_npz('test_sparse_matrix.npz')
    sample_test_sparse_matrix = get_sample_sparse_matrix(test_sparse_matrix, no_users=5000, no_movies=500,
                                                       print(datetime.now() - start))

Original Matrix : (users, movies) -- (307632 17377)
Original Matrix : Ratings -- 16076881

Sampled Matrix : (users, movies) -- (5000 500)
Sampled Matrix : Ratings -- 7116
Saving it into disk for furthur usage..
Done..

0:00:06.992341
```

```
In [6]: sample_test_sparse_matrix.shape
```

```
Out[6]: (2648402, 17766)
```

```
In [7]: sample_train_sparse_matrix.shape
```

```
Out[7]: (2649380, 17763)
```

## 4.2 Finding Global Average of all movie ratings, Average rating per User, and Average rating per Movie (from sampled train)

```
In [ ]: sample_train_averages = dict()
```

### 4.2.1 Finding Global Average of all movie ratings

```
In [ ]: # get the global average of ratings in our train set.
global_average = sample_train_sparse_matrix.sum()/sample_train_sparse_matrix.count_nonzero()
sample_train_averages['global'] = global_average
sample_train_averages
```

```
Out[ ]: {'global': 3.581679377504138}
```

### 4.2.2 Finding Average rating per User

```
In [ ]: sample_train_averages['user'] = get_average_ratings(sample_train_sparse_matrix, of_users=True)
print('\nAverage rating of user 1515220 :',sample_train_averages['user'][1515220])

Average rating of user 1515220 : 3.9655172413793105
```

### 4.2.3 Finding Average rating per Movie

```
In [ ]: sample_train_averages['movie'] = get_average_ratings(sample_train_sparse_matrix, of_users=False)
print('\n Average rating of movie 15153 :',sample_train_averages['movie'][15153])

AVerage rating of movie 15153 : 2.6458333333333335
```

## 4.3 Featurizing data

```
In [8]: print("\n No of ratings in Our Sampled train matrix is : {}".format(sample_train_sparse_matrix.count_nonzero))
print("\n No of ratings in Our Sampled test matrix is : {}".format(sample_test_sparse_matrix.count_nonzero))
```

```
No of ratings in Our Sampled train matrix is : 126757
```

```
No of ratings in Our Sampled test matrix is : 7116
```

### 4.3.1 Featurizing data for regression problem

#### 4.3.1.1 Featurizing train data

```
In [9]: # get users, movies and ratings from our samples train sparse matrix
sample_train_users, sample_train_movies, sample_train_ratings = sparse.find(sample_train_sparse_matrix)

In [ ]: #####
# It took me almost 10 hours to prepare this train dataset.#
#####
start = datetime.now()
if os.path.isfile('reg_train.csv'):
    print("File already exists you don't have to prepare again... ")
else:
    print('preparing {} tuples for the dataset..\n'.format(len(sample_train_ratings)))
    with open('reg_train.csv', mode='w') as reg_data_file:
        count = 0
        for (user, movie, rating) in zip(sample_train_users, sample_train_movies, sample_train_ratings):
            st = datetime.now()
            # print(user, movie)
            #----- Ratings of "movie" by similar users of "user" -----
            # compute the similar Users of the "user"
            user_sim = cosine_similarity(sample_train_sparse_matrix[user], sample_train_sparse_matrix).ravel()
            top_sim_users = user_sim.argsort()[:-1][1:] # we are ignoring 'The User' from its similar use
            # get the ratings of most similar users for this movie
            top_ratings = sample_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([sample_train_averages['movie'][movie]]*(5 - len(top_sim_users_ra
            #     print(top_sim_users_ratings, end=" ")

            #----- Ratings by "user" to similar movies of "movie" -----
            # compute the similar movies of the "movie"
            movie_sim = cosine_similarity(sample_train_sparse_matrix[:, movie].T, sample_train_sparse_matrix).ravel()
            top_sim_movies = movie_sim.argsort()[:-1][1:] # we are ignoring 'The User' from its similar u
            # get the ratings of most similar movie rated by this user..
            top_ratings = sample_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([sample_train_averages['user'][user]]*(5 - len(top_sim_movies_rati
            #     print(top_sim_movies_ratings, end=" : -- ")

            #-----prepare the row to be stores in a file-----
            row = list()
            row.append(user)
            row.append(movie)
            # Now add the other features to this data...
            row.append(sample_train_averages['global']) # first feature
            # next 5 features are similar_users "movie" ratings
            row.extend(top_sim_users_ratings)
            # next 5 features are "user" ratings for similar_movies
            row.extend(top_sim_movies_ratings)
            # Avg_user rating
            row.append(sample_train_averages['user'][user])
            # Avg_movie rating
            row.append(sample_train_averages['movie'][movie])

            # finalley, The actual Rating of this user-movie pair...
            row.append(rating)
            count = count + 1

            # add rows to the file opened..
            reg_data_file.write(','.join(map(str, row)))
            reg_data_file.write('\n')
            if (count)%10000 == 0:
                # print(','.join(map(str, row)))
                print("Done for {} rows---- {} ".format(count, datetime.now() - start))
```

```

print(datetime.now() - start)

preparing 129286 tuples for the dataset..

Done for 10000 rows---- 0:53:13.974716
Done for 20000 rows---- 1:47:58.228942
Done for 30000 rows---- 2:42:46.963119
Done for 40000 rows---- 3:36:44.807894
Done for 50000 rows---- 4:28:55.311500
Done for 60000 rows---- 5:24:18.493104
Done for 70000 rows---- 6:17:39.669922
Done for 80000 rows---- 7:11:23.970879
Done for 90000 rows---- 8:05:33.787770
Done for 100000 rows---- 9:00:25.463562
Done for 110000 rows---- 9:51:28.530010
Done for 120000 rows---- 10:42:05.382141
11:30:13.699183

```

#### Reading from the file to make a Train\_dataframe

```
In [13]: reg_train = pd.read_csv('reg_train.csv', names = ['user', 'movie', 'GAvg', 'sur1', 'sur2', 'sur3', 'sur4',
                                                    'sur5', 'smr1', 'smr2', 'smr3', 'smr4', 'smr5', 'UAvg',
                                                    'MAvg', 'rat1'])

reg_train.head()
```

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UAvg	MAvg	rat1
0	53406	33	3.581679	4.0	5.0	5.0	4.0	1.0	5.0	2.0	5.0	3.0	1.0	3.370370	4.092437	
1	99540	33	3.581679	5.0	5.0	5.0	4.0	5.0	3.0	4.0	4.0	3.0	5.0	3.555556	4.092437	
2	99865	33	3.581679	5.0	5.0	4.0	5.0	3.0	5.0	4.0	4.0	5.0	4.0	3.714286	4.092437	
3	101620	33	3.581679	2.0	3.0	5.0	5.0	4.0	4.0	3.0	3.0	4.0	5.0	3.584416	4.092437	
4	112974	33	3.581679	5.0	5.0	5.0	5.0	5.0	3.0	5.0	5.0	5.0	3.0	3.750000	4.092437	

- 
- **GAvg**: Average rating of all the 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 this movie.. )
  - **UAvg**: User's Average rating
  - **MAvg**: Average rating of this movie
  - **rating**: Rating of this movie by this user.
- 

#### 4.3.1.2 Featurizing test data

```
In [ ]: # get users, movies and ratings from the Sampled Test
sample_test_users, sample_test_movies, sample_test_ratings = sparse.find(sample_test_sparse_matrix)

In [ ]: sample_train_averages['global']

Out[ ]: 3.581679377504138

In [ ]: start = datetime.now()

if os.path.isfile('reg_test.csv'):
    print("It is already created...")
else:

    print('preparing {} tuples for the dataset..\n'.format(len(sample_test_ratings)))
    with open('reg_test.csv', mode='w') as reg_data_file:
```

```

count = 0
for (user, movie, rating) in zip(sample_test_users, sample_test_movies, sample_test_ratings):
    st = datetime.now()

----- Ratings of "movie" by similar users of "user" -----
#print(user, movie)
try:
    # compute the similar Users of the "user"
    user_sim = cosine_similarity(sample_train_sparse_matrix[user], sample_train_sparse_matrix)
    top_sim_users = user_sim.argsort()[:-1][1:] # we are ignoring 'The User' from its similar
    # get the ratings of most similar users for this movie
    top_ratings = sample_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([sample_train_averages['movie'][movie]]*(5 - len(top_sim_users)))
    # print(top_sim_users_ratings, end="--")

except (IndexError, KeyError):
    # It is a new User or new Movie or there are no ratings for given user for top similar mov
    ##### Cold Start Problem #####
    top_sim_users_ratings.extend([sample_train_averages['global']]*(5 - len(top_sim_users)))
    #print(top_sim_users_ratings)
except:
    print(user, movie)
    # we just want KeyErrors to be resolved. Not every Exception...
    raise

----- Ratings by "user" to similar movies of "movie" -----
try:
    # compute the similar movies of the "movie"
    movie_sim = cosine_similarity(sample_train_sparse_matrix[:, movie].T, sample_train_sparse_matrix)
    top_sim_movies = movie_sim.argsort()[:-1][1:] # we are ignoring 'The User' from its similar
    # get the ratings of most similar movie rated by this user..
    top_ratings = sample_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([sample_train_averages['user'][user]]*(5 - len(top_sim_movies)))
    #print(top_sim_movies_ratings)
except (IndexError, KeyError):
    #print(top_sim_movies_ratings, end=" : -- ")
    top_sim_movies_ratings.extend([sample_train_averages['global']]*(5 - len(top_sim_movies)))
    #print(top_sim_movies_ratings)
except :
    raise

-----prepare the row to be stores in a file-----#
row = list()
# add usser and movie name first
row.append(user)
row.append(movie)
row.append(sample_train_averages['global']) # first feature
#print(row)
# next 5 features are similar_users "movie" ratings
row.extend(top_sim_users_ratings)
#print(row)
# next 5 features are "user" ratings for similar_movies
row.extend(top_sim_movies_ratings)
#print(row)
# Avg_user rating
try:
    row.append(sample_train_averages['user'][user])
except KeyError:
    row.append(sample_train_averages['global'])
except:
    raise
#print(row)
# Avg_movie rating
try:
    row.append(sample_train_averages['movie'][movie])
except KeyError:
    row.append(sample_train_averages['global'])
except:
    raise
#print(row)

```

```

# finalley, The actual Rating of this user-movie pair...
row.append(rating)
#print(row)
count = count + 1

# add rows to the file opened..
reg_data_file.write(','.join(map(str, row)))
#print(','.join(map(str, row)))
reg_data_file.write('\n')
if (count)%1000 == 0:
    #print(','.join(map(str, row)))
    print("Done for {} rows---- {} ".format(count, datetime.now() - start))
print("",datetime.now() - start)

preparing 7333 tuples for the dataset..

```

Done for 1000 rows---- 0:04:29.293783  
 Done for 2000 rows---- 0:08:57.208002  
 Done for 3000 rows---- 0:13:30.333223  
 Done for 4000 rows---- 0:18:04.050813  
 Done for 5000 rows---- 0:22:38.671673  
 Done for 6000 rows---- 0:27:09.697009  
 Done for 7000 rows---- 0:31:41.933568  
 0:33:12.529731

\_Reading from the file to make a test dataframe \_

```
In [10]: reg_test_df = pd.read_csv('reg_test.csv', names = ['user', 'movie', 'GAvg', 'sur1', 'sur2', 'sur3', 'sur4',
                                                               'smr1', 'smr2', 'smr3', 'smr4', 'smr5',
                                                               'UAvg', 'MAvg', 'rating'], header=None)
reg_test_df.head(4)
```

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	sm
0	808635	71	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.5816
1	941866	71	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.5816
2	1737912	71	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.5816
3	1849204	71	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.5816

• **GAvg** : Average rating of all the ratings

• **Similar users rating of this movie:**

- sur1, sur2, sur3, sur4, sur5 ( top 5 similiar users who rated that movie.. )

• **Similar movies rated by this user:**

- smr1, smr2, smr3, smr4, smr5 ( top 5 similiar movies rated by this movie.. )

• **UAvg** : User AVerage rating

• **MAvg** : Average rating of this movie

• **rating** : Rating of this movie by this user.

## 4.3.2 Transforming data for Surprise models

```
In [11]: from surprise import Reader, Dataset
```

### 4.3.2.1 Transforming train data

- We can't give raw data (movie, user, rating) to train the model in Surprise library.
- They have a saperate format for TRAIN and TEST data, which will be useful for training the models like SVD, KNNBaseLineOnly....etc.,in Surprise.

- We can form the trainset from a file, or from a Pandas DataFrame.  
[http://surprise.readthedocs.io/en/stable/getting\\_started.html#load-dom-dataframe-py](http://surprise.readthedocs.io/en/stable/getting_started.html#load-dom-dataframe-py)

```
In [14]: # 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 = Dataset.load_from_df(reg_train[['user', 'movie', 'rating']], reader)  
  
# build the trainset from traindata.., It is of dataset format from surprise library..  
trainset = train_data.build_full_trainset()
```

### 4.3.2.2 Transforming test data

- Testset is just a list of (user, movie, rating) tuples. (Order in the tuple is important)

```
In [15]: testset = list(zip(reg_test_df.user.values, reg_test_df.movie.values, reg_test_df.rating.values))
testset[:3]
```

```
Out[15]: [(808635, 71, 5), (941866, 71, 4), (1737912, 71, 3)]
```

## 4.4 Applying Machine Learning models

- Global dictionary that stores rmse and mape for all the models....

- It stores the metrics in a dictionary of dictionaries

**keys** : model names(string)

```
In [16]: models_evaluation_train = dict()
models_evaluation_test = dict()

models_evaluation_train, models_evaluation_test
```

Out[16]: ( $\{\}$ ,  $\{\}$ )

## Utility functions for running regression models

```
In [17]: # to get rmse and mape given actual and predicted ratings..
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

#####
#####
#####
def run_xgboost(algo, x_train, y_train, x_test, y_test, verbose=True):
    """
    It will return train_results and test_results
    """

    # dictionaries for storing train and test results
    train_results = dict()
    test_results = dict()

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

```

# from the trained model, get the predictions....
print('Evaluating the model with TRAIN data...')
start = datetime.now()
y_train_pred = algo.predict(x_train)
# get the rmse and mape of train data...
rmse_train, mape_train = get_error_metrics(y_train.values, y_train_pred)

# store the results in train_results dictionary..
train_results = {'rmse': rmse_train,
                 'mape' : mape_train,
                 'predictions' : y_train_pred}

#####
# get the test data predictions and compute rmse and mape
print('Evaluating Test data')
y_test_pred = algo.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}

if verbose:
    print('\nTEST DATA')
    print('-'*30)
    print('RMSE : ', rmse_test)
    print('MAPE : ', mape_test)

# return these train and test results...
return train_results, test_results

```

## Utility functions for Surprise modes

```

In [18]: # it is just to makesure that all of our algorithms should produce same results
# everytime they run...

my_seed = 15
random.seed(my_seed)
np.random.seed(my_seed)

#####
# get (actual_list , predicted_list) ratings given List
# of predictions (prediction is a class in Surprise).
#####
def get_ratings(predictions):
    actual = np.array([pred.r_ui for pred in predictions])
    pred = np.array([pred.est for pred in predictions])

    return actual, pred

#####
# get 'rmse' and 'mape' , given List of prediction objects
#####
def get_errors(predictions, print_them=False):

    actual, pred = get_ratings(predictions)
    rmse = np.sqrt(np.mean((pred - actual)**2))
    mape = np.mean(np.abs(pred - actual)/actual)

    return rmse, mape*100

#####
# It will return predicted ratings, rmse and mape of both train and test data #
#####
def run_surprise(algo, trainset, testset, verbose=True):
    ...
    return train_dict, test_dict

    It returns two dictionaries, one for train and the other is for test
    Each of them have 3 key-value pairs, which specify 'rmse', 'mape', and 'predicted ratings'.
    ...
    start = datetime.now()
    # dictionaries that stores metrics for train and test..

```

```

train = dict()
test = dict()

# train the algorithm with the trainset
st = datetime.now()
print('Training the model...')
algo.fit(trainset)
print('Done. time taken : {} \n'.format(datetime.now()-st))

# ----- Evaluating train data-----#
st = datetime.now()
print('Evaluating the model with train data..')
# get the train predictions (list of prediction class inside Surprise)
train_preds = algo.test(trainset.build_testset())
# get predicted ratings from the train predictions..
train_actual_ratings, train_pred_ratings = get_ratings(train_preds)
# get 'rmse' and 'mape' from the train predictions.
train_rmse, train_mape = get_errors(train_preds)
print('time taken : {}'.format(datetime.now()-st))

if verbose:
    print('*'*15)
    print('Train Data')
    print('*'*15)
    print("RMSE : {}\nMAPE : {}".format(train_rmse, train_mape))

#store them in the train dictionary
if verbose:
    print('adding train results in the dictionary..')
train['rmse'] = train_rmse
train['mape'] = train_mape
train['predictions'] = train_pred_ratings

#----- Evaluating Test data-----#
st = datetime.now()
print('\nEvaluating for test data...')
# get the predictions( list of prediction classes) of test data
test_preds = algo.test(testset)
# get the predicted ratings from the list of predictions
test_actual_ratings, test_pred_ratings = get_ratings(test_preds)
# get error metrics from the predicted and actual ratings
test_rmse, test_mape = get_errors(test_preds)
print('time taken : {}'.format(datetime.now()-st))

if verbose:
    print('*'*15)
    print('Test Data')
    print('*'*15)
    print("RMSE : {}\nMAPE : {}".format(test_rmse, test_mape))
# store them in test dictionary
if verbose:
    print('storing the test results in test dictionary...')
test['rmse'] = test_rmse
test['mape'] = test_mape
test['predictions'] = test_pred_ratings

print('\n'+ '*'*45)
print('Total time taken to run this algorithm :', datetime.now() - start)

# return two dictionaries train and test
return train, test

```

#### 4.4.1 XGBoost with initial 13 features

In [19]: `import xgboost as xgb`

In [21]: `%matplotlib inline`

```

# prepare Train data
x_train = reg_train.drop(['user','movie','rating'], axis=1)
y_train = reg_train['rating']

# Prepare Test data
x_test = reg_test_df.drop(['user','movie','rating'], axis=1)

```

```

y_test = reg_test_df['rating']

# initialize Our first XGBoost model...
first_xgb = xgb.XGBRegressor(silent=False, n_jobs=13, random_state=15, n_estimators=100)
train_results, test_results = run_xgboost(first_xgb, x_train, y_train, x_test, y_test)

# store the results in models_evaluations dictionaries
models_evaluation_train['first_algo'] = train_results
models_evaluation_test['first_algo'] = test_results

xgb.plot_importance(first_xgb)
plt.show()

```

Training the model..

```
[19:31:13] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/learner.cc:573:
Parameters: { "silent" } might not be used.
```

This may not be accurate due to some parameters are only used in language bindings but passed down to XGBoost core. Or some parameters are not used but slip through this verification. Please open an issue if you find above cases.

Done. Time taken : 0:00:01.515979

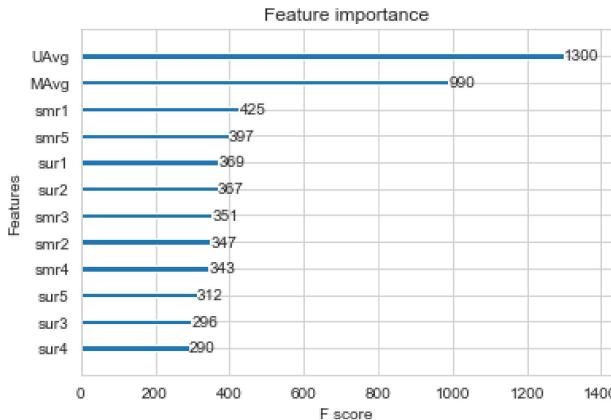
Done

Evaluating the model with TRAIN data...

Evaluating Test data

TEST DATA

```
-----  
RMSE : 1.1016770831525975  
MAPE : 33.379025364595144
```



#### 4.4.2 Surprise BaselineModel

\_Predicted\_rating : ( baseline prediction ) \_

- [http://surprise.readthedocs.io/en/stable/basic\\_algorithms.html#surprise.prediction\\_algorithms.b](http://surprise.readthedocs.io/en/stable/basic_algorithms.html#surprise.prediction_algorithms.b)

$$\hat{r}_{ui} = b_{ui} = \mu + b_u + b_i$$

- $\mu$  : Average of all trainings in training data.
- $b_u$  : User bias
- $b_i$  : Item bias (movie biases)

\_Optimization function ( Least Squares Problem ) \_

- [http://surprise.readthedocs.io/en/stable/prediction\\_algorithms.html#baselines-estimates-configuration](http://surprise.readthedocs.io/en/stable/prediction_algorithms.html#baselines-estimates-configuration)

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - (\mu + b_u + b_i))^2 + \lambda (b_u^2 + b_i^2) . \text{ [minimize } b_u, b_i]$$

```
In [22]: from surprise import BaselineOnly

In [23]: # options are to specify..., how to compute those user and item biases
bsl_options = {'method': 'sgd',
              'learning_rate': .001
              }
bsl_algo = BaselineOnly(bsl_options=bsl_options)
# run this algorithm..., It will return the train and test results..
bsl_train_results, bsl_test_results = run_surprise(bsl_algo, trainset, testset, verbose=True)

# Just store these error metrics in our models_evaluation datastructure
models_evaluation_train['bsl_algo'] = bsl_train_results
models_evaluation_test['bsl_algo'] = bsl_test_results

Training the model...
Estimating biases using sgd...
Done. time taken : 0:00:00.774815

Evaluating the model with train data..
time taken : 0:00:00.645447
-----
Train Data
-----
RMSE : 0.9347153928678286

MAPE : 29.389572652358183

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:00.036008
-----
Test Data
-----
RMSE : 1.0730330260516174

MAPE : 35.04995544572911

storing the test results in test dictionary..

-----
Total time taken to run this algorithm : 0:00:01.458271
```

#### 4.4.3 XGBoost with initial 13 features + Surprise Baseline predictor

##### Updating Train Data

```
In [24]: # add our baseline_predicted value as our feature..
reg_train['bslpr'] = models_evaluation_train['bsl_algo']['predictions']
reg_train.head(2)
```

Out[24]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UAvg	MAvg	rating
<b>0</b>	53406	33	3.581679	4.0	5.0	5.0	4.0	1.0	5.0	2.0	5.0	3.0	1.0	3.370370	4.092437	
<b>1</b>	99540	33	3.581679	5.0	5.0	5.0	4.0	5.0	3.0	4.0	4.0	3.0	5.0	3.555556	4.092437	

##### Updating Test Data

```
In [25]: # add that baseline predicted ratings with Surprise to the test data as well
reg_test_df['bslpr'] = models_evaluation_test['bsl_algo']['predictions']

reg_test_df.head(2)
```

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr
0	808635	71	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.58167
1	941866	71	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.58167

```
In [30]: # prepare train data
x_train = reg_train.drop(['user', 'movie', 'rating'], axis=1)
y_train = reg_train['rating']

# Prepare Test data
x_test = reg_test_df.drop(['user', 'movie', 'rating'], axis=1)
y_test = reg_test_df['rating']

# initialize Our first XGBoost model...
xgb_bsl = xgb.XGBRegressor(silent=False, n_jobs=13, random_state=15, n_estimators=100)
train_results, test_results = run_xgboost(xgb_bsl, x_train, y_train, x_test, y_test)

# store the results in models_evaluations dictionaries
models_evaluation_train['xgb_bsl'] = train_results
models_evaluation_test['xgb_bsl'] = test_results

xgb.plot_importance(xgb_bsl)
plt.show()
```

Training the model..

```
[20:49:30] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/learner.cc:573:
Parameters: { "silent" } might not be used.
```

This may not be accurate due to some parameters are only used in language bindings but passed down to XGBoost core. Or some parameters are not used but slip through this verification. Please open an issue if you find above cases.

Done. Time taken : 0:00:03.581609

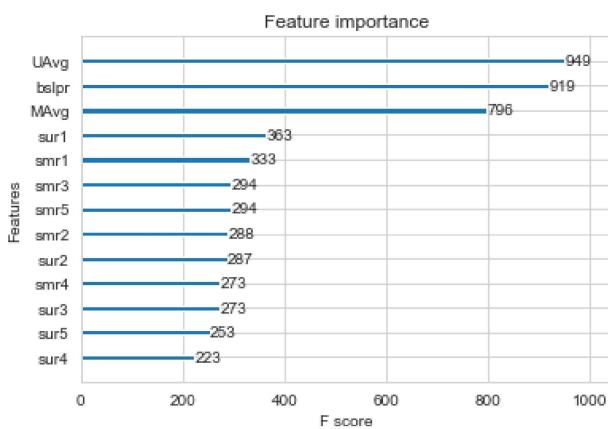
Done

Evaluating the model with TRAIN data...
Evaluating Test data

TEST DATA

```
-----
```

RMSE : 1.1000463052939882  
MAPE : 33.440093738533456



#### 4.4.4 Surprise KNNBaseline predictor

- KNN BASELINE
  - [http://surprise.readthedocs.io/en/stable/knn\\_inspired.html#surprise.prediction\\_algorithms.KNNBaseline](http://surprise.readthedocs.io/en/stable/knn_inspired.html#surprise.prediction_algorithms.KNNBaseline)
- PEARSON\_BASELINE SIMILARITY
  - [http://surprise.readthedocs.io/en/stable/similarities.html#surprise.similarities.pearson\\_baseline](http://surprise.readthedocs.io/en/stable/similarities.html#surprise.similarities.pearson_baseline)

- SHRINKAGE
  - 2.2 Neighborhood Models in <http://courses.ischool.berkeley.edu/i290-dm/s11/SECURE/a1-koren.pdf>

- **predicted Rating** : ( \_\_\_ based on User-User similarity \_\_\_ )

$$\hat{r}_{ui} = b_{ui} + \frac{\sum_{v \in N_i^k(u)} \text{sim}(u, v) \cdot (r_{vi} - b_{vi})}{\sum_{v \in N_i^k(u)} \text{sim}(u, v)}$$

- $b_{ui}$  - Baseline prediction of (user,movie) rating

- $N_i^k(u)$  - Set of K similar users (neighbours) of user (u) who rated movie(i)

- $\text{sim}(u, v)$  - Similarity between users u and v

- Generally, it will be cosine similarity or Pearson correlation coefficient.
- But we use **shrunk Pearson-baseline correlation coefficient**, which is based on the pearsonBaseline similarity (we take base line predictions instead of mean rating of user/item)

- \_\_\_ Predicted rating \_\_\_ ( based on Item Item similarity ):

$$\hat{r}_{ui} = b_{ui} + \frac{\sum_{j \in N_u^k(i)} \text{sim}(i, j) \cdot (r_{uj} - b_{uj})}{\sum_{j \in N_u^k(i)} \text{sim}(i, j)}$$

- \_\_\_Notations follows same as above (user user based predicted rating ) \_\_\_

#### 4.4.4.1 Surprise KNNBaseline with user user similarities

```
In [27]: from surprise import KNNBaseline
```

```
In [28]: # we specify , how to compute similarities and what to consider with sim_options to our algorithm
sim_options = {'user_based' : True,
               'name': 'pearson_baseline',
               'shrinkage': 100,
               'min_support': 2
             }
# we keep other parameters like regularization parameter and Learning_rate as default values.
bsl_options = {'method': 'sgd'}

knn_bsl_u = KNNBaseline(k=40, sim_options = sim_options, bsl_options = bsl_options)
knn_bsl_u_train_results, knn_bsl_u_test_results = run_surprise(knn_bsl_u, trainset, testset, verbose=True)

# Just store these error metrics in our models_evaluation datastructure
models_evaluation_train['knn_bsl_u'] = knn_bsl_u_train_results
models_evaluation_test['knn_bsl_u'] = knn_bsl_u_test_results
```

```

Training the model...
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Done. time taken : 0:00:25.839115

Evaluating the model with train data..
time taken : 0:01:01.236935
-----
Train Data
-----
RMSE : 0.33642097416508826

MAPE : 9.145093375416348

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:00.040009
-----
Test Data
-----
RMSE : 1.0726493739667242

MAPE : 35.02094499698424

storing the test results in test dictionary..

-----
Total time taken to run this algorithm : 0:01:27.116891

```

#### 4.4.4.2 Surprise KNNBaseline with movie movie similarities

```

In [31]: # we specify , how to compute similarities and what to consider with sim_options to our algorithm

# 'user_based' : False => this considers the similarities of movies instead of users

sim_options = {'user_based' : False,
               'name': 'pearson_baseline',
               'shrinkage': 100,
               'min_support': 2
             }
# we keep other parameters like regularization parameter and Learning_rate as default values.
bsl_options = {'method': 'sgd'}

knn_bsl_m = KNNBaseline(k=40, sim_options = sim_options, bsl_options = bsl_options)

knn_bsl_m_train_results, knn_bsl_m_test_results = run_surprise(knn_bsl_m, trainset, testset, verbose=True)

# Just store these error metrics in our models_evaluation datastructure
models_evaluation_train['knn_bsl_m'] = knn_bsl_m_train_results
models_evaluation_test['knn_bsl_m'] = knn_bsl_m_test_results

```

```
Training the model...
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Done. time taken : 0:00:01.086416
```

Evaluating the model with train data..  
time taken : 0:00:05.180025

## Train Data

RMSE : 0.32584796251610554

MAPE : 8.447062581998374

adding train results in the dictionary..

Evaluating for test data...  
time taken : 0:00:00.042010

## Test Data

RMSE : 1.072758832653683

MAPE : 35.02269653015042

storing the test results in test dictionary...

Total time taken to run this algorithm : 0:00:06.308451

#### 4.4.5 XGBoost with initial 13 features + Surprise Baseline predictor + KNNBaseline predictor

- ○ First we will run XGBoost with predictions from both KNN's ( that uses User\_User and Item\_Item similarities along with our previous features.
  - ○ Then we will run XGBoost with just predictions from both knn models and predictions from our baseline model.

## Preparing Train data

```
In [32]: # add the predicted values from both knns to this dataframe  
reg_train['knn_bsl_u'] = models_evaluation_train['knn_bsl_u']['predictions']  
reg_train['knn_bsl_m'] = models_evaluation_train['knn_bsl_m']['predictions']  
  
reg_train.head(2)
```

Out[32]:	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UAvg	MAvg	ratin
<b>0</b>	53406	33	3.581679	4.0	5.0	5.0	4.0	1.0	5.0	2.0	5.0	3.0	1.0	3.370370	4.092437	
<b>1</b>	99540	33	3.581679	5.0	5.0	5.0	4.0	5.0	3.0	4.0	4.0	3.0	5.0	3.555556	4.092437	

## Preparing Test data

```
In [33]: reg_test_df['knn_bsl_u'] = models_evaluation_test['knn_bsl_u']['predictions']
reg_test_df['knn_bsl_m'] = models_evaluation_test['knn_bsl_m']['predictions']

reg_test_df.head(2)
```

```
In [34]: # prepare the train data....
x_train = reg_train.drop(['user', 'movie', 'rating'], axis=1)
y_train = reg_train['rating']

# prepare the train data....
x_test = reg_test_df.drop(['user','movie','rating'], axis=1)
y_test = reg_test_df['rating']

# declare the model
xgb_knn_bsl = xgb.XGBRegressor(n_jobs=10, random_state=15)
train_results, test_results = run_xgboost(xgb_knn_bsl, x_train, y_train, x_test, y_test)

# store the results in models_evaluations dictionaries
models_evaluation_train['xgb_knn_bsl'] = train_results
models_evaluation_test['xgb_knn_bsl'] = test_results

xgb.plot_importance(xgb_knn_bsl)
plt.show()
```

Training the model..

Done. Time taken : 0:00:04.400183

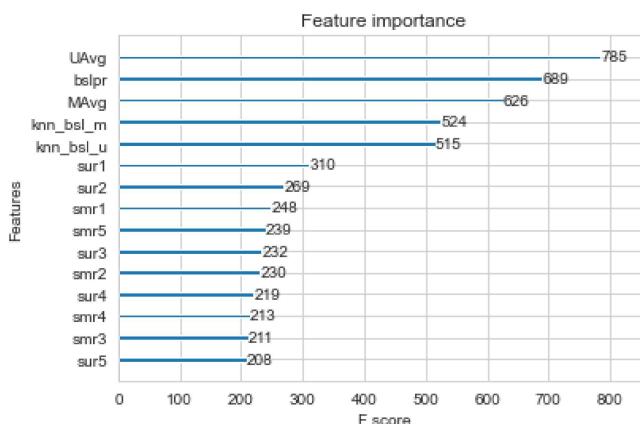
Done

Evaluating the model with TRAIN data...

Evaluating Test data

TEST DATA

-----  
RMSE : 1.0887576141058135  
MAPE : 33.83328394123822



## 4.4.6 Matrix Factorization Techniques

### 4.4.6.1 SVD Matrix Factorization User Movie interactions

```
In [35]: from surprise import SVD
```

[http://surprise.readthedocs.io/en/stable/matrix\\_factorization.html#surprise.prediction\\_algorithms.matrix\\_factorization.SVD](http://surprise.readthedocs.io/en/stable/matrix_factorization.html#surprise.prediction_algorithms.matrix_factorization.SVD)

  Predicted Rating : \_

- $\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$ 
  - $q_i$  - Representation of item(movie) in latent factor space
  - $p_u$  - Representation of user in new latent factor space

- A BASIC MATRIX FACTORIZATION MODEL in [https://datajobs.com/data-science-repo/Recommender-Systems-\[Netflix\].pdf](https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf)

## Optimization problem with user item interactions and regularization (to avoid overfitting)

- $\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda (b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2)$

```
In [36]: # initialize the model
svd = SVD(n_factors=100, biased=True, random_state=15, verbose=True)
svd_train_results, svd_test_results = run_surprise(svd, trainset, testset, verbose=True)

# Just store these error metrics in our models_evaluation datastructure
models_evaluation_train['svd'] = svd_train_results
models_evaluation_test['svd'] = svd_test_results
```

```

Training the model...
Processing epoch 0
Processing epoch 1
Processing epoch 2
Processing epoch 3
Processing epoch 4
Processing epoch 5
Processing epoch 6
Processing epoch 7
Processing epoch 8
Processing epoch 9
Processing epoch 10
Processing epoch 11
Processing epoch 12
Processing epoch 13
Processing epoch 14
Processing epoch 15
Processing epoch 16
Processing epoch 17
Processing epoch 18
Processing epoch 19
Done. time taken : 0:00:05.616780

```

```

Evaluating the model with train data..
time taken : 0:00:00.868914
-----
```

```
Train Data
```

```
-----
```

```
RMSE : 0.6574721240954099
```

```
MAPE : 19.704901088660478
```

```
adding train results in the dictionary..
```

```
Evaluating for test data...
```

```
time taken : 0:00:00.039009
```

```
-----
```

```
Test Data
```

```
-----
```

```
RMSE : 1.0726046873826458
```

```
MAPE : 35.01953535988152
```

```
storing the test results in test dictionary...
```

```
-----
```

```
Total time taken to run this algorithm : 0:00:06.525704
```

#### 4.4.6.2 SVD Matrix Factorization with implicit feedback from user ( user rated movies )

```
In [37]: from surprise import SVDpp
```

- -----> 2.5 Implicit Feedback in <http://courses.ischool.berkeley.edu/i290-dm/s11/SECURE/a1-koren.pdf>

#### Predicted Rating :

$$\circ \hat{r}_{ui} = \mu + b_u + b_i + q_i^T \left( p_u + |I_u|^{-\frac{1}{2}} \sum_{j \in I_u} y_j \right)$$

- $I_u$  --- the set of all items rated by user u
- $y_j$  --- Our new set of item factors that capture implicit ratings.

## Optimization problem with user item interactions and regularization (to avoid overfitting)

$$\circ \sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda (b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2 + ||y_j||^2)$$

```
In [38]: # initialize the model
svdpp = SVDpp(n_factors=50, random_state=15, verbose=True)
svdpp_train_results, svdpp_test_results = run_surprise(svdpp, trainset, testset, verbose=True)

# Just store these error metrics in our models_evaluation datastructure
models_evaluation_train['svdpp'] = svdpp_train_results
models_evaluation_test['svdpp'] = svdpp_test_results
```

```
Training the model...
processing epoch 0
processing epoch 1
processing epoch 2
processing epoch 3
processing epoch 4
processing epoch 5
processing epoch 6
processing epoch 7
processing epoch 8
processing epoch 9
processing epoch 10
processing epoch 11
processing epoch 12
processing epoch 13
processing epoch 14
processing epoch 15
processing epoch 16
processing epoch 17
processing epoch 18
processing epoch 19
Done. time taken : 0:01:31.840000

Evaluating the model with train data..
time taken : 0:00:03.673950
-----
Train Data
-----
RMSE : 0.6032438403305899

MAPE : 17.49285063490268

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:00.040008
-----
Test Data
-----
RMSE : 1.0728491944183447

MAPE : 35.03817913919887

storing the test results in test dictionary..

-----
Total time taken to run this algorithm : 0:01:35.553958
```

### 4.4.7 XgBoost with 13 features + Surprise Baseline + Surprise KNNbaseline + MF Techniques

#### Preparing Train data

```
In [39]: # add the predicted values from both knns to this dataframe  
reg_train['svd'] = models_evaluation_train['svd']['predictions']  
reg_train['svdpp'] = models_evaluation_train['svdpp']['predictions']  
  
reg_train.head(2)
```

```
Out[39]:   user movie    GAvg  sur1  sur2  sur3  sur4  sur5  smr1  smr2 ...  smr4  smr5    UAvg    MAvg  rating  
0  53406     33  3.581679  4.0   5.0   5.0   4.0   1.0   5.0   2.0 ...  3.0   1.0  3.370370  4.092437   4  
1  99540     33  3.581679  5.0   5.0   5.0   4.0   5.0   3.0   4.0 ...  3.0   5.0  3.555556  4.092437   3
```

2 rows × 21 columns



\_Preparing Test data \_

```
In [40]: reg_test_df['svd'] = models_evaluation_test['svd']['predictions']  
reg_test_df['svdpp'] = models_evaluation_test['svdpp']['predictions']  
  
reg_test_df.head(2)
```

```
Out[40]:   user movie    GAvg  sur1  sur2  sur3  sur4  sur5  smr1  smr2 ...  smr4  
0  808635     71  3.581679  3.581679  3.581679  3.581679  3.581679  3.581679  3.581679  3.581679 ...  3.581679  3.58  
1  941866     71  3.581679  3.581679  3.581679  3.581679  3.581679  3.581679  3.581679  3.581679 ...  3.581679  3.58
```

2 rows × 21 columns



```
In [41]: # prepare x_train and y_train  
x_train = reg_train.drop(['user', 'movie', 'rating'], axis=1)  
y_train = reg_train['rating']  
  
# prepare test data  
x_test = reg_test_df.drop(['user', 'movie', 'rating'], axis=1)  
y_test = reg_test_df['rating']  
  
  
xgb_final = xgb.XGBRegressor(n_jobs=10, random_state=15)  
train_results, test_results = run_xgboost(xgb_final, x_train, y_train, x_test, y_test)  
  
# store the results in models_evaluations dictionaries  
models_evaluation_train['xgb_final'] = train_results  
models_evaluation_test['xgb_final'] = test_results  
  
xgb.plot_importance(xgb_final)  
plt.show()
```

Training the model..

Done. Time taken : 0:00:04.775536

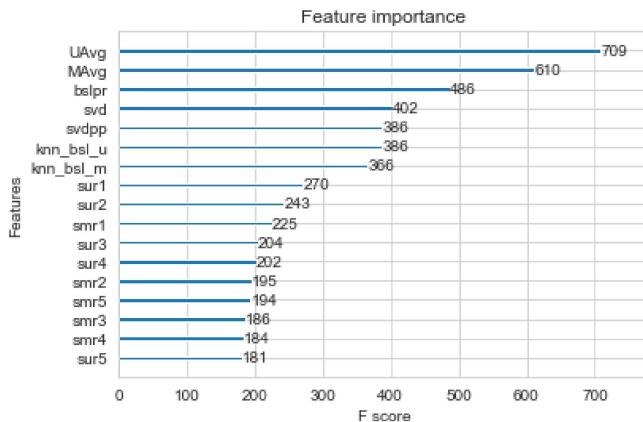
Done

Evaluating the model with TRAIN data...

Evaluating Test data

TEST DATA

```
-----  
RMSE :  1.0858226171103014  
MAPE :  33.98041189391535
```



#### 4.4.8 XgBoost with Surprise Baseline + Surprise KNNbaseline + MF Techniques

```
In [45]: # prepare train data
x_train = reg_train[['knn_bsl_u', 'knn_bsl_m', 'svd', 'svdpp','bslpr']]
y_train = reg_train['rating']

# test data
x_test = reg_test_df[['knn_bsl_u', 'knn_bsl_m', 'svd', 'svdpp','bslpr']]
y_test = reg_test_df['rating']

xgb_all_models = xgb.XGBRegressor(n_jobs=10, random_state=15)
train_results, test_results = run_xgboost(xgb_all_models, x_train, y_train, x_test, y_test)

# store the results in models_evaluations dictionaries
models_evaluation_train['xgb_all_models'] = train_results
models_evaluation_test['xgb_all_models'] = test_results

xgb.plot_importance(xgb_all_models)
plt.show()
```

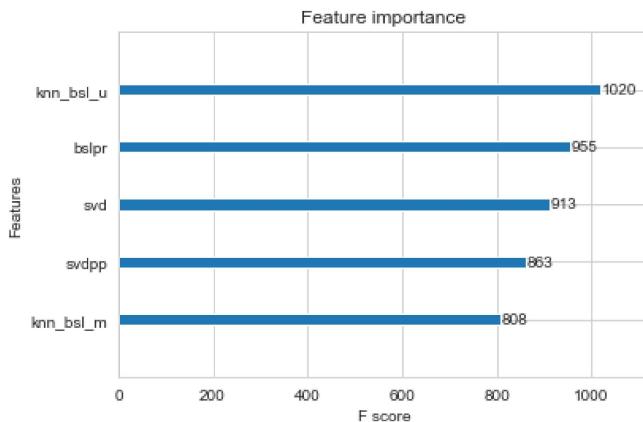
Training the model..  
Done. Time taken : 0:00:03.894027

Done

Evaluating the model with TRAIN data...  
Evaluating Test data

TEST DATA

-----  
RMSE : 1.0753789980923527  
MAPE : 35.049538657450555



#### 4.5 Comparision between all models

```
In [46]: # Saving our TEST_RESULTS into a dataframe so that you don't have to run it again
pd.DataFrame(models_evaluation_test).to_csv('small_sample_results.csv')
```

```
models = pd.read_csv('small_sample_results.csv', index_col=0)
models.loc['rmse'].sort_values()
```

```
Out[46]: svd           1.0726046873826458
knn_bsl_u        1.0726493739667242
knn_bsl_m        1.072758832653683
svdpp          1.0728491944183447
bsl_algo         1.0730330260516174
xgb_all_models   1.0753789980923527
xgb_final        1.0858226171103014
xgb_knn_bsl     1.0887576141058135
xgb_bsl          1.1000463052939882
first_algo       1.1016770831525975
Name: rmse, dtype: object
```

```
In [1]: from prettytable import PrettyTable
```

```
data = [
    ["svd", 1.0726046873826458],
    ["knn_bsl_u", 1.0726493739667242],
    ["knn_bsl_m", 1.072758832653683],
    ["svdpp", 1.0728491944183447],
    ["bsl_algo", 1.0730330260516174],
    ["xgb_all_models", 1.0753789980923527],
    ["xgb_final", 1.0858226171103014],
    ["xgb_knn_bsl", 1.0887576141058135],
    ["xgb_bsl", 1.1000463052939882],
    ["first_algo", 1.1016770831525975]
]

table = PrettyTable()
table.field_names = ["Model", "RMSE= Root Mean Square Error"]

for row in data:
    table.add_row(row)

print(table)
```

Model	RMSE= Root Mean Square Error
svd	1.0726046873826458
knn_bsl_u	1.0726493739667242
knn_bsl_m	1.072758832653683
svdpp	1.0728491944183447
bsl_algo	1.0730330260516174
xgb_all_models	1.0753789980923527
xgb_final	1.0858226171103014
xgb_knn_bsl	1.0887576141058135
xgb_bsl	1.1000463052939882
first_algo	1.1016770831525975