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 (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 subsequen
t 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
- Root Mean Square Error: 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')
```

```python
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

from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import TimeSeriesSplit
```

3. Exploratory Data Analysis

3.1 Preprocessing

3.1.1 Converting / Merging whole data to required format: u_i, m_j, r_ij

In [2]:

```python
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)
```

```
Time taken : 0:00:00
```

In [3]:

```python
print("creating the dataframe from data.csv file..")
df = pd.read_csv('data.csv', sep=',',
                       names=['movie', 'user','rating','date'])
df.date = pd.to_datetime(df.date)
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 [4]:

```
df.head()
```

Out[4]:

	movie	user	rating	date
56431994	10341	510180	4	1999-11-11
9056171	1798	510180	5	1999-11-11
58698779	10774	510180	3	1999-11-11
48101611	8651	510180	2	1999-11-11
81893208	14660	510180	2	1999-11-11

In [5]:

```
df.describe()['rating']
```

Out[5]:

```
count    1.004805e+08
mean     3.604290e+00
std      1.085219e+00
min      1.000000e+00
25%      3.000000e+00
50%      4.000000e+00
75%      4.000000e+00
max      5.000000e+00
Name: rating, dtype: float64
```

3.1.2 Checking for NaN values

In [7]:

```
# Just to make sure that all Nan containing rows are deleted..
print("Number of Nan values in our dataframe : ", sum(df.isnull().any()))
```

```
Number of Nan values in our dataframe :  0
```

3.1.3 Removing Duplicates

In [8]:

```
dup_bool = df.duplicated(['movie','user','rating'])
dups = sum(dup_bool) # by considering all columns..( including timestamp)
print("There are {} duplicate rating entries in the data..".format(dups))
```

```
There are 0 duplicate rating entries in the data..
```

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

In [9]:

```
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 : 100480507
Total No of Users   : 480189
Total No of movies  : 17770
```

3.2 Spliting data into Train and Test(80:20)

In [10]:

```
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 [11]:

```
# 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 : 80384405
Total No of Users   : 405041
Total No of movies  : 17424
```

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

In [12]:

```
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 : 20096102
Total No of Users   : 349312
Total No of movies  : 17757
```

3.3 Exploratory Data Analysis on Train data

In [13]:

```python
# 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 [14]:

```python
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()
```

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

In [15]:

```python
# 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.weekday_name

train_df.tail()
```

Out[15]:

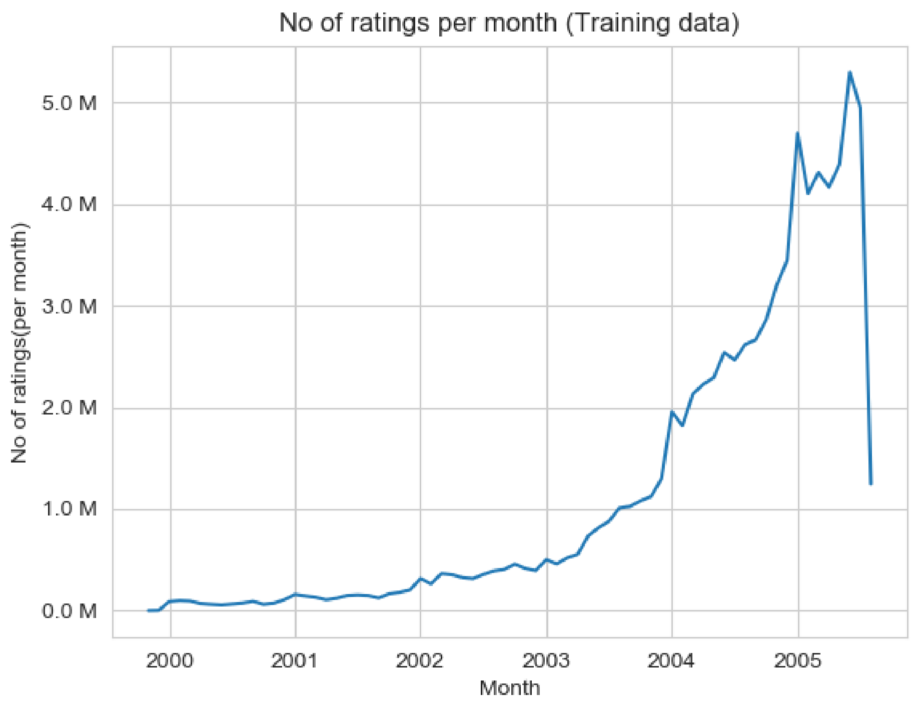
	movie	user	rating	date	day_of_week
80384400	12074	2033618	4	2005-08-08	Monday
80384401	862	1797061	3	2005-08-08	Monday

	movie	user	rating	date	day_of_week
80384402	10986	1498715	5	2005-08-08	Monday
80384403	14861	500016	4	2005-08-08	Monday
80384404	5926	1044015	5	2005-08-08	Monday

3.3.2 Number of Ratings per a month

In [16]:

```
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()
```



3.3.3 Analysis on the Ratings given by user

In [17]:

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

no_of_rated_movies_per_user.head()
```

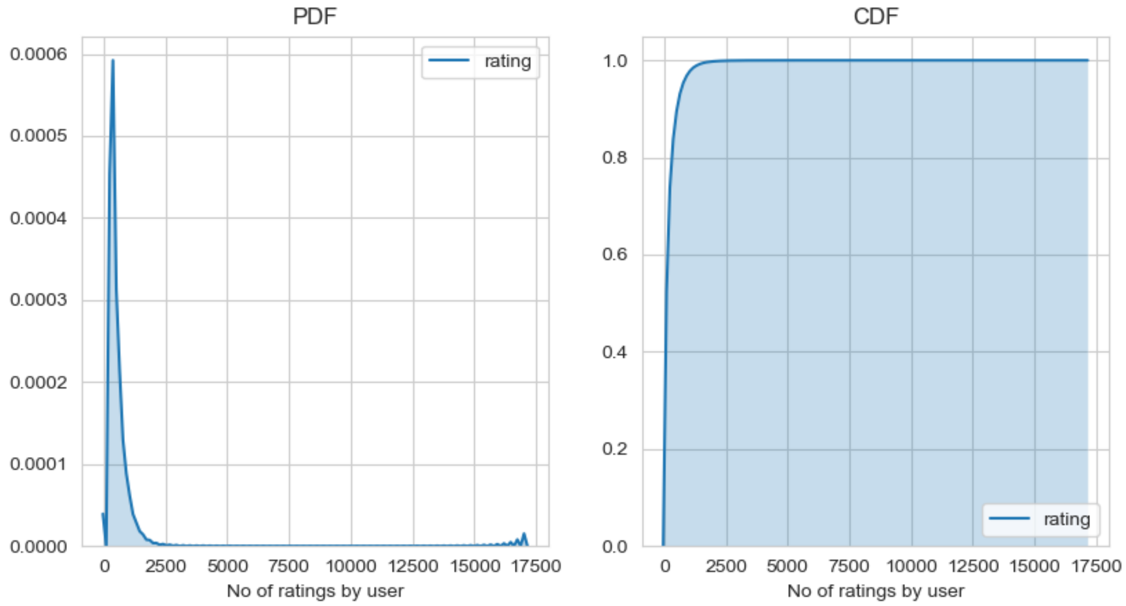
Out[17]:

```
user
305344     17112
2439493    15896
387418     15402
1639792     9767
1461435     9447
Name: rating, dtype: int64
```

In [18]:

```
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 [19]:

```
no_of_rated_movies_per_user.describe()
```

Out[19]:

```
count    405041.000000
mean        198.459921
std         290.793238
min           1.000000
25%          34.000000
50%          89.000000
75%         245.000000
max       17112.000000
Name: rating, dtype: float64
```

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

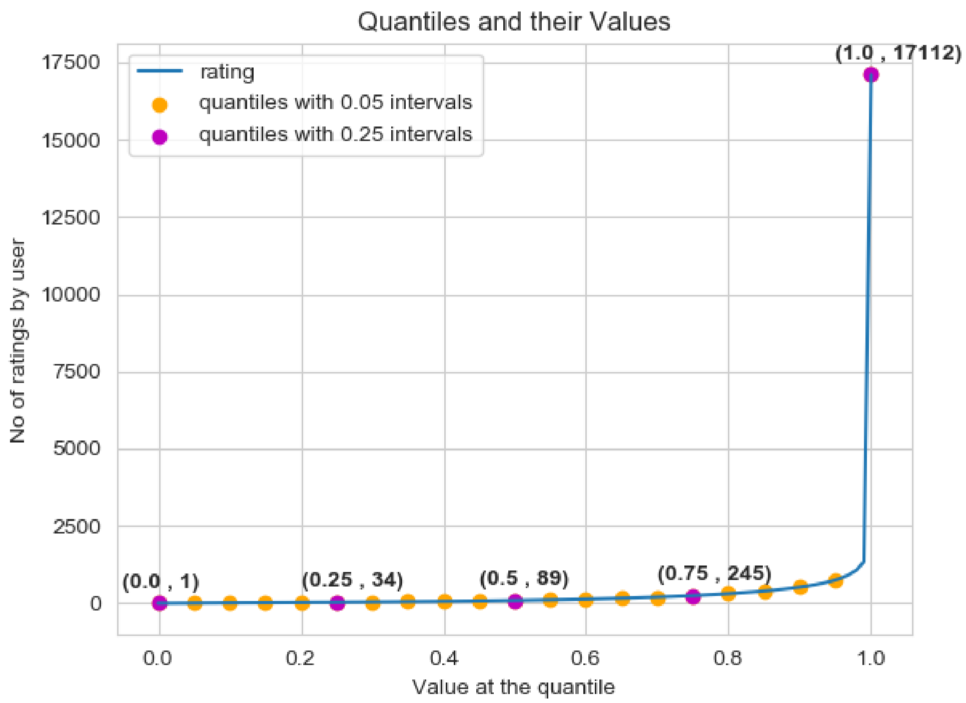
In [20]:

```
quantiles = no_of_rated_movies_per_user.quantile(np.arange(0,1.01,0.01), interpolation='higher')
```

In [21]:

```
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
intervals")
# quantiles with 0.25 difference
plt.scatter(x=quantiles.index[::25], y=quantiles.values[::25], c='m', label = "quantiles with 0.25
```

```
intervals")
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]):
    s= s="({} , {})".format(x,y)
    plt.annotate(s, xy=(x,y), xytext=(x-0.05, y+500)
                ,fontweight='bold')


plt.show()
```



In [22]:

```
quantiles[::5]
```

Out[22]:

```
0.00        1
0.05        7
0.10       15
0.15       21
0.20       27
0.25       34
0.30       41
0.35       50
0.40       60
0.45       73
0.50       89
0.55      109
0.60      133
0.65      163
0.70      199
0.75      245
0.80      307
0.85      392
0.90      520
0.95      749
1.00    17112
Name: rating, dtype: int64
```

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

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

In [23]:

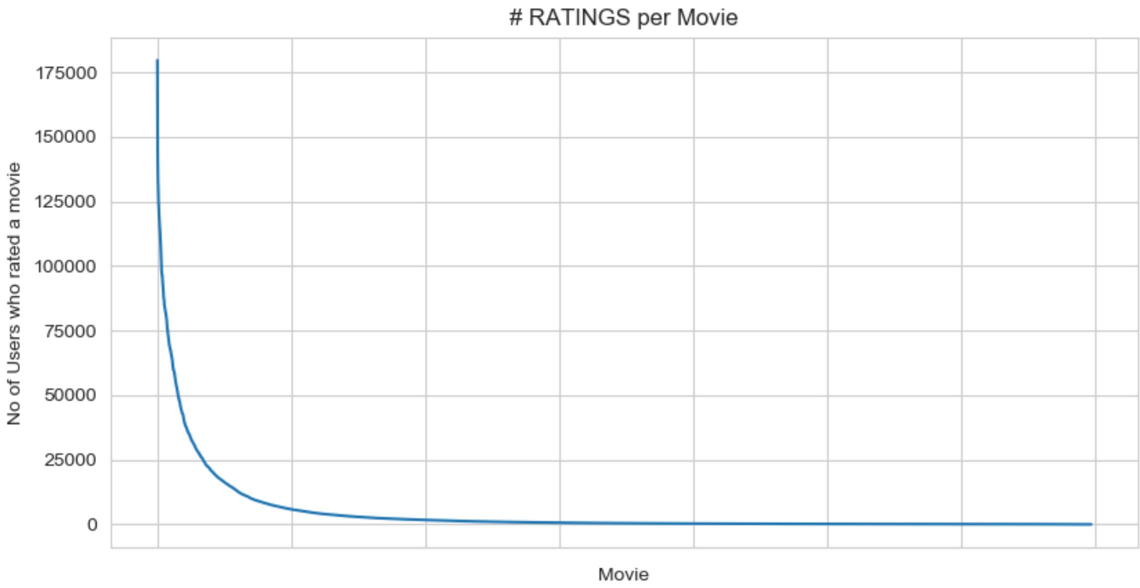
```
print('\n No of ratings at last 5 percentile : {}\n'.format(sum(no_of_rated_movies_per_user>= 749)
) )
```

```
 No of ratings at last 5 percentile : 20305
```

3.3.4 Analysis of ratings of a movie given by a user

In [24]:

```
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()
```



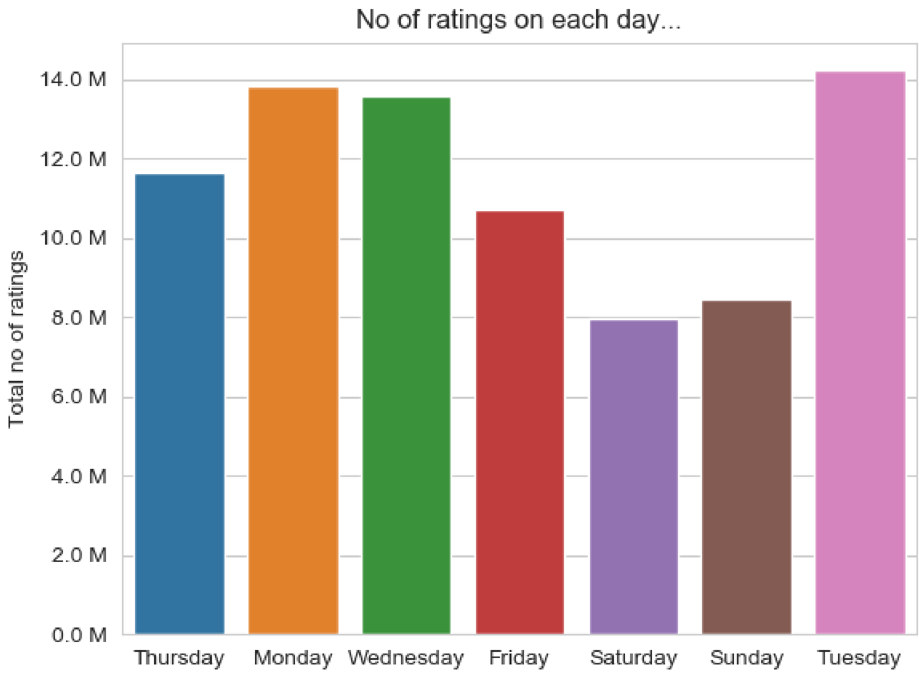
- **It is very skewed.. just like nunmber of ratings given per user.**

```
- 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 hundereds of ratings.
```

3.3.5 Number of ratings on each day of the week

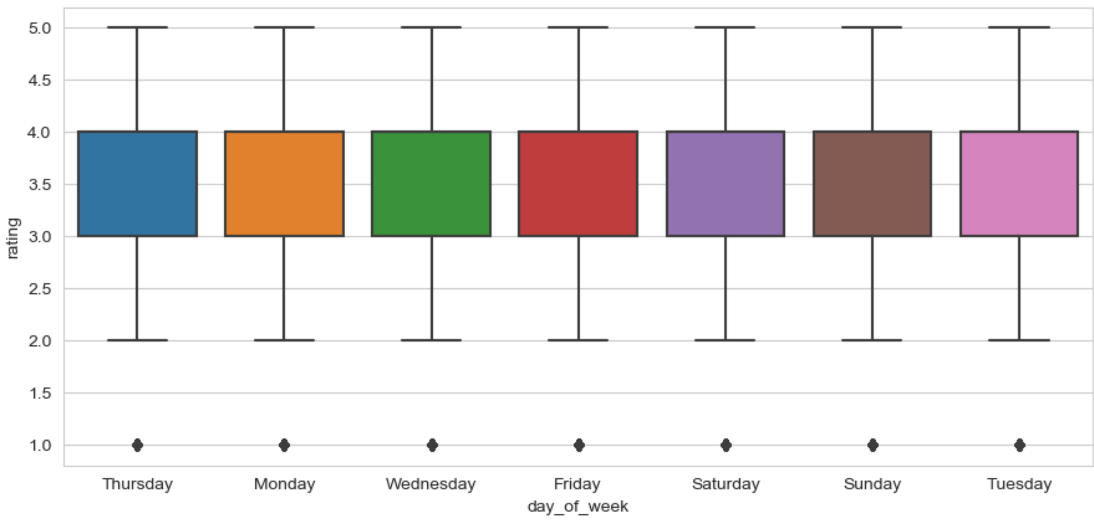
In [25]:

```
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()
```



In [26]:

```
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)
```




```
0:00:53.880971
```

In [27]:

```
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
Friday       3.585274
Monday       3.577250
Saturday     3.591791
Sunday       3.594144
Thursday     3.582463
Tuesday      3.574438
Wednesday    3.583751
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 [28]:

```
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.values)),)

    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..
0:00:02.812914
```

The Sparsity of Train Sparse Matrix

In [29]:

```
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 : 99.8292709259195 %
```

3.3.6.2 Creating sparse matrix from test data frame

In [30]:

```
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..
0:00:00.789087
```

The Sparsity of Test data Matrix

In [31]:

```
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 : 99.95731772988694 %
```

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

In [32]:

```
# 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
    # creae a dictonary of users and their average ratigns..
    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
```

3.3.7.1 finding global average of all movie ratings

In [33]:

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

```
train_averages['global'] = train_global_average
train_averages
```

Out[33]:

```
{'global': 3.582890686321557}
```

3.3.7.2 finding average rating per user

In [34]:

```
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 : 3.3781094527363185
```

3.3.7.3 finding average rating per movie

In [35]:

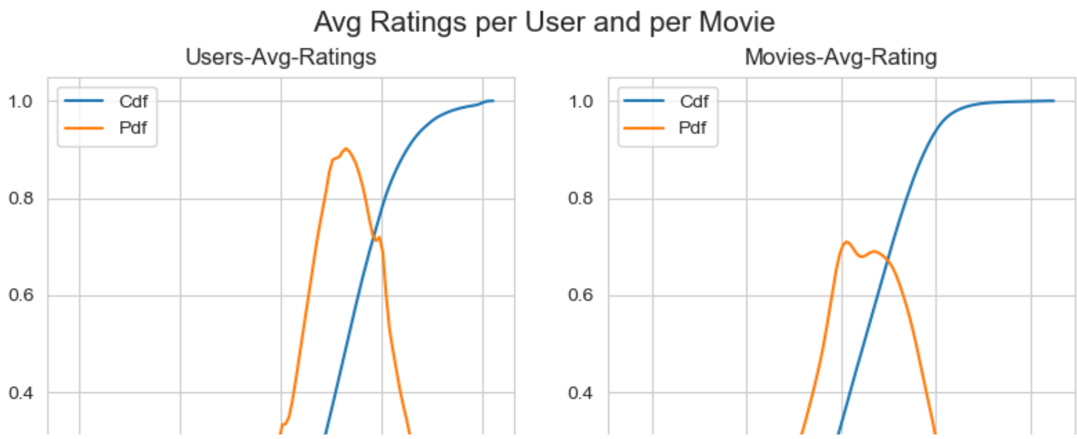
```
train_averages['movie'] =  get_average_ratings(train_sparse_matrix, of_users=False)
print('\n AVerage rating of movie 15 :',train_averages['movie'][15])
```

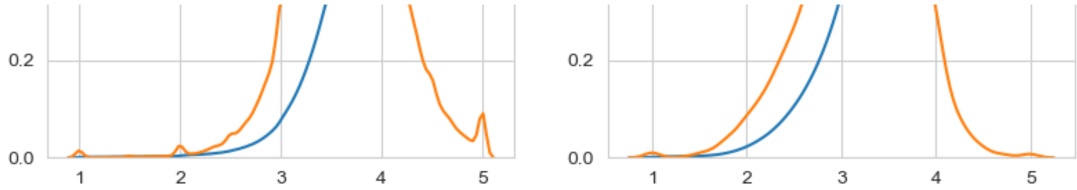
```
 AVerage rating of movie 15 : 3.3038461538461537
```

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

In [36]:

```
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 = [rat for rat in 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 = [rat for rat in 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)
```





```
0:00:46.116597
```

3.3.8 Cold Start problem

3.3.8.1 Cold Start problem with Users

In [37]:

```
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)*100, 2)))
```

```
Total number of Users  : 480189

Number of Users in Train data : 405041

No of Users that didn't appear in train data: 75148(15.65 %)
```

> 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 [38]:

```
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)*100, 2)))
```

```
Total number of Movies  : 17770

Number of Users in Train data : 17424

No of Movies that didn't appear in train data: 346(1.95 %)
```

> 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 [44]:

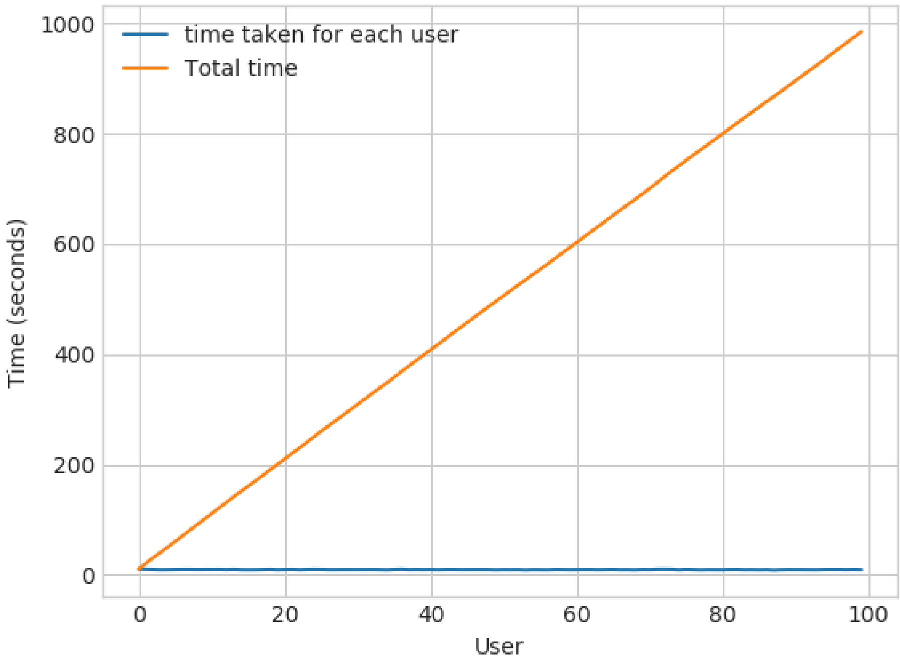
```
from sklearn.metrics.pairwise import cosine_similarity


def compute_user_similarity(sparse_matrix, compute_for_few=False, top = 100, verbose=False, verb_fo
r_n_rows = 20,
                            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 [45]:

```
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 : 0:03:20.300488  ]
computing done for 40 users [  time elapsed : 0:06:38.518391  ]
computing done for 60 users [  time elapsed : 0:09:53.143126  ]
computing done for 80 users [  time elapsed : 0:13:10.080447  ]
computing done for 100 users [  time elapsed : 0:16:24.711032  ]
Creating Sparse matrix from the computed similarities
```




```
----------------------------------------------------------------------------------------------------
Time taken : 0:16:33.618931
```

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

- We have **405,041 users** in out training set and computing similarities between them..(**17K dimensional vector..**) is time consuming..
- From above plot, It took roughly **8.88 sec** for computing simlilar users for **one user**
- We have **405,041 users** with us in training set.
- $405041 \times 8.88 = 3596764.08 \sec = 59946.068 \min$
  - 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 dimentsions using SVD, so that **it might** speed up the process...

In [46]:

```
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,

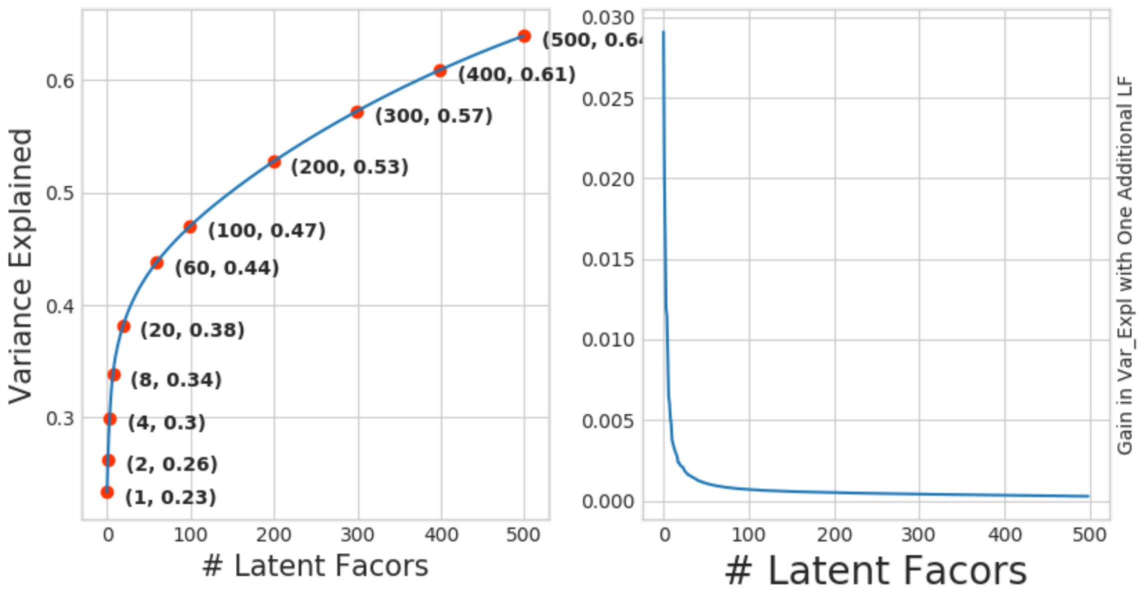
- \sum \longleftarrow (netflix_svd.**singular_values_**)
- \bigvee^T \longleftarrow (netflix_svd.**components_**)
- \bigcup is not returned. instead **Projection_of_X** onto the new vectorspace is returned.
- It uses **randomized svd** internally, which returns **All 3 of them saperately**. Use that instead..

In [47]:

```
expl_var = np.cumsum(netflix_svd.explained_variance_ratio_)
```

In [50]:

```
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)
# annote 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 [51]:

```
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 factore too it, the **_gain in expained variance** with that addition is decreasing. (Obviously, because they are sorted that way).
- ***LHS Graph***:
  - **x** --- (No of latent factos),
  - **y** --- (The variance explained by taking x latent factors)

- **More decrease in the line (RHS graph)** :
  - We are getting more expained variance than before.
- **Less decrease in that line (RHS graph)** :
  - We are not getting benifitted from adding latent factor furthur. This is what is shown in the plots.

- ***RHS Graph***:
  - **x** --- (No of latent factors),
  - **y** --- (Gain n Expl_Var by taking one additional latent factor)

In [52]:

```
# Let's project our Original U_M matrix into 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 [53]:

```
type(trunc_matrix), trunc_matrix.shape
```

Out[53]:

```
(numpy.ndarray, (2649430, 500))
```

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

In [54]:

```
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 [55]:

```
trunc_sparse_matrix.shape
```

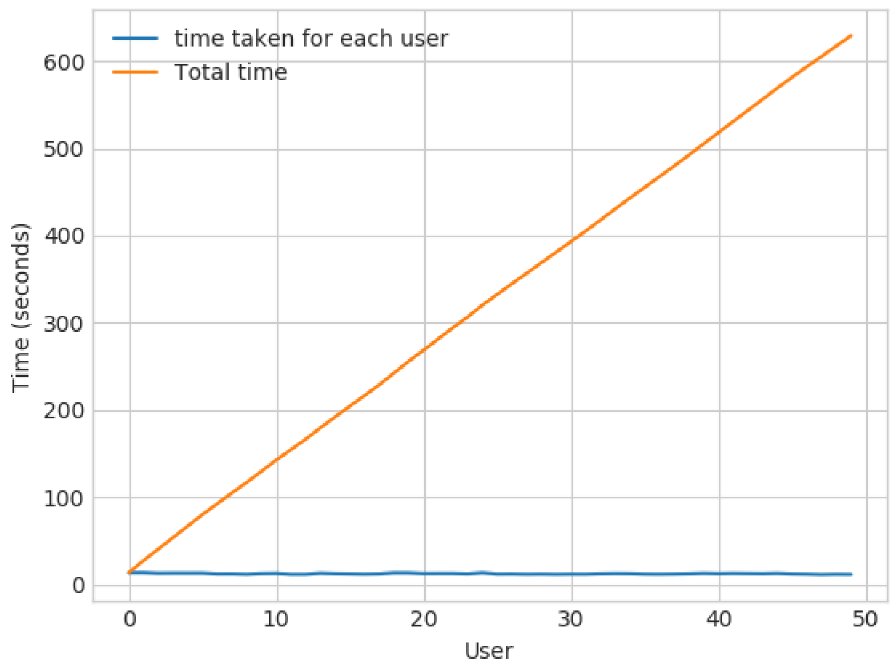
Out[55]:

```
(2649430, 500)
```

In [56]:

```
start = datetime.now()
trunc_u_u_sim_matrix, _ = compute_user_similarity(trunc_sparse_matrix, compute_for_few=True, top=50
, verbose=True,
                                                   verb_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 simlilar users for **one user**
- We have **405041 users** with us in training set.
- { 405041 \times 12.18 ==== 4933399.38 \sec } ==== 82223.323 \min ==== 1370.388716667 \text{ hours} ==== 57.099529861 \text{ days}...
  - 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, whenenver 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 [39]:

```
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
0:00:19.280507
```

In [40]:

```
m_m_sim_sparse.shape
```

Out[40]:

```
(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 [41]:

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

In [42]:

```
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]
```

```
0:00:21.033046
```

Out[42]:

```
array([ 8279,  8013, 16528,  5927, 13105, 12049,  4424, 10193, 17590,
         4549,  3755,   590, 14059, 15144, 15054,  9584,  9071,  6349,
        16402,  3973,  1720,  5370, 16309,  9376,  6116,  4706,  2818,
          778, 15331,  1416, 12979, 17139, 17710,  5452,  2534,   164,
        15188,  8323,  2450, 16331,  9566, 15301, 13213, 14308, 15984,
        10597,  6426,  5500,  7068,  7328,  5720,  9802,   376, 13013,
         8003, 10199,  3338, 15390,  9688, 16455, 11730,  4513,   598,
        12762,  2187,   509,  5865,  9166, 17115, 16334,  1942,  7282,
        17584,  4376,  8988,  8873,  5921,  2716, 14679, 11947, 11981,
         4649,   565, 12954, 10788, 10220, 10963,  9427,  1690,  5107,
         7859,  5969,  1510,  2429,   847,  7845,  6410, 13931,  9840,
         3706], dtype=int64)
```

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 [43]:

```
# First Let's load the movie details into soe dataframe..
# movie details are in 'netflix/movie_titles.csv'

movie_titles = pd.read_csv("data_folder/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: 10.94 ms
Parser memory cleanup took: 0.00 ms
```

Out[43]:

	year_of_release	title
movie_id		
1	2003.0	Dinosaur Planet
2	2004.0	Isle of Man TT 2004 Review

movie_id	year_of_release	title
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 'Vampire Journals'

In [44]:

```
mv_id = 67

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_s
im_sparse[:,mv_id].getnnz()))
```

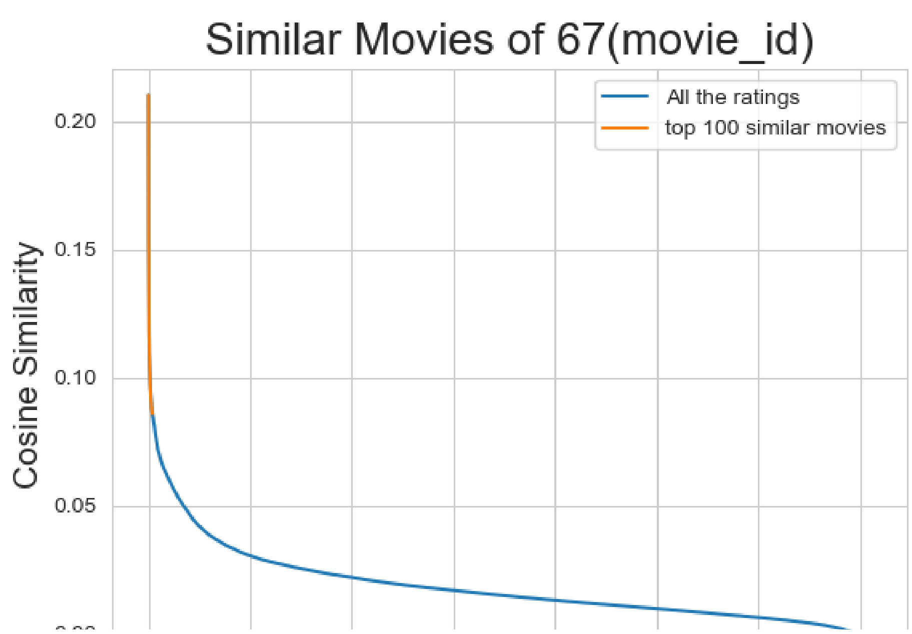
```
Movie -----> Vampire Journals

It has 270 Ratings from users.

We have 17284 movies which are similarto this  and we will get only top most..
```

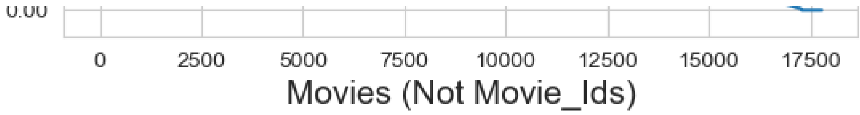
In [45]:

```
similarities = m_m_sim_sparse[mv_id].toarray().ravel()

similar_indices = similarities.argsort()[::-1][1:]

similarities[similar_indices]

sim_indices = similarities.argsort()[::-1][1:] # It will sort and reverse the array and ignore its
similarity (ie.,1)
                                               # and return its indices(movie_ids)
```

In [46]:

```
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 [47]:

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

Out[47]:

	year_of_release	title
movie_id		
323	1999.0	Modern Vampires
4044	1998.0	Subspecies 4: Bloodstorm
1688	1993.0	To Sleep With a Vampire
13962	2001.0	Dracula: The Dark Prince
12053	1993.0	Dracula Rising
16279	2002.0	Vampires: Los Muertos
4667	1996.0	Vampirella
1900	1997.0	Club Vampire
13873	2001.0	The Breed
15867	2003.0	Dracula II: Ascension

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

4. Machine Learning Models

In [48]:

```
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 -- {}\n".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(row_ind, sample_users),
                      np.isin(col_ind, sample_movies) )

    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_mo
vies)))
        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 [50]:

```
start = datetime.now()
path = "sample/small/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 25k users and 3k movies from available data
    sample_train_sparse_matrix = get_sample_sparse_matrix(train_sparse_matrix, no_users=25000, no_m
ovies=3000,
                                             path = path)

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

```
Original Matrix : (users, movies) -- (405041 17424)
Original Matrix : Ratings -- 80384405

Sampled Matrix : (users, movies) -- (25000 3000)
Sampled Matrix : Ratings -- 856986
Saving it into disk for furthur usage..
Done..

0:00:31.136247
```

4.1.2 Build sample test data from the test data

In [51]:

```
start = datetime.now()

path = "sample/small/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 20k users and 1000 movies from available data
    sample_test_sparse_matrix = get_sample_sparse_matrix(test_sparse_matrix, no_users=20000, no_mov
ies=1000,
                                                 path = "sample/small/sample_test_sparse_matrix.npz
)
```

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

```
Original Matrix : (users, movies) -- (349312 17757)
Original Matrix : Ratings -- 20096102

Sampled Matrix : (users, movies) -- (20000 1000)
Sampled Matrix : Ratings -- 71392
Saving it into disk for furthur usage..
Done..

0:00:07.585923
```

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

In [52]:

```
sample_train_averages = dict()
```

4.2.1 Finding Global Average of all movie ratings

In [53]:

```
# 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[53]:

```
{'global': 3.5875813607223455}
```

4.2.2 Finding Average rating per User

In [54]:

```
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 : 3.923076923076923
```

4.2.3 Finding Average rating per Movie

In [55]:

```
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 : 2.752
```

4.3 Featurizing data

In [56]:

```
print('\n No of ratings in Our Sampled train matrix is : {}\n'.format(sample_train_sparse_matrix.c
ount_nonzero()))
print('\n No of ratings in Our Sampled test  matrix is : {}\n'.format(sample_test_sparse_matrix.co
unt_nonzero()))
```

```
No of ratings in Our Sampled train matrix is : 856986


No of ratings in Our Sampled test  matrix is : 71392
```

4.3.1 Featurizing data for regression problem

4.3.1.1 Featurizing train data

In [57]:

```
# 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 30 hours to prepare this train dataset.#
############################################################
start = datetime.now()
if os.path.isfile('sample/small/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('sample/small/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 simi
lar users.
            # 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_ratings)))
        #     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.T).ravel()
            top_sim_movies = movie_sim.argsort()[::-1][1:] # we are ignoring 'The User' from its si
milar users.
            # 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_ratings)))
        #     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)
```

```
            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)
```

Reading from the file to make a Train_dataframe

In [2]:

```
reg_train = pd.read_csv('sample/small/reg_train.csv', names = ['user', 'movie', 'GAvg', 'sur1', 'su
r2', 'sur3', 'sur4', 'sur5','smr1', 'smr2', 'smr3', 'smr4', 'smr5', 'UAvg', 'MAvg', 'rating'],
header=None)
reg_train.head()
```

Out[2]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UAvg	MAvg	rating
0	174683	10	3.587581	5.0	5.0	3.0	4.0	4.0	3.0	5.0	4.0	3.0	2.0	3.882353	3.611111	5
1	233949	10	3.587581	4.0	4.0	5.0	1.0	3.0	2.0	3.0	2.0	3.0	3.0	2.692308	3.611111	3
2	555770	10	3.587581	4.0	5.0	4.0	4.0	5.0	4.0	2.0	5.0	4.0	4.0	3.795455	3.611111	4
3	767518	10	3.587581	2.0	5.0	4.0	4.0	3.0	5.0	5.0	4.0	4.0	3.0	3.884615	3.611111	5
4	894393	10	3.587581	3.0	5.0	4.0	4.0	3.0	4.0	4.0	4.0	4.0	4.0	4.000000	3.611111	4

- **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 [59]:

```
# 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 [60]:

```
sample_train_averages['global']
```

Out[60]:

```
3.5875813607223455
```

In [ ]:

```
start = datetime.now()

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

    print('preparing {} tuples for the dataset..\n'.format(len(sample_test_ratings)))
    with open('sample/small/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).ravel()
                top_sim_users = user_sim.argsort()[::-1][1:] # we are ignoring 'The User' from its
similar users.
                # get the ratings of most similar users for this movie
                top_ratings = 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_ratings)))
                # 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 simi
lar movies...
                ########## Cold STart Problem ##########
                top_sim_users_ratings.extend([sample_train_averages['global']]*(5 -
len(top_sim_users_ratings)))
                #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.T).ravel()
                top_sim_movies = movie_sim.argsort()[::-1][1:] # we are ignoring 'The User' from it
s similar users.
                # 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_ratings)))
                #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_ratings)))
                #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)
```

Reading from the file to make a test dataframe

In [3]:

```
reg_test_df = pd.read_csv('sample/small/reg_test.csv', names = ['user', 'movie', 'GAvg', 'sur1', 'sur2', 'sur3', 'sur4', 'sur5',
                                                          'smr1', 'smr2', 'smr3', 'smr4', 'smr5',
                                                          'UAvg', 'MAvg', 'rating'], header=None)
reg_test_df.head(4)
```

Out[3]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	s
0	1129620	2	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.58
1	3321	5	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.58
2	508584	5	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.58
3	731988	5	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.58

- **GAvg** : Average rating of all the ratings
- **Similar users rating of this movie**:
  - sur1, sur2, sur3, sur4, sur5 (top 5 simiular users who rated that movie..)
- **Similar movies rated by this user**:

- smr1, smr2, smr3, smr4, smr5 (top 5 simiular 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 [19]:

```
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

In [20]:

```
# 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 impotant)

In [21]:

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

Out[21]:

```
[(1129620, 2, 3), (3321, 5, 4), (508584, 5, 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)
    >
    > **value**: dict(**key** : metric, **value** : value)

In [22]:

```
models_evaluation_train = dict()
models_evaluation_test = dict()

models_evaluation_train, models_evaluation_test
```

Out[22]:

```
({}, {})
```

Utility functions for running regression models

In [23]:

```
# 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 : {}\n'.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 [24]:

```
# 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 objecs
################################################################
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 rat
ings''.
    '''
    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 : {}\n\nMAPE : {}\n".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 : {}\n\nMAPE : {}\n".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 [34]:

```
import xgboost as xgb
```

In [9]:

```
# 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']
```

Before running XGBRegressor, we will tune hyperparameter using gridsearch cross validation.

In [16]:

```
parameters = {'max_depth':[1,2,3],
              'learning_rate':[0.001,0.01,0.1],
              'n_estimators':[100,300,500,700,900,1100,1300]}
```

In [14]:

```
start = datetime.now()

# Initialize Our first XGBoost model
first_xgb = xgb.XGBRegressor(nthread=-1)

# Perform cross validation
gscv = GridSearchCV(first_xgb,
                    param_grid = parameters,
                    scoring="neg_mean_squared_error",
```

```
                     cv = TimeSeriesSplit(n_splits=5),
                     n_jobs = -1,
                     verbose = 1)
gscv_result = gscv.fit(x_train, y_train)

# Summarize results
print("Best: %f using %s" % (gscv_result.best_score_, gscv_result.best_params_))
means = gscv_result.cv_results_['mean_test_score']
stds = gscv_result.cv_results_['std_test_score']
params = gscv_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

print("\nTime Taken: ",start - datetime.now())
```

```
Fitting 5 folds for each of 63 candidates, totalling 315 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 6 concurrent workers.
[Parallel(n_jobs=-1)]: Done  38 tasks      | elapsed:  6.0min
[Parallel(n_jobs=-1)]: Done 188 tasks      | elapsed: 53.4min
[Parallel(n_jobs=-1)]: Done 315 out of 315 | elapsed: 106.2min finished
```

```
Best: -0.745368 using {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 500}
-8.945703 (0.137510) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 100}
-6.327478 (0.104875) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 300}
-4.565558 (0.079707) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 500}
-3.380229 (0.061259) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 700}
-2.582680 (0.048964) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 900}
-2.044285 (0.040267) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 1100}
-1.679693 (0.033609) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 1300}
-8.914855 (0.117058) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 100}
-6.273837 (0.083362) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 300}
-4.499942 (0.063916) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 500}
-3.308289 (0.050271) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 700}
-2.505816 (0.039182) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 900}
-1.963202 (0.030638) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 1100}
-1.595940 (0.024345) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 1300}
-8.905508 (0.117877) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 100}
-6.247740 (0.081909) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 300}
-4.466706 (0.061004) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 500}
-3.270048 (0.044598) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 700}
-2.464795 (0.033138) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 900}
-1.920709 (0.025080) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1100}
-1.553256 (0.019707) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1300}
-2.274944 (0.043579) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 100}
-0.900042 (0.022040) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 300}
-0.821135 (0.023411) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 500}
-0.791372 (0.023354) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 700}
-0.774759 (0.023179) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 900}
-0.764954 (0.023098) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 1100}
-0.758936 (0.023106) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 1300}
-2.195542 (0.034471) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 100}
-0.824037 (0.019728) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 300}
-0.767430 (0.023112) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 500}
-0.754642 (0.023585) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 700}
-0.749864 (0.023484) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 900}
-0.747864 (0.023287) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 1100}
-0.746903 (0.023098) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 1300}
-2.154470 (0.028357) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100}
-0.795094 (0.020759) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 300}
-0.753017 (0.023484) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 500}
-0.747948 (0.023575) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 700}
-0.746570 (0.023457) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 900}
-0.745930 (0.023312) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1100}
-0.745617 (0.023259) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1300}
-0.767599 (0.023340) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 100}
-0.748313 (0.023040) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 300}
-0.747817 (0.022750) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 500}
-0.747758 (0.022691) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 700}
-0.747812 (0.022755) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 900}
-0.747840 (0.022799) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 1100}
-0.747879 (0.022835) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 1300}
-0.749680 (0.023277) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 100}
-0.746150 (0.022890) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 300}
-0.745956 (0.023104) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 500}
-0.745973 (0.023155) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 700}
```

```
-0.745975 (0.023155) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 700}
-0.746090 (0.023357) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 900}
-0.746282 (0.023427) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 1100}
-0.746507 (0.023597) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 1300}
-0.747077 (0.023291) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
-0.745460 (0.023526) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 300}
-0.745368 (0.023600) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 500}
-0.745689 (0.023732) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 700}
-0.746000 (0.023941) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 900}
-0.746268 (0.024177) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 1100}
-0.746524 (0.024701) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 1300}

Time Taken:  -1 day, 22:13:07.013313
```

In [15]:

```
print("\nTime Taken: ",datetime.now()- start)
```

```
Time Taken:  1:49:38.732357
```

In [17]:

```
# Create new instance of XGBRegressor with tuned hyperparameters
first_xgb = xgb.XGBRegressor(max_depth=3,learning_rate = 0.1,n_estimators=500,nthread=-1)
first_xgb
```

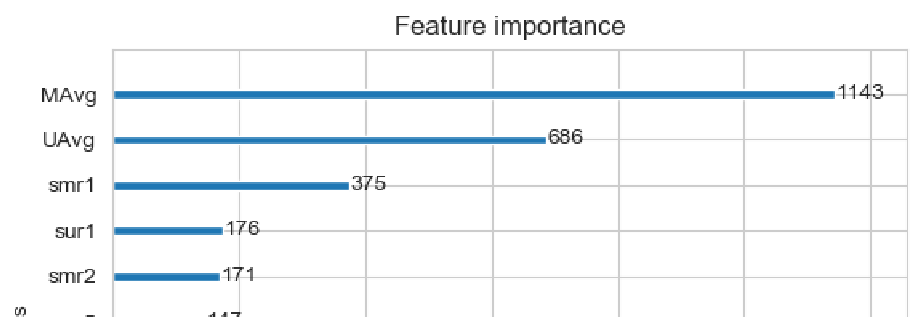
Out[17]:

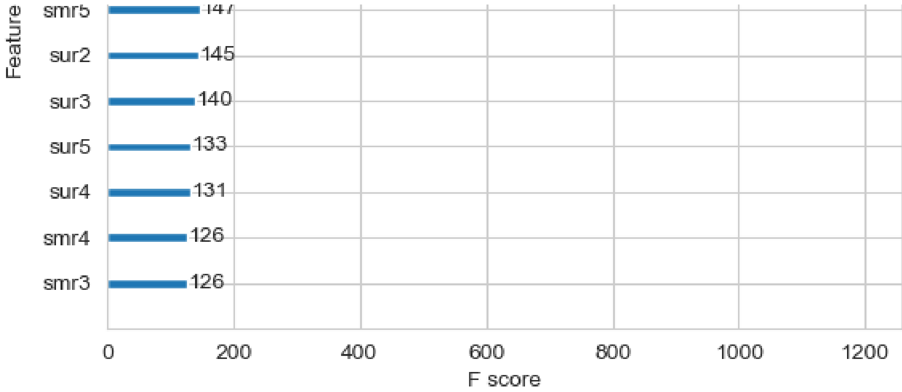
```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=500,
       n_jobs=1, nthread=-1, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1)
```

In [25]:

```
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..
Done. Time taken : 0:00:40.392801

Done

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

TEST DATA
------------------------------
RMSE :  1.0904043649061108
MAPE :  34.459846201416916
```





4.4.2 Suprise BaselineModel

In [26]:

```
from surprise import BaselineOnly
```

Predicted_rating : (baseline prediction)

```
-
http://surprise.readthedocs.io/en/stable/basic_algorithms.html#surprise.prediction_algorithm
seline_only.BaselineOnly
```

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

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

Optimization function (Least Squares Problem)

```
- http://surprise.readthedocs.io/en/stable/prediction_algorithms.html#baselines-estimates-c
onfiguration
```

> \large \sum_{r_{ui} \in R_{train}} \left(r_{ui} - (\mu + b_u + b_i)\right)^2 + \lambda \left(b_u^2 + b_i^2 \right).\text {
> [mimimize } {b_u, b_i]}

In [27]:

```
# Instantiate BaselineOnly
bsl_options = {'method': 'sgd',
               'reg':0.01,
               'learning_rate': 0.001,
               'n_epochs':120
               }
bsl_algo = BaselineOnly(bsl_options=bsl_options)
bsl_algo
```

Out[27]:

```
<surprise.prediction_algorithms.baseline_only.BaselineOnly at 0x1be65020c18>
```

In [28]:

```
%%time

# 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 : 0:00:21.416539

Evaluating the model with train data..
time taken : 0:00:04.768295
---------------
Train Data
---------------
RMSE : 0.899910618083272

MAPE : 27.478869818662556

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:00.936322
---------------
Test Data
---------------
RMSE : 1.085201374793734

MAPE : 34.47186282789672

storing the test results in test dictionary...

---------------------------------------------
Total time taken to run this algorithm : 0:00:27.121156
Wall time: 27.2 s
```

4.4.3 XGBoost with initial 13 features + Surprise Baseline predictor

Updating Train Data

In [29]:

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

Out[29]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UAvg	MAvg	rating	bslp
0	174683	10	3.587581	5.0	5.0	3.0	4.0	4.0	3.0	5.0	4.0	3.0	2.0	3.882353	3.611111	5	3.87725
1	233949	10	3.587581	4.0	4.0	5.0	1.0	3.0	2.0	3.0	2.0	3.0	3.0	2.692308	3.611111	3	3.88710

Updating Test Data

In [30]:

```
# 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)
```

Out[30]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	s
0	1129620	2	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.58
1	3321	5	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.58

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	s

In [31]:

```
# 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']
```

Before running XGBRegressor, we will tune hyperparameter using gridsearch cross validation.

In [32]:

```
start = datetime.now()

# Initialize Our first XGBoost model
xgb = xgb.XGBRegressor(nthread=-1)

# Perform cross validation
gscv = GridSearchCV(xgb,
                    param_grid = parameters,
                    scoring="neg_mean_squared_error",
                    cv = TimeSeriesSplit(n_splits=5),
                    n_jobs = -1,
                    verbose = 1)
gscv_result = gscv.fit(x_train, y_train)

# Summarize results
print("Best: %f using %s" % (gscv_result.best_score_, gscv_result.best_params_))
print()
means = gscv_result.cv_results_['mean_test_score']
stds = gscv_result.cv_results_['std_test_score']
params = gscv_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

print("\nTime Taken: ",datetime.now() -start)
```

```
Fitting 5 folds for each of 63 candidates, totalling 315 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 6 concurrent workers.
[Parallel(n_jobs=-1)]: Done  38 tasks      | elapsed:  7.4min
[Parallel(n_jobs=-1)]: Done 188 tasks      | elapsed: 65.3min
[Parallel(n_jobs=-1)]: Done 315 out of 315 | elapsed: 126.8min finished
```

```
Best: -0.745661 using {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1300}

-8.945703 (0.137510) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 100}
-6.327478 (0.104875) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 300}
-4.565558 (0.079707) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 500}
-3.380229 (0.061259) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 700}
-2.582680 (0.048964) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 900}
-2.044285 (0.040267) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 1100}
-1.679693 (0.033609) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 1300}
-8.914855 (0.117058) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 100}
-6.273837 (0.083362) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 300}
-4.499942 (0.063916) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 500}
-3.308289 (0.050271) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 700}
-2.505816 (0.039182) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 900}
-1.963202 (0.030638) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 1100}
-1.595940 (0.024345) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 1300}
-8.905508 (0.117877) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 100}
-6.247740 (0.081909) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 300}
-4.466706 (0.061004) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 500}
-3.270048 (0.044598) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 700}
-2.464795 (0.033138) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 900}
-1.920709 (0.025080) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1100}
-1.553256 (0.019707) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1300}
-2.274944 (0.043579) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 100}
-0.900042 (0.022040) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 300}
```

```
-0.821135 (0.023411) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 500}
-0.791372 (0.023354) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 700}
-0.774759 (0.023179) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 900}
-0.764954 (0.023098) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 1100}
-0.758936 (0.023106) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 1300}
-2.195542 (0.034471) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 100}
-0.824037 (0.019728) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 300}
-0.767430 (0.023112) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 500}
-0.754642 (0.023585) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 700}
-0.749864 (0.023484) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 900}
-0.747864 (0.023287) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 1100}
-0.746904 (0.023099) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 1300}
-2.154470 (0.028357) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100}
-0.795094 (0.020759) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 300}
-0.753019 (0.023487) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 500}
-0.747950 (0.023563) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 700}
-0.746587 (0.023439) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 900}
-0.745977 (0.023324) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1100}
-0.745661 (0.023264) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1300}
-0.767599 (0.023340) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 100}
-0.748327 (0.023064) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 300}
-0.747828 (0.022771) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 500}
-0.747779 (0.022714) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 700}
-0.747825 (0.022759) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 900}
-0.747860 (0.022799) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 1100}
-0.747890 (0.022836) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 1300}
-0.749680 (0.023277) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 100}
-0.746225 (0.022905) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 300}
-0.746023 (0.023080) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 500}
-0.746002 (0.023206) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 700}
-0.746121 (0.023261) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 900}
-0.746329 (0.023526) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 1100}
-0.746564 (0.023616) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 1300}
-0.747100 (0.023334) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
-0.745680 (0.023441) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 300}
-0.745692 (0.023488) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 500}
-0.745797 (0.023662) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 700}
-0.746108 (0.023736) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 900}
-0.746523 (0.023923) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 1100}
-0.746627 (0.024125) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 1300}

Time Taken:  2:08:55.556027
```

In [35]:

```
# Create new instance of XGBRegressor with tuned hyperparameters
xgb_bsl = xgb.XGBRegressor(max_depth=3,learning_rate = 0.01,n_estimators=1300,nthread=-1)
xgb_bsl
```

Out[35]:

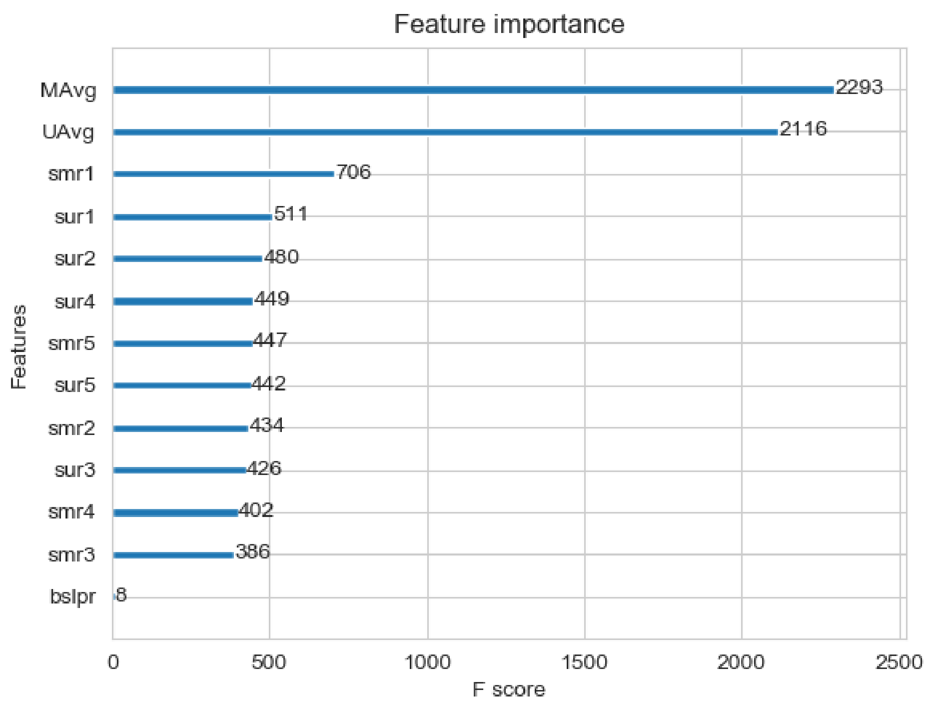
```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.01, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=1300,
       n_jobs=1, nthread=-1, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1)
```

In [36]:

```
# Run XGBRegressor
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..
Done. Time taken : 0:02:07.816007

Done
```

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

TEST DATA
------------------------------
RMSE :  1.0897253482697786
MAPE :  34.50826793446596
```



4.4.4 Surprise KNNBaseline predictor

In [37]:

```
from surprise import KNNBaseline
```

- KNN BASELINE
  - http://surprise.readthedocs.io/en/stable/knn_inspired.html#surprise.prediction_algorithms.knns.KNNBaseline
- PEARSON_BASELINE SIMILARITY
  - 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***)

\begin{align} \hat{r}_{ui} = b_{ui} + \frac{ \sum\limits_{v \in N^k_i(u)} \text{sim}(u, v) \cdot (r_{vi} - b_{vi})} {\sum\limits_{v \in N^k_i(u)} \text{sim}(u, v)} \end{align}

- \pmb{b_{ui}} - *Baseline prediction* of (user,movie) rating
- \pmb {N_i^k (u)} - Set of **K similar** users (neighbours) of **user (u)** who rated **movie(i)**
- *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): \begin{align} \hat{r}_{ui} = b_{ui} + \frac{ \sum\limits_{j \in N^k_u(i)}\text{sim}(i, j) \cdot (r_{uj} - b_{uj})} {\sum\limits_{j \in N^k_u(j)} \text{sim}(i, j)} \end{align}
    - ***Notations follows same as above (user user based predicted rating)***

4.4.4.1 Surprise KNNBaseline with user user similarities

In [38]:

```
# 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 : 0:10:00.104928

Evaluating the model with train data..
time taken : 0:15:38.442814
---------------
Train Data
---------------
RMSE : 0.4536279292470732

MAPE : 12.840252350475915

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:02.058800
---------------
Test Data
---------------
RMSE : 1.0850618463554647

MAPE : 34.48062216705011

storing the test results in test dictionary...

---------------------------------------------
Total time taken to run this algorithm : 0:25:40.608556
```

4.4.4.2 Surprise KNNBaseline with movie movie similarities

In [39]:

```
# we specify , how to compute similarities and what to consider with sim_options to our algorithm

# 'user_based' : Fals => 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'}
```

```
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 : 0:00:14.337379

Evaluating the model with train data..
time taken : 0:01:21.398122
---------------
Train Data
---------------
RMSE : 0.5038994796517224

MAPE : 14.168515366483724

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:01.148349
---------------
Test Data
---------------
RMSE : 1.0852678745012594

MAPE : 34.48337123552355

storing the test results in test dictionary...

---------------------------------------------
Total time taken to run this algorithm : 0:01:36.883850
```

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 form both knn models and preditions from our baseline model.

Preparing Train data

In [40]:

```
# 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[40]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UAvg	MAvg	rating	bslp
0	174683	10	3.587581	5.0	5.0	3.0	4.0	4.0	3.0	5.0	4.0	3.0	2.0	3.882353	3.611111	5	3.87725
1	233949	10	3.587581	4.0	4.0	5.0	1.0	3.0	2.0	3.0	2.0	3.0	3.0	2.692308	3.611111	3	3.88710

Preparing Test data

In [41]:

```
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)
```

Out[41]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	s
0	1129620	2	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.58
1	3321	5	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.58

In [42]:

```
# 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']
```

Before running XGBRegressor, we will tune hyperparameter using gridsearch cross validation.

In [43]:

```
start = datetime.now()

# Initialize Our first XGBoost model
model = xgb.XGBRegressor(nthread=-1)

# Perform cross validation
gscv = GridSearchCV(model,
                    param_grid = parameters,
                    scoring="neg_mean_squared_error",
                    cv = TimeSeriesSplit(n_splits=5),
                    n_jobs = -1,
                    verbose = 1)
gscv_result = gscv.fit(x_train, y_train)

# Summarize results
print("Best: %f using %s" % (gscv_result.best_score_, gscv_result.best_params_))
print()
means = gscv_result.cv_results_['mean_test_score']
stds = gscv_result.cv_results_['std_test_score']
params = gscv_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

print("\nTime Taken: ",datetime.now() - start)
```

```
Fitting 5 folds for each of 63 candidates, totalling 315 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 6 concurrent workers.
[Parallel(n_jobs=-1)]: Done  38 tasks      | elapsed:  9.2min
[Parallel(n_jobs=-1)]: Done 188 tasks      | elapsed: 81.0min
[Parallel(n_jobs=-1)]: Done 315 out of 315 | elapsed: 161.1min finished
```

```
Best: -0.745604 using {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 300}

-8.945703 (0.137510) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 100}
-6.327478 (0.104875) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 300}
-4.565558 (0.079707) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 500}
-3.380229 (0.061259) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 700}
-2.582680 (0.048964) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 900}
-2.044285 (0.040267) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 1100}
```

```
-1.679693 (0.033609) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 1300}
-8.914855 (0.117058) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 100}
-6.273837 (0.083362) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 300}
-4.499942 (0.063916) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 500}
-3.308289 (0.050271) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 700}
-2.505816 (0.039182) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 900}
-1.963202 (0.030638) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 1100}
-1.595940 (0.024345) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 1300}
-8.905508 (0.117877) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 100}
-6.247740 (0.081909) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 300}
-4.466706 (0.061004) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 500}
-3.270048 (0.044598) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 700}
-2.464795 (0.033138) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 900}
-1.920709 (0.025080) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1100}
-1.553256 (0.019707) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1300}
-2.274944 (0.043579) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 100}
-0.900042 (0.022040) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 300}
-0.821135 (0.023411) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 500}
-0.791372 (0.023354) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 700}
-0.774759 (0.023179) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 900}
-0.764954 (0.023098) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 1100}
-0.758936 (0.023106) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 1300}
-2.195542 (0.034471) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 100}
-0.824037 (0.019728) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 300}
-0.767430 (0.023112) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 500}
-0.754642 (0.023585) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 700}
-0.749866 (0.023487) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 900}
-0.747867 (0.023291) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 1100}
-0.746911 (0.023111) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 1300}
-2.154470 (0.028357) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100}
-0.795094 (0.020759) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 300}
-0.753015 (0.023486) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 500}
-0.747955 (0.023574) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 700}
-0.746591 (0.023438) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 900}
-0.745986 (0.023329) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1100}
-0.745676 (0.023278) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1300}
-0.767599 (0.023340) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 100}
-0.748327 (0.023064) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 300}
-0.747834 (0.022782) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 500}
-0.747787 (0.022720) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 700}
-0.747818 (0.022745) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 900}
-0.747848 (0.022781) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 1100}
-0.747890 (0.022826) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 1300}
-0.749688 (0.023292) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 100}
-0.746259 (0.022890) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 300}
-0.746029 (0.023047) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 500}
-0.745990 (0.023076) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 700}
-0.746224 (0.023252) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 900}
-0.746459 (0.023288) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 1100}
-0.746670 (0.023333) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 1300}
-0.747132 (0.023387) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
-0.745604 (0.023512) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 300}
-0.745664 (0.023484) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 500}
-0.746019 (0.023614) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 700}
-0.746262 (0.023712) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 900}
-0.746625 (0.023894) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 1100}
-0.747108 (0.023996) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 1300}

Time Taken:  2:41:40.855026
```

In [44]:

```python
# Create new instance of XGBRegressor with tuned hyperparameters
xgb_knn_bsl = xgb.XGBRegressor(max_depth=3,learning_rate = 0.1,n_estimators=300,nthread=-1)
xgb_knn_bsl
```

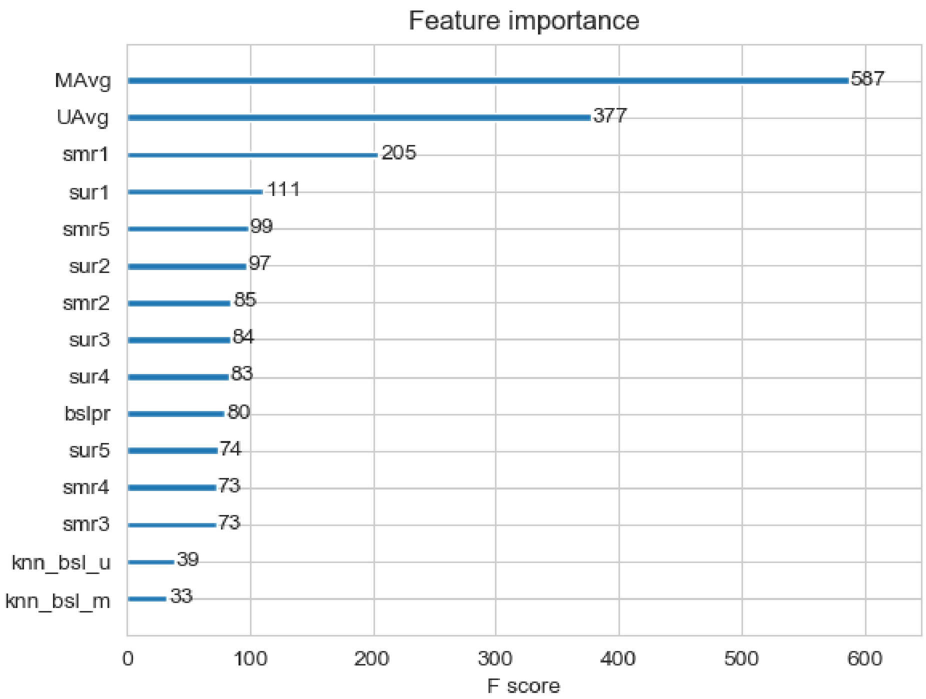
Out[44]:

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=300,
       n_jobs=1, nthread=-1, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1)
```

In [45]:

```
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 : 0:00:36.846977

Done

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

TEST DATA
------------------------------
RMSE :  1.0922009859268562
MAPE :  34.37736563892511
```



4.4.6 Matrix Factorization Techniques

4.4.6.1 SVD Matrix Factorization User Movie intractions

In [46]:

```
from surprise import SVD
```

http://surprise.readthedocs.io/en/stable/matrix_factorization.html#surprise.prediction_algorithms.matrix_factorization.SVD

- Predicted Rating :

```
- $ \large  \hat r_{ui} = \mu + b_u + b_i + q_i^Tp_u $
```

```
- $\pmb q_i$ - Representation of item(movie) in latent factor space

- $\pmb p_u$ - Representation of user in new latent factor space
```

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

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

```
- $\large \sum_{r_{ui} \in R_{train}} \left(r_{ui} - \hat{r}_{ui} \right)^2 +
```

\lambda\left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2\right) $

In [47]:

```
# initiallize 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 : 0:00:40.598189

Evaluating the model with train data..
time taken : 0:00:05.575806
---------------
Train Data
---------------
RMSE : 0.6746731413267192

MAPE : 20.05479554670084

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:01.078472
---------------
Test Data
---------------
RMSE : 1.0848131688964942

MAPE : 34.42227772904655

storing the test results in test dictionary...

---------------------------------------------
Total time taken to run this algorithm : 0:00:47.252467
```

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

In [48]:

```
from surprise import SVDpp
```

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

- Predicted Rating :

```
- $ \large \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) $
```

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

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

```
- $ \large \sum_{r_{ui} \in R_{train}} \left(r_{ui} - \hat{r}_{ui} \right)^2 +
```

\lambda\left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2 + ||y_j||^2\right) $

In [49]:

```
# initiallize 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 : 0:28:50.435547

Evaluating the model with train data..
time taken : 0:01:06.080751
---------------
Train Data
---------------
RMSE : 0.6641918784333875

MAPE : 19.24213231265533
```

```
adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:01.121147
---------------
Test Data
---------------
RMSE : 1.0854698955190794

MAPE : 34.387935054377735

storing the test results in test dictionary...

---------------------------------------------
Total time taken to run this algorithm : 0:29:57.637445
```

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

Preparing Train data

In [54]:

```python
# 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[54]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	...	smr4	smr5	UAvg	MAvg	rating	bslpr	k
0	174683	10	3.587581	5.0	5.0	3.0	4.0	4.0	3.0	5.0	...	3.0	2.0	3.882353	3.611111	5	3.877252	4
1	233949	10	3.587581	4.0	4.0	5.0	1.0	3.0	2.0	3.0	...	3.0	3.0	2.692308	3.611111	3	3.887108	3

2 rows × 21 columns

Preparing Test data

In [55]:

```python
reg_test_df['svd'] = models_evaluation_test['svd']['predictions']
reg_test_df['svdpp'] = models_evaluation_test['svdpp']['predictions']

reg_test_df.head(2)
```

Out[55]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	...	smr4	smr5	
0	1129620	2	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	...	3.587581	3.587581	3
1	3321	5	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	3.587581	...	3.587581	3.587581	3

2 rows × 21 columns

In [56]:

```python
# 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)
```

```
x_test = reg_test_df.drop(['user', 'movie', 'rating'], axis=1)
y_test = reg_test_df['rating']
```

Before running XGBRegressor, we will tune hyperparameter using gridsearch cross validation.

In [57]:

```
start = datetime.now()

# Initialize Our first XGBoost model
model = xgb.XGBRegressor(nthread=-1)

# Perform cross validation
gscv = GridSearchCV(model,
                    param_grid = parameters,
                    scoring="neg_mean_squared_error",
                    cv = TimeSeriesSplit(n_splits=5),
                    n_jobs = -1,
                    verbose = 1)
gscv_result = gscv.fit(x_train, y_train)

# Summarize results
print("Best: %f using %s" % (gscv_result.best_score_, gscv_result.best_params_))
print()
means = gscv_result.cv_results_['mean_test_score']
stds = gscv_result.cv_results_['std_test_score']
params = gscv_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

print("\nTime Taken: ",datetime.now() - start)
```

```
Fitting 5 folds for each of 63 candidates, totalling 315 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 6 concurrent workers.
[Parallel(n_jobs=-1)]: Done  38 tasks      | elapsed: 11.0min
[Parallel(n_jobs=-1)]: Done 188 tasks      | elapsed: 103.5min
[Parallel(n_jobs=-1)]: Done 315 out of 315 | elapsed: 199.7min finished
```

```
Best: -0.745669 using {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1300}

-8.945703 (0.137510) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 100}
-6.327478 (0.104875) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 300}
-4.565558 (0.079707) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 500}
-3.380229 (0.061259) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 700}
-2.582680 (0.048964) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 900}
-2.044285 (0.040267) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 1100}
-1.679693 (0.033609) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 1300}
-8.914855 (0.117058) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 100}
-6.273837 (0.083362) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 300}
-4.499942 (0.063916) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 500}
-3.308289 (0.050271) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 700}
-2.505816 (0.039182) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 900}
-1.963202 (0.030638) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 1100}
-1.595940 (0.024345) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 1300}
-8.905508 (0.117877) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 100}
-6.247740 (0.081909) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 300}
-4.466706 (0.061004) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 500}
-3.270048 (0.044598) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 700}
-2.464795 (0.033138) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 900}
-1.920709 (0.025080) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1100}
-1.553256 (0.019707) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1300}
-2.274944 (0.043579) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 100}
-0.900042 (0.022040) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 300}
-0.821135 (0.023411) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 500}
-0.791372 (0.023354) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 700}
-0.774759 (0.023179) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 900}
-0.764954 (0.023098) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 1100}
-0.758936 (0.023106) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 1300}
-2.195542 (0.034471) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 100}
-0.824037 (0.019728) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 300}
-0.767430 (0.023112) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 500}
-0.754642 (0.023585) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 700}
-0.749866 (0.023487) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 900}
```

```
-0.747871 (0.023298) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 1100}
-0.746915 (0.023118) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 1300}
-2.154470 (0.028357) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100}
-0.795094 (0.020759) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 300}
-0.753012 (0.023477) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 500}
-0.747956 (0.023548) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 700}
-0.746567 (0.023403) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 900}
-0.745967 (0.023315) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1100}
-0.745669 (0.023272) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1300}
-0.767599 (0.023340) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 100}
-0.748327 (0.023064) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 300}
-0.747839 (0.022791) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 500}
-0.747811 (0.022732) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 700}
-0.747825 (0.022746) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 900}
-0.747854 (0.022787) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 1100}
-0.747893 (0.022828) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 1300}
-0.749690 (0.023296) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 100}
-0.746257 (0.022894) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 300}
-0.746092 (0.023046) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 500}
-0.746160 (0.023070) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 700}
-0.746323 (0.023133) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 900}
-0.746476 (0.023244) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 1100}
-0.746704 (0.023367) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 1300}
-0.747126 (0.023398) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
-0.745733 (0.023520) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 300}
-0.745920 (0.023724) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 500}
-0.746206 (0.023864) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 700}
-0.746497 (0.023963) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 900}
-0.746857 (0.024106) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 1100}
-0.747307 (0.024252) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 1300}

Time Taken:  3:22:38.392183
```

In [63]:

```
# Create new instance of XGBRegressor with tuned hyperparameters
xgb_final = xgb.XGBRegressor(max_depth=3,learning_rate = 0.01,n_estimators=1300,nthread=-1)
xgb_final
```

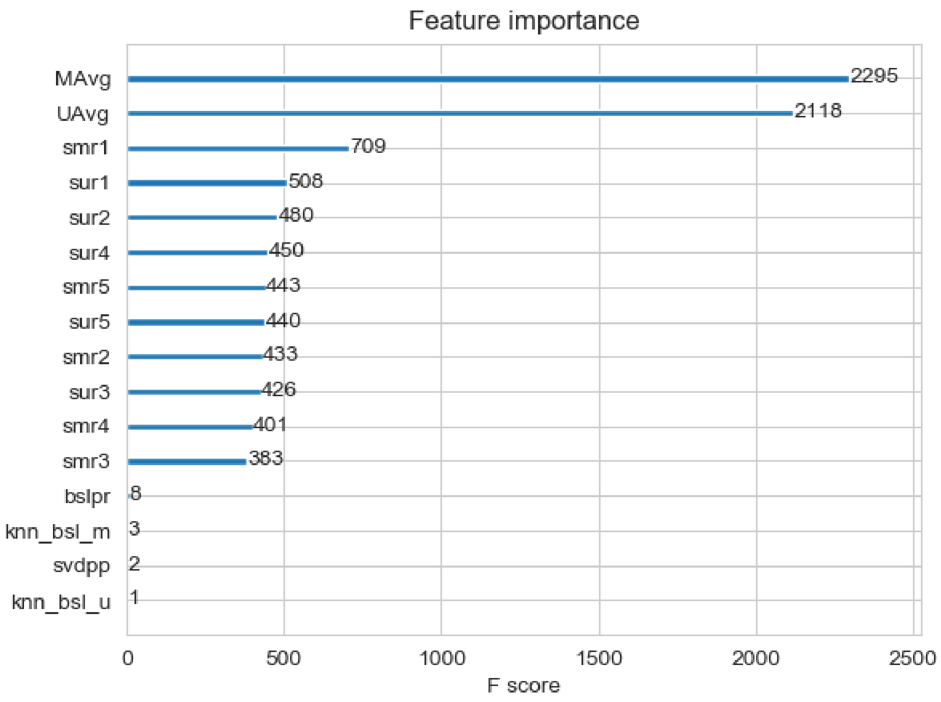
Out[63]:

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.01, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=1300,
       n_jobs=1, nthread=-1, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1)
```

In [64]:

```
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 : 0:03:04.821340

Done

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

TEST DATA
------------------------------
RMSE :  1.0897126087679558
MAPE :  34.50899275985467
```



4.4.8 XgBoost with Surprise Baseline + Surprise KNNbaseline + MF Techniques

In [65]:

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

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

Before running XGBRegressor, we will tune hyperparameter using gridsearch cross validation.

In [66]:

```
start = datetime.now()

# Initialize Our first XGBoost model
model = xgb.XGBRegressor(nthread=-1)

# Perform cross validation
gscv = GridSearchCV(model,
                    param_grid = parameters,
                    scoring="neg_mean_squared_error",
                    cv = TimeSeriesSplit(n_splits=5),
                    n_jobs = -1,
                    verbose = 1)
gscv_result = gscv.fit(x_train, y_train)

# Summarize results
print("Best: %f using %s" % (gscv_result.best_score_, gscv_result.best_params_))
print()
means = gscv_result.cv_results_['mean_test_score']
stds = gscv_result.cv_results_['std_test_score']
params = gscv_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

print("\nTime Taken: ",datetime.now() - start)
```

```
Fitting 5 folds for each of 63 candidates, totalling 315 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 6 concurrent workers.
[Parallel(n_jobs=-1)]: Done  38 tasks      | elapsed:  5.2min
[Parallel(n_jobs=-1)]: Done 188 tasks      | elapsed: 47.3min
[Parallel(n_jobs=-1)]: Done 315 out of 315 | elapsed: 91.9min finished
```

```
Best: -1.173157 using {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 700}

-8.968738 (0.138065) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 100}
-6.397479 (0.117359) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 300}
-4.674276 (0.099791) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 500}
-3.519387 (0.085075) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 700}
-2.745412 (0.072920) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 900}
-2.226718 (0.063052) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 1100}
-1.879120 (0.055187) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 1300}
-8.968667 (0.138093) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 100}
-6.397357 (0.117353) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 300}
-4.674108 (0.099788) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 500}
-3.519203 (0.085139) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 700}
-2.745237 (0.073044) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 900}
-2.226575 (0.063197) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 1100}
-1.878999 (0.055327) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 1300}
-8.968640 (0.138100) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 100}
-6.397309 (0.117409) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 300}
-4.674021 (0.099911) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 500}
-3.519134 (0.085235) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 700}
-2.745181 (0.073104) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 900}
-2.226509 (0.063282) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1100}
-1.878937 (0.055417) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1300}
-2.448580 (0.067498) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 100}
-1.195929 (0.033308) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 300}
-1.173539 (0.032246) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 500}
-1.173157 (0.032223) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 700}
-1.173158 (0.032223) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 900}
-1.173163 (0.032224) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 1100}
-1.173166 (0.032224) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 1300}
-2.448423 (0.067641) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 100}
-1.195906 (0.033345) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 300}
-1.173554 (0.032267) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 500}
-1.173199 (0.032247) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 700}
-1.173220 (0.032246) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 900}
-1.173249 (0.032250) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 1100}
-1.173275 (0.032255) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 1300}
-2.448359 (0.067716) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100}
-1.195910 (0.033385) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 300}
-1.173606 (0.032262) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 500}
-1.173280 (0.032236) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 700}
-1.173325 (0.032227) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 900}
-1.173385 (0.032238) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1100}
-1.173443 (0.032254) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1300}
-1.173160 (0.032223) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 100}
-1.173193 (0.032219) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 300}
-1.173213 (0.032223) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 500}
-1.173228 (0.032221) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 700}
-1.173245 (0.032222) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 900}
-1.173260 (0.032221) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 1100}
-1.173274 (0.032221) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 1300}
-1.173259 (0.032249) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 100}
-1.173556 (0.032309) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 300}
-1.173909 (0.032384) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 500}
-1.174260 (0.032447) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 700}
-1.174601 (0.032535) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 900}
-1.174923 (0.032614) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 1100}
-1.175277 (0.032706) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 1300}
-1.173397 (0.032262) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
-1.174046 (0.032359) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 300}
-1.174797 (0.032496) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 500}
-1.175569 (0.032685) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 700}
-1.176248 (0.032816) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 900}
-1.176901 (0.032991) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 1100}
-1.177422 (0.033104) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 1300}

Time Taken:  1:32:19.580100
```

In [67]:

```
# Create new instance of XGBRegressor with tuned hyperparameters
xgb_all_models = xgb.XGBRegressor(max_depth=1,learning_rate = 0.01,n_estimators=700,nthread=-1)
xgb_all_models
```

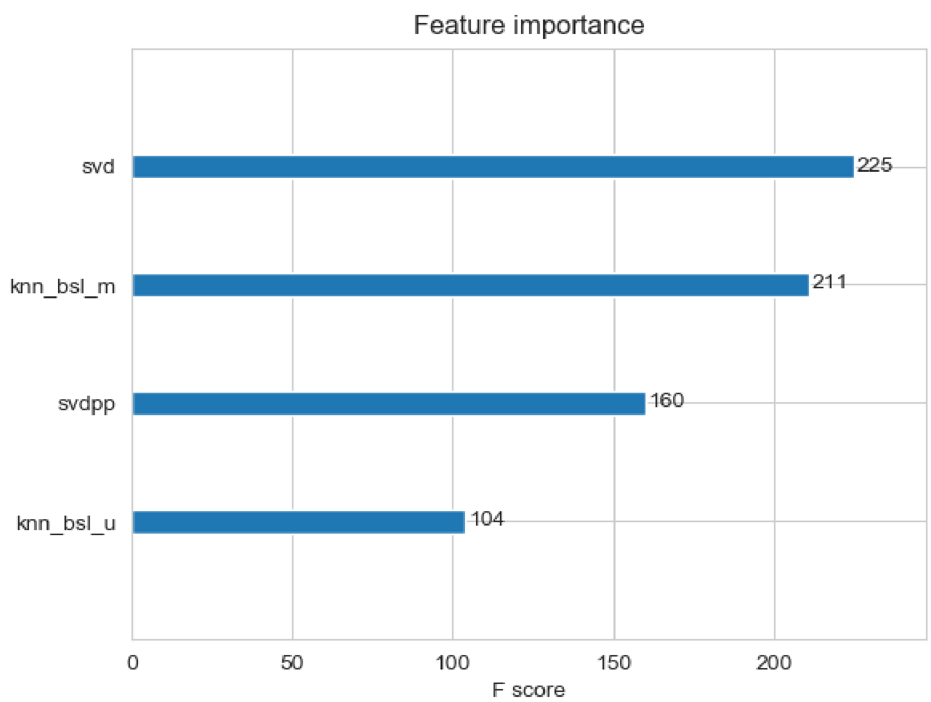
Out[67]:

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.01, max_delta_step=0,
       max_depth=1, min_child_weight=1, missing=None, n_estimators=700,
       n_jobs=1, nthread=-1, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1)
```

In [68]:

```
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 : 0:00:23.082510

Done

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

TEST DATA
------------------------------
RMSE :  1.0935465877256518
MAPE :  34.996438085718346
```



4.5 Comparision between all models

In [69]:

```
# Saving our TEST_RESULTS into a dataframe so that you don't have to run it again
pd.DataFrame(models_evaluation_test).to_csv('sample/small/small_sample_results.csv')
models = pd.read_csv('sample/small/small_sample_results.csv', index_col=0)
models.loc['rmse'].sort_values()
```

Out[69]:

```
svd                 1.0848131688964942
knn_bsl_u           1.0850618463554647
bsl_algo             1.085201374793734
knn_bsl_m           1.0852678745012594
svdpp               1.0854698955190794
xgb_final           1.0897126087679558
xgb_bsl             1.0897253482697786
first_algo          1.0904043649061108
xgb_knn_bsl         1.0922009859268562
xgb_all_models      1.0935465877256518
Name: rmse, dtype: object
```

Conclusion

> 1. Due to high computational cost, I have completed this case study on (25000,3000) training dataset and (20000,1000) testing dataset.
> 2. Similar approach is followed as mentioned in this research paper, [https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf](https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf)
> 3. Every regressor model is hyper tuned for optimal parameters.
> 4. SVD model showed good result among all the models we tried.
> 5. Small decrease in 'RMSE' score is observed, but this can be drastically improved by using the whole dataset for modeling.(Not feasible at the moment)