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 ahs 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. Ea
ch subsequent line in the file corresponds to a rating from a customer and its
 date in the following format:

CustomerID,Rating,Date


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

2.1.2 Example Data point

```
1:
1488844,3,2005-09-06
822109,5,2005-05-13
885013,4,2005-10-19
30878,4,2005-12-26
823519,3,2004-05-03
893988,3,2005-11-17
124105,4,2004-08-05
1248029,3,2004-04-22
```

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

```
1927580,4,2004-11-08
716874,5,2005-05-06
4326,4,2005-10-29
```

2.2 Mapping the real world problem to a Machine Learning Problem

2.2.1 Type of Machine Learning Problem

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

2.2.2 Performance metric

- Mean Absolute Percentage Error: https://en.wikipedia.org/wiki/Mean_absolute_percentage_error
- 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 [223]:

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

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

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

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

3. Exploratory Data Analysis

3.1 Preprocessing

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

In [3]:

```
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 fil
    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 : 0:00:00.000302
```

In [4]:

```
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..

/opt/anaconda3/lib/python3.7/site-packages/IPython/core/interactiveshell.p
y:3051: DtypeWarning: Columns (0) have mixed types. Specify dtype option o
n import or set low_memory=False.
  interactivity=interactivity, compiler=compiler, result=result)

Done.

Sorting the dataframe by date..
Done..
```

In [5]:

```
df.head()
```

Out[5]:

	movie	user	rating	date
0	9211:	NaN	NaN	NaT
1	1277134	1.0	2003-12-02	NaT
2	2435457	2.0	2005-06-01	NaT
3	2338545	3.0	2001-02-17	NaT
4	2218269	1.0	2002-12-27	NaT

In [6]:

```
# removing the : in the movie column
import regex as re
df.movie=df.movie.astype(str).apply(lambda x:re.sub(':','',x))
```

In [7]:

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

Out[7]:

```
count       22601629
unique          2181
top       2005-01-19
freq          163001
Name: rating, dtype: object
```

3.1.2 Checking for NaN values

In [8]:

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

```
No of Nan values in our dataframe :  3
```

3.1.3 Removing Duplicates

In [9]:

```
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 11222273 duplicate rating entries in the data..
```

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

In [10]:

```
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.astype(int))))
```

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

Total no of ratings : 22605786
Total No of Users   : 4162
Total No of movies  : 478107
```

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

In [11]:

```
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 [12]:

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

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

In [13]:

```
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

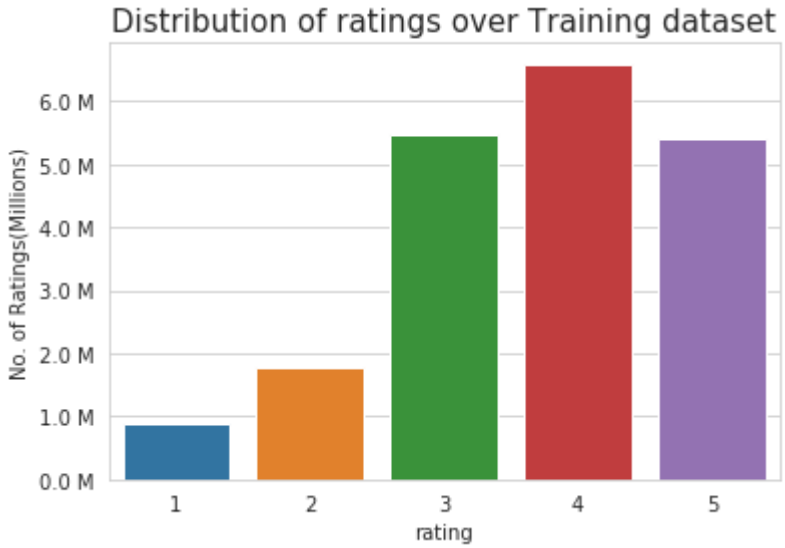
In [14]:

```
# 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 [15]:

```
%matplotlib inline
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 [16]:

```
# 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()
```

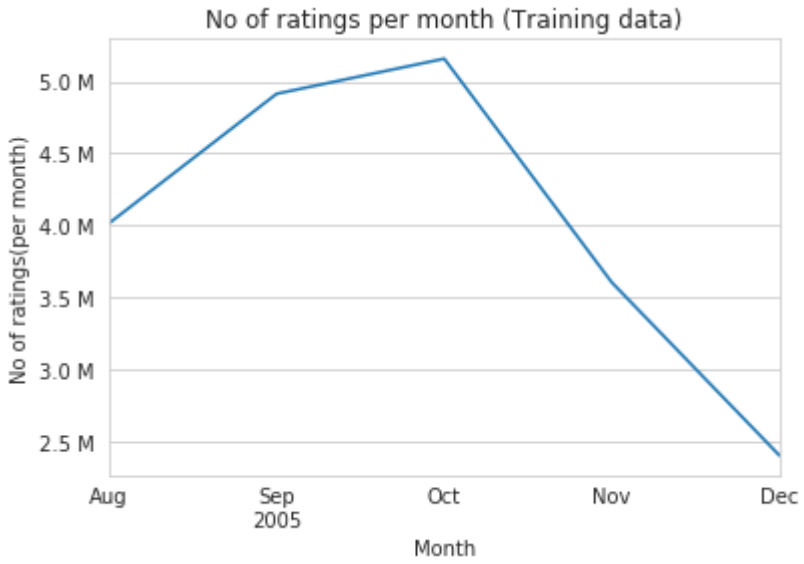
Out[16]:

	movie	user	rating	date	day_of_week
20096097	8993	2183787	4	2005-12-31	Saturday
20096098	7430	258170	4	2005-12-31	Saturday
20096099	8467	1534359	5	2005-12-31	Saturday
20096100	10168	2543295	2	2005-12-31	Saturday
20096101	4736	1346243	5	2005-12-31	Saturday

3.3.2 Number of Ratings per a month

In [17]:

```
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 [18]:

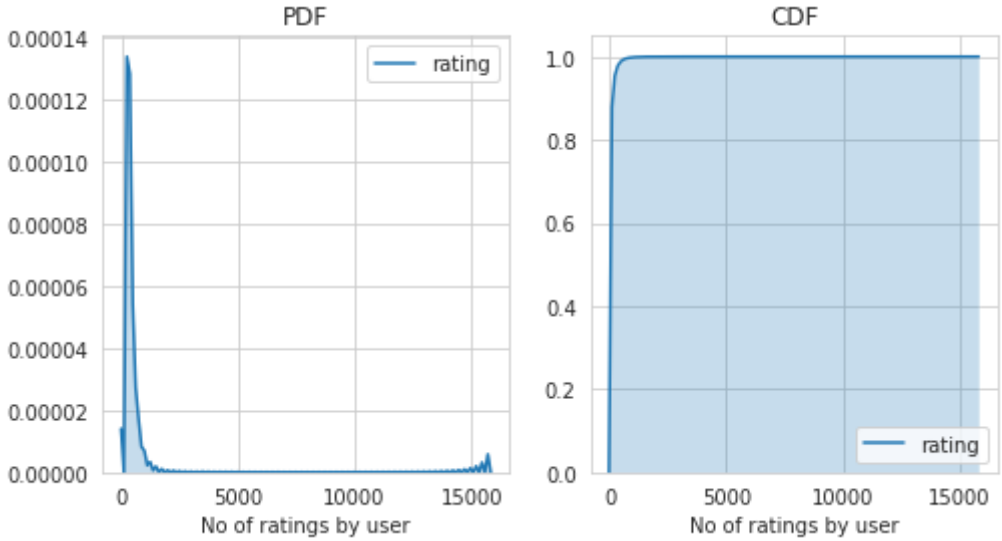
```
no_of_rated_movies_per_user = train_df.groupby(by='user')['rating'].count().sort_values(

no_of_rated_movies_per_user.head()
```

Out[18]:

```
user
1664010    15813
2118461    14831
1473980     6790
1710658     5533
2147527     5357
Name: rating, dtype: int64
```

In [19]:

```
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 [20]:

```
no_of_rated_movies_per_user.describe()
```

Out[20]:

```
count    349312.000000
mean         57.530523
std         117.498700
min           1.000000
25%           7.000000
50%          25.000000
75%          59.000000
max       15813.000000
Name: rating, dtype: float64
```

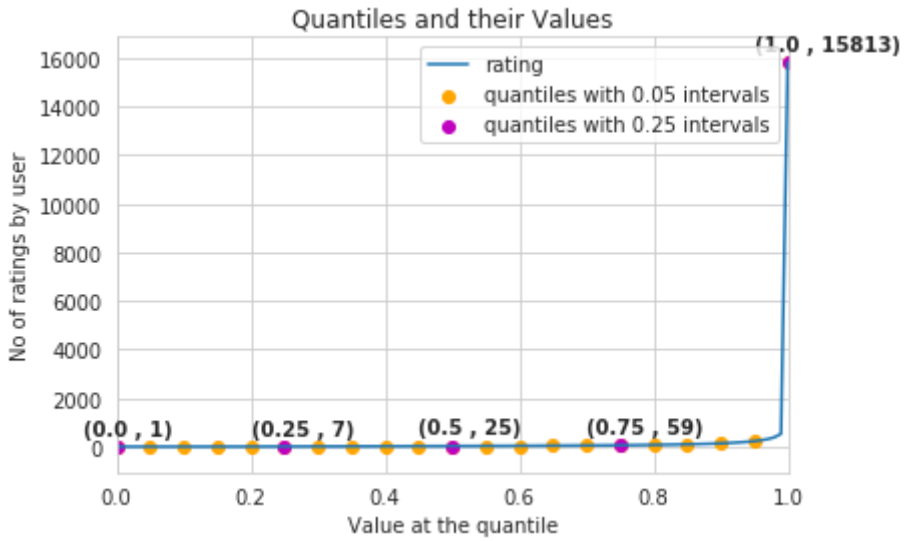
> *There, is something interesting going on with the quantiles..*

In [21]:

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

In [22]:

```
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="quantile
# quantiles with 0.25 difference
plt.scatter(x=quantiles.index[::25], y=quantiles.values[::25], c='m', label = "quantiles
plt.ylabel('No of ratings by user')
plt.xlabel('Value at the quantile')
plt.legend(loc='best')

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


plt.show()
```



In [23]:

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

Out[23]:

```
0.00        1
0.05        1
0.10        2
0.15        3
0.20        4
0.25        7
0.30       10
0.35       13
0.40       17
0.45       20
0.50       25
0.55       29
0.60       35
0.65       41
0.70       49
0.75       59
0.80       73
0.85       97
0.90      137
0.95      227
1.00    15813
Name: rating, dtype: int64
```

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

In [24]:

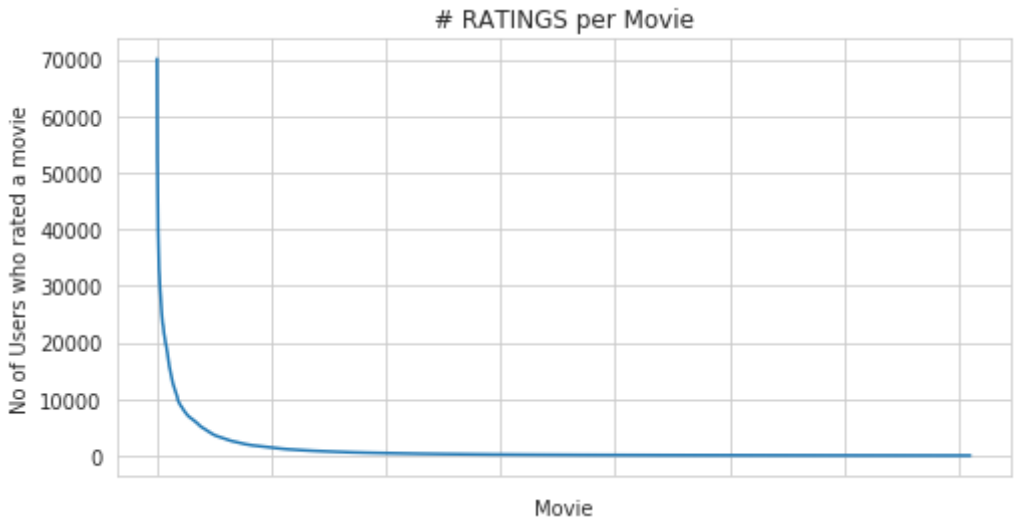
```
print('\n No of ratings at last 5 percentile : {}\n'.format(sum(no_of_rated_movies_per_u
```

```
 No of ratings at last 5 percentile : 1415
```

3.3.4 Analysis of ratings of a movie given by a user

In [25]:

```
no_of_ratings_per_movie = train_df.groupby(by='movie')['rating'].count().sort_values(as

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.**

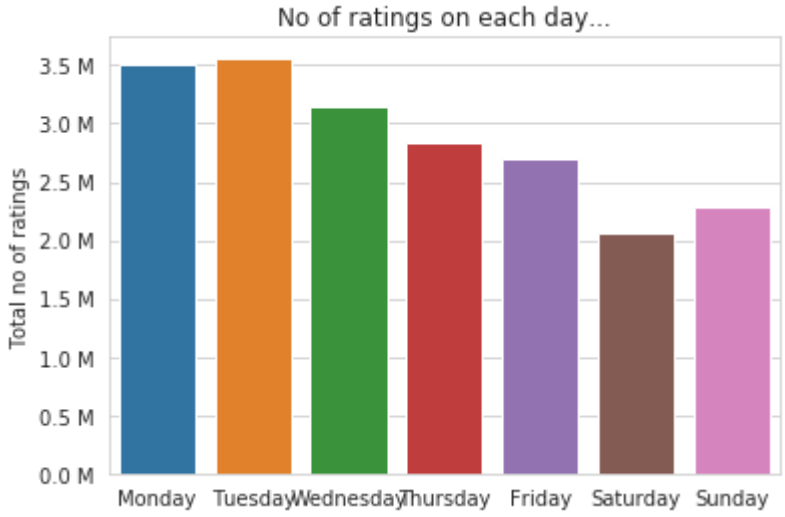
  - There are some movies (which are very popular) which are rated by huge number of users.

  - But most of the movies(like 90%) got some hundereds of ratings.

3.3.5 Number of ratings on each day of the week

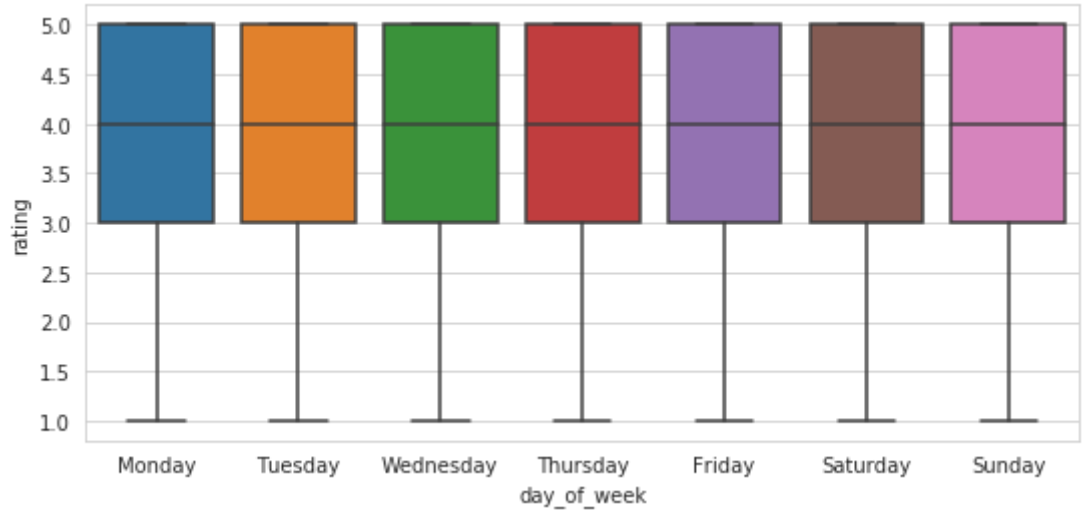
In [26]:

```
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 [27]:

```
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)
```




```
0:00:17.633996
```

In [28]:

```
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       3.687586
Monday       3.678614
Saturday     3.703333
Sunday       3.698910
Thursday     3.693734
Tuesday      3.681093
Wednesday    3.695524
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 [35]:

```
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.valu
                                             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)
```

```
We are creating sparse_matrix from the dataframe..
Done. It's shape is : (user, movie) :  (2649430, 17771)
Saving it into disk for furthur usage..
Done..

0:00:18.158348
```

The Sparsity of Train Sparse Matrix

In [36]:

```
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 : 99.95731772988694 %
```

3.3.6.2 Creating sparse matrix from test data frame

In [37]:

```
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)
```

```
We are creating sparse_matrix from the dataframe..
Done. It's shape is : (user, movie) :  (2649430, 17771)
Saving it into disk for furthur usage..
Done..

0:00:17.929615
```

The Sparsity of Test data Matrix

In [38]:

```
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 [39]:

```
# 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 [40]:

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

Out[40]:

```
{'global': 3.689887073622536}
```

3.3.7.2 finding average rating per user

In [41]:

```
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 : 3.440677966101695
```

3.3.7.3 finding average rating per movie

In [42]:

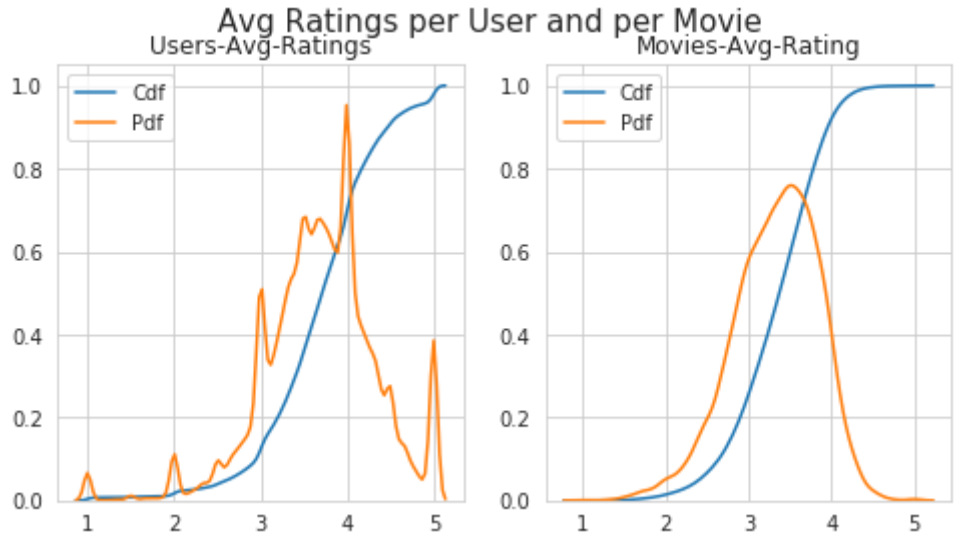
```
train_averages['movie'] =  get_average_ratings(train_sparse_matrix, of_users=False)
print('\n AVerage rating of movie 15 :',train_averages['movie'][15])
```

```
 AVerage rating of movie 15 : 3.1333333333333333
```

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

In [43]:

```
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)
```




```
0:00:30.260287
```

3.3.8 Cold Start problem

3.3.8.1 Cold Start problem with Users

In [44]:

```
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_us
```

```
Total number of Users  : 4162

Number of Users in Train data : 349312

No of Users that didn't appear in train data: -345150(-8292.89 %)
```

> 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 [45]:

```
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_mo
```

```
Total number of Movies  : 478107

Number of Users in Train data : 17757

No of Movies that didn't appear in train data: 460350(96.29 %)
```

> 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)

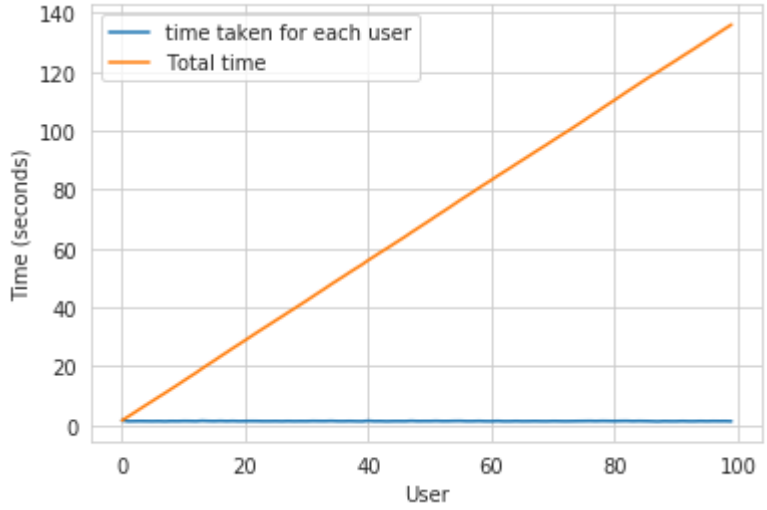
In [46]:

```
from sklearn.metrics.pairwise import cosine_similarity


def compute_user_similarity(sparse_matrix, compute_for_few=False, top = 100, verbose=Fal
                            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 matrice
    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)), ti
```

In [47]:

```
start = datetime.now()
u_u_sim_sparse, _ = compute_user_similarity(train_sparse_matrix, compute_for_few=True, t
                                                    verbose=True)
print("-"*100)
print("Time taken :",datetime.now()-start)
```

```
Computing top 100 similarities for each user..
computing done for 20 users [  time elapsed : 0:00:27.459177  ]
computing done for 40 users [  time elapsed : 0:00:54.545491  ]
computing done for 60 users [  time elapsed : 0:01:21.930028  ]
computing done for 80 users [  time elapsed : 0:01:49.031488  ]
computing done for 100 users [  time elapsed : 0:02:16.023078  ]
Creating Sparse matrix from the computed similarities
```




```
--------------------------------------------------------------------------------
--------------------------
Time taken : 0:02:18.512668
```

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.
- $405041 \times 8.88 = 3596764.08 \text{ sec} = 59946.068 \text{ min} = 999.101133333 \text{ hours} = 41.629213889 \text{ days}...$
  - Even if we run on 4 cores parallelly (a typical system now a days), It will still take almost **10 and 1/2** days.

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

In [48]:

```
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)
```

```
0:12:07.294217
```

Here,

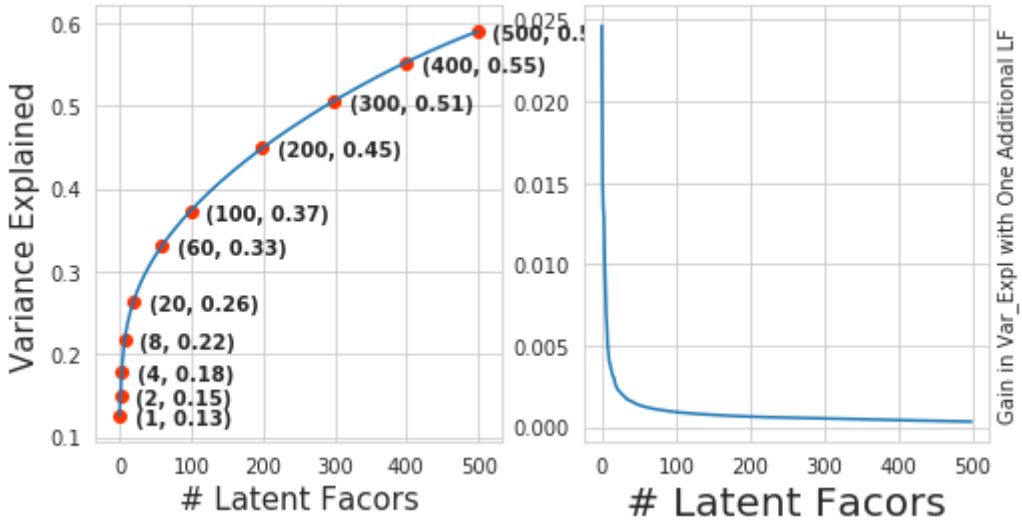
- $\sum \longleftarrow$ (netflix_svd.**singular_values_**)
- $\bigvee^T \longleftarrow$ (netflix_svd.**components_**)
- $\bigcup$ is not returned. instead **Projection_of_X** onto the new vectorspace is returned.
- It uses **randomized svd** internally, which returns **All 3 of them saperately**. Use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

In [49]:

```
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
                 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, 0.13)
(2, 0.15)
(4, 0.18)
(8, 0.22)
(20, 0.26)
(60, 0.33)
(100, 0.37)
(200, 0.45)
(300, 0.51)
(400, 0.55)
(500, 0.59)
```

> 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)
```

```
0:00:12.970803
```

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
```

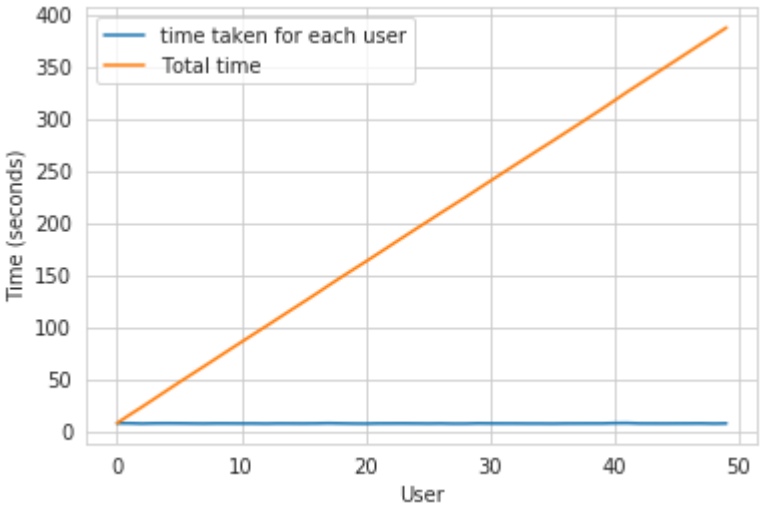
Out[55]:

```
(2649430, 500)
```

In [56]:

```
start = datetime.now()
trunc_u_u_sim_matrix, _ = compute_user_similarity(trunc_sparse_matrix, compute_for_few=T
                                                  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 : 0:01:18.204412  ]
computing done for 20 users [  time elapsed : 0:02:35.670267  ]
computing done for 30 users [  time elapsed : 0:03:52.767811  ]
computing done for 40 users [  time elapsed : 0:05:09.869585  ]
computing done for 50 users [  time elapsed : 0:06:27.973059  ]
Creating Sparse matrix from the computed similarities
```




```
--------------------------------------------------
time: 0:06:46.604198
```

: 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.
- $405041 \times 12.18 ==== 4933399.38 \text{ sec} ==== 82223.323 \text{ min} ==== 1370.388716667 \text{ hours} ==== 5$
  - 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 comp
uted or not..
- ***If not*** :
    - Compute top (let's just say, 1000) most similar users for this given use
r, and add this to our datastructure, so that we can just access it(similar use
rs) 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 com
puted 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 ( recomp
ute 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 [57]:

```
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 seems you don't have that file. Computing movie_movie similarity...
Done..
Saving it to disk without the need of re-computing it again..
Done..
It's a  (17771, 17771)  dimensional matrix
0:05:52.738955
```

In [58]:

```
m_m_sim_sparse.shape
```

Out[58]:

```
(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 [59]:

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

In [60]:

```
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]
```

```
0:00:31.018688
```

Out[60]:

```
array([14893, 10220, 13000,  8532, 10400,  6933, 17731, 17545,   219,
        1368,    99, 16761,   415, 17566,   218,   882,   464,  1864,
       15056,  7458,  7012,  8899,  6116, 16625, 15439, 12705, 16684,
       13156,  5370,  2541, 13395,  1003, 11809, 10544,  6986, 10862,
        6838,   669, 10332, 15050,  8692, 15878, 11539, 11943,  1383,
        3261, 14956,  8309,  9221, 15789,  8222,  4997, 15589,  4332,
       15763, 15910,  1474,  9956, 16528,  2818,  6410,  8374, 11016,
        8291, 15391,  6217, 17542, 12199,  4734, 15030,  1910,  6209,
        8153,  9050, 10687,  1900,  8279, 10652, 13631, 10323, 10258,
       15448, 13525,  9372, 14679, 11742,  4236, 17477, 10427, 10974,
        1949, 12700, 11115, 13183,  1671, 16817, 10753,  1772,  4391,
        9358])
```

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 [62]:

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

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

movie_titles.head()
```

```
Type conversion took: 108.75 ms
Parser memory cleanup took: 0.01 ms
Tokenization took: 80.88 ms
Type conversion took: 114.27 ms
Parser memory cleanup took: 0.01 ms
Tokenization took: 63.43 ms
Type conversion took: 128.45 ms
Parser memory cleanup took: 0.01 ms
Tokenization took: 103.08 ms
Type conversion took: 110.95 ms
Parser memory cleanup took: 0.01 ms
Tokenization took: 146.25 ms
Type conversion took: 118.42 ms
Parser memory cleanup took: 0.01 ms
Tokenization took: 57.15 ms
Type conversion took: 105.70 ms
Parser memory cleanup took: 0.02 ms
Tokenization took: 62.61 ms
Type conversion took: 109.03 ms
```

Similar Movies for 'Vampire Journals'

In [67]:

```
mv_id = 67

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..".fo
```

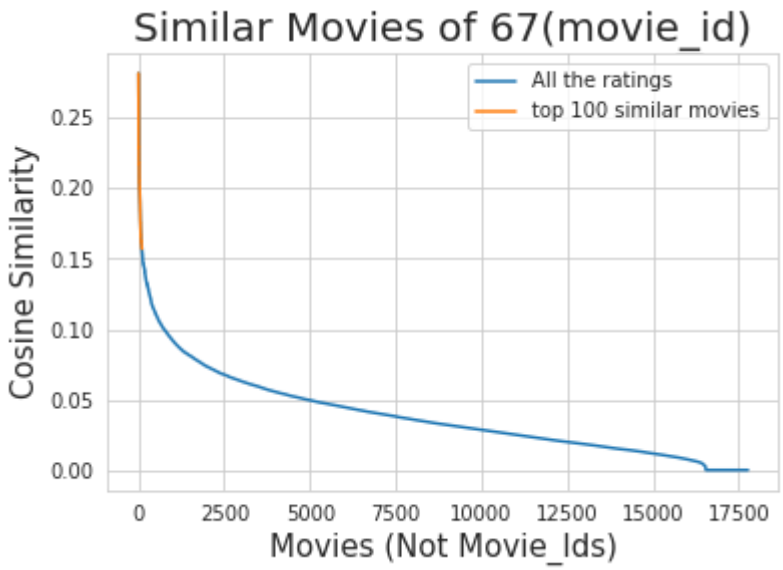
```
It has 19 Ratings from users.

We have 16537 movies which are similarto this  and we will get only top mo
st..
```

In [68]:

```
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
                                               # and return its indices(movie_ids)
```

In [69]:

```
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 [75]:

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

Out[75]:

movie_id	year_of_release	title
107338	3.0	2005-10-03
127401	5.0	2005-03-17
1654464	3.0	2004-01-16
425898	3.0	2005-08-25
67658	3.0	2004-05-03
1279971	4.0	2003-07-18
293360	3.0	2005-01-19
2264752	4.0	2003-09-12
760211	3.0	2004-06-25
1558907	3.0	2004-12-18

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

4. Machine Learning Models

In [76]:

```
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[mas
                                             shape=(max(sample_users)+1, max(sample_movi

    if verbose:
        print("Sampled Matrix : (users, movies) -- ({} {})".format(len(sample_users), le
        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 [78]:

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

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

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

Sampled Matrix : (users, movies) -- (10000 1000)
Sampled Matrix : Ratings -- 36017
Saving it into disk for furthur usage..
Done..

0:00:17.869007
```

4.1.2 Build sample test data from the test data

In [81]:

```
start = datetime.now()

path = "sample_test_sparse_matrix.npz"
if os.path.isfile(path):
    print("It is present in your pwd, getting it from disk....")
    # just get it from the disk instead of computing it
    sample_test_sparse_matrix = sparse.load_npz(path)
    print("DONE..")
else:
    # get 5k users and 500 movies from available data
    sample_test_sparse_matrix = get_sample_sparse_matrix(test_sparse_matrix, no_users=50
                                            path = "sample_test_sparse_matrix.npz")
print(datetime.now() - start)
```

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

Sampled Matrix : (users, movies) -- (5000 500)
Sampled Matrix : Ratings -- 7333
Saving it into disk for furthur usage..
Done..

0:00:17.960582
```

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

In [82]:

```
sample_train_averages = dict()
```

4.2.1 Finding Global Average of all movie ratings

In [83]:

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

Out[83]:

```
{'global': 3.6071854957381237}
```

4.2.2 Finding Average rating per User

In [100]:

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

```
Average rating of user  : {7: 5.0, 126: 4.6, 268: 4.333333333333333, 147
5: 3.0, 1809: 3.8095238095238093, 2040: 5.0, 2050: 3.090909090909091, 29
75: 4.0, 3300: 3.625, 3321: 2.185567010309278, 3952: 3.0, 3959: 4.25, 40
16: 3.5, 4422: 3.6, 4580: 4.5, 4752: 4.0, 4796: 4.5, 4882: 2.0, 5454: 4.
0, 5474: 4.0, 5827: 4.375, 5851: 5.0, 6181: 4.5, 7419: 3.0, 7851: 4.5, 7
921: 4.833333333333333, 8043: 4.0, 8367: 3.2, 8427: 3.6666666666666665,
8439: 4.0, 8623: 2.0, 10380: 2.625, 10670: 3.0, 10897: 3.636363636363636
2, 11310: 3.0, 11452: 2.0, 11676: 3.0714285714285716, 12217: 4.0, 13717:
3.6666666666666665, 14060: 4.5, 14158: 4.5, 14179: 2.0, 14204: 3.3448275
862068964, 14326: 3.5, 14793: 4.0, 15314: 4.0, 16168: 2.0, 17189: 4.25,
17330: 2.0, 17872: 4.5, 18309: 3.0, 19117: 3.5517241379310347, 19832: 2.
0, 19917: 2.923076923076923, 20295: 1.6666666666666667, 20607: 4.0, 2122
9: 3.0, 21271: 4.0, 21851: 2.0, 21923: 3.5, 22338: 3.5, 22964: 3.6666666
666666665, 23354: 4.0, 23404: 4.0, 24811: 4.0, 27242: 2.0, 28572: 3.5, 2
8668: 3.0, 29719: 3.8947368421052633, 29777: 3.0, 30666: 3.0, 30976: 3.
0, 31164: 3.0, 31458: 4.4, 31829: 4.5, 32037: 3.0, 32243: 2.75, 32389:
5.0, 32426: 2.6666666666666665, 32668: 3.6666666666666665, 33484: 4.0, 3
4211: 4.0, 34910: 3.0, 35315: 3.0, 35898: 3.3333333333333335, 35994: 3.3
```

4.2.3 Finding Average rating per Movie

In [99]:

```
sample_train_averages['movie'] =  get_average_ratings(sample_train_sparse_matrix, of_use
print('\n AVerage rating of movie :',sample_train_averages['movie'])
```

```
 AVerage rating of movie : {71: 3.2857142857142856, 90: 4.5, 111: 3.2473
11827956989, 155: 3.0, 188: 2.875, 208: 3.7, 251: 3.75, 262: 3.648648648
6486487, 280: 3.0, 293: 4.75, 308: 2.5, 309: 4.0, 348: 3.775510204081632
6, 394: 3.4, 420: 3.7916666666666665, 425: 3.75, 466: 3.375, 468: 3.7078
18930041152, 482: 3.7577319587628866, 587: 3.5, 595: 3.5, 603: 4.5, 607:
3.5389473684210526, 608: 2.0, 614: 3.0, 644: 2.0, 658: 3.69322709163346
6, 666: 3.3333333333333335, 677: 3.7777777777777777, 678: 5.0, 681: 3.44
44444444444446, 690: 4.090909090909091, 695: 3.0, 705: 3.447368421052631
4, 746: 3.5660377358490565, 750: 3.617021276595745, 752: 4.0508474576271
185, 755: 4.0, 766: 2.0, 775: 4.0, 780: 5.0, 793: 2.5, 808: 3.6, 811: 3.
1875, 823: 3.0, 838: 3.0, 859: 3.357142857142857, 890: 2.5, 943: 3.8, 95
7: 3.6, 962: 3.3620689655172415, 985: 3.9483282674772036, 997: 5.0, 102
0: 4.327102803738318, 1033: 3.0, 1103: 3.75, 1114: 2.75, 1134: 3.4931506
84931507, 1140: 3.111111111111111, 1163: 3.6363636363636362, 1164: 2.666
6666666666665, 1189: 2.0, 1201: 3.8461538461538463, 1209: 3.666666666666
6665, 1241: 3.0, 1247: 3.0, 1318: 3.0, 1326: 3.0, 1381: 3.0, 1391: 2.5,
1399: 3.7142857142857144, 1426: 2.6666666666666665, 1429: 3.0, 1435: 4.2
13793103448276, 1445: 3.5, 1471: 3.1, 1506: 4.363636363636363, 1551: 3.
```

4.3 Featurizing data

In [101]:

```
print('\n No of ratings in Our Sampled train matrix is : {}\n'.format(sample_train_spars
print('\n No of ratings in Our Sampled test  matrix is : {}\n'.format(sample_test_sparse
```

```
 No of ratings in Our Sampled train matrix is : 36017


 No of ratings in Our Sampled test  matrix is : 7333
```

4.3.1 Featurizing data for regression problem

4.3.1.1 Featurizing train data

In [102]:

```
# get users, movies and ratings from our samples train sparse matrix
sample_train_users, sample_train_movies, sample_train_ratings = sparse.find(sample_trai
```

In [104]:

```
############################################################
# It took me almost 10 hours to prepare this train dataset.#
############################################################
start = datetime.now()
if os.path.isfile('reg_train.csv'):
    print("File already exists you don't have to prepare again..." )
else:
    print('preparing {} tuples for the dataset..\n'.format(len(sample_train_ratings)))
    with open('reg_train.csv', mode='w') as reg_data_file:
        count = 0
        for (user, movie, rating)  in zip(sample_train_users, sample_train_movies, samp
            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_
            top_sim_users = user_sim.argsort()[::-1][1:] # we are ignoring 'The User' fr
            # get the ratings of most similar users for this movie
            top_ratings = sample_train_sparse_matrix[top_sim_users, movie].toarray().ra
            # 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 - le
        #     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_
            top_sim_movies = movie_sim.argsort()[::-1][1:] # we are ignoring 'The User'
            # get the ratings of most similar movie rated by this user..
            top_ratings = sample_train_sparse_matrix[user, top_sim_movies].toarray().ra
            # 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(t
        #     print(top_sim_movies_ratings, end=" : -- ")

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

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

            # add rows to the file opened..
            reg_data_file.write(','.join(map(str, row)))
            reg_data_file.write('\n')
            if (count)%10000 == 0:
```

```
                # print(','.join(map(str, row)))
                print("Done for {} rows----- {}".format(count, datetime.now() - start))


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

```
preparing 36017 tuples for the dataset..

Done for 10000 rows----- 0:38:07.607409
Done for 20000 rows----- 1:16:15.428604
Done for 30000 rows----- 1:54:17.930351
2:17:09.837003
```

Reading from the file to make a Train_dataframe

In [106]:

```
reg_train = pd.read_csv('reg_train.csv', names = ['user', 'movie', 'GAvg', 'sur1', 'sur2
reg_train.head()
```

Out[106]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	
0	808635	71	3.607185	3.0	4.0	4.0	3.0	3.0	3.0	4.0	3.0	3.75	3.75	3
1	898730	71	3.607185	3.0	4.0	4.0	3.0	5.0	1.0	4.0	4.0	3.00	4.00	3
2	941866	71	3.607185	3.0	4.0	5.0	3.0	3.0	5.0	4.5	4.5	4.50	4.50	4
3	1280761	71	3.607185	4.0	3.0	5.0	4.0	3.0	5.0	5.0	5.0	5.00	4.00	3
4	1386925	71	3.607185	3.0	4.0	4.0	3.0	5.0	4.0	4.0	4.0	4.00	3.00	3

- **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 [107]:

```
# get users, movies and ratings from the Sampled Test
sample_test_users, sample_test_movies, sample_test_ratings = sparse.find(sample_test_spa
```

In [108]:

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

Out[108]:

```
3.6071854957381237
```

In [165]:

```
start = datetime.now()

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

    print('preparing {} tuples for the dataset..\n'.format(len(sample_test_ratings)))
    with open('reg_test.csv', mode='w') as reg_data_file:
        count = 0
        for (user, movie, rating)  in zip(sample_test_users, sample_test_movies, sample_
            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_tr
                top_sim_users = user_sim.argsort()[::-1][1:] # we are ignoring 'The User
                # get the ratings of most similar users for this movie
                top_ratings = sample_train_sparse_matrix[top_sim_users, movie].toarray()
                # 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
                # 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 j
                ########## Cold STart Problem ##########
                top_sim_users_ratings.extend([sample_train_averages['global']]*(5 - len(
                #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, sam
                top_sim_movies = movie_sim.argsort()[::-1][1:] # we are ignoring 'The Us
                # get the ratings of most similar movie rated by this user..
                top_ratings = sample_train_sparse_matrix[user, top_sim_movies].toarray()
                # 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-l
                #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(t
                #print(top_sim_movies_ratings)
            except :
                raise

            #-----------------prepare the row to be stores in a file-----------------#
            row = list()
            # add usser and movie name first
            row.append(user)
```

```
            row.append(movie)
            row.append(sample_train_averages['global']) # first feature
            #print(row)
            # next 5 features are similar_users "movie" ratings
            row.extend(top_sim_users_ratings)
            #print(row)
            # next 5 features are "user" ratings for similar_movies
            row.extend(top_sim_movies_ratings)
            #print(row)
            # Avg_user rating
            try:
                row.append(sample_train_averages['user'][user])
            except KeyError:
                row.append(sample_train_averages['global'])
            except:
                raise
            #print(row)
            # Avg_movie rating
            try:
                row.append(sample_train_averages['movie'][movie])
            except KeyError:
                row.append(sample_train_averages['global'])
            except:
                raise
            #print(row)
            # finalley, The actual Rating of this user-movie pair...
            row.append(rating)
            #print(row)
            count = count + 1

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

```
preparing 7333 tuples for the dataset..

Done for 1000 rows----- 0:03:49.224918
Done for 2000 rows----- 0:07:39.443069
Done for 3000 rows----- 0:11:29.176915
Done for 4000 rows----- 0:15:19.558227
Done for 5000 rows----- 0:19:09.032166
Done for 6000 rows----- 0:22:58.872801
Done for 7000 rows----- 0:26:48.450619
 0:28:04.704096
```

__Reading from the file to make a test dataframe __

In [169]:

```
reg_test_df = pd.read_csv('reg_test.csv', names = ['user', 'movie', 'GAvg', 'sur1', 'sur
                                                   'smr1', 'smr2', 'smr3', 'smr4
                                                   'UAvg', 'MAvg', 'rating'], hea

reg_test_df.head(4)
```

Out[169]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	U
0	808635	71	3.607185	3.0	4.0	4.0	3.0	3.0	3.0	4.0	3.0	3.75	3.75	
1	941866	71	3.607185	3.0	4.0	5.0	3.0	3.0	5.0	4.5	4.5	4.50	4.50	
2	1737912	71	3.607185	3.0	5.0	4.0	3.0	3.0	3.0	3.0	3.0	3.00	3.00	
3	1849204	71	3.607185	3.0	5.0	4.0	3.0	3.0	4.0	4.0	4.0	4.00	4.00	

- **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 [113]:

```
from surprise import Reader, Dataset
```

4.3.2.1 Transforming train data

- We can't give raw data (movie, user, rating) to train the model in Surprise library.
- They have a saperate format for TRAIN and TEST data, which will be useful for training the models like SVD, KNNBaseLineOnly....etc..,in Surprise.
- We can form the trainset from a file, or from a Pandas DataFrame. http://surprise.readthedocs.io/en/stable/getting_started.html#load-dom-dataframe-py (http://surprise.readthedocs.io/en/stable/getting_started.html#load-dom-dataframe-py)

In [114]:

```
# 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 [192]:

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

Out[192]:

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

4.4 Applying Machine Learning models

- Global dictionary that stores rmse and mape for all the models....
  - It stores the metrics in a dictionary of dictionaries

> **keys** : model names(string)

> **value**: dict(**key** : metric, **value** : value)

In [116]:

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

models_evaluation_train, models_evaluation_test
```

Out[116]:

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

> **Utility functions for running regression models**

In [149]:

```
def tune(x_train,y_train,model):
    '''Performing hyperparameter tuning and predicting using best estimators'''
    parameters = {'max_depth':[2,3,5,6,8],'n_estimators':[50,100,200,300,500,1000]}
    clf = GridSearchCV(model, parameters,scoring='neg_root_mean_squared_error',cv=2,n_jo
    clf.fit(x_train,y_train)

    print(abs(clf.best_score_)) # returning absolute value of neg_rmse best score
    print(clf.best_estimator_)  #printing the best estimators for the model
```

In [118]:

```
# 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 [119]:

```
# 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 ''pre
    '''
    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 LIGHTGBM with initial 13 features

In [123]:

```
import xgboost as xgb
import lightgbm as lgbm
```

In [170]:

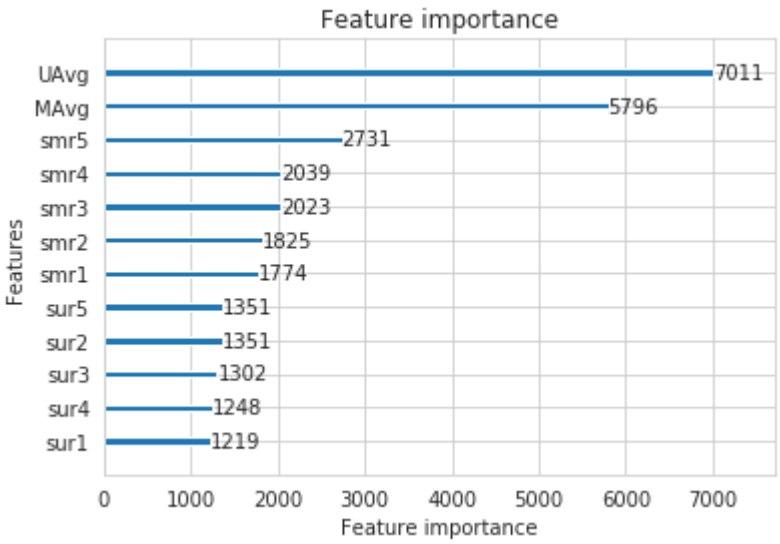
```
%%time
import sklearn
from sklearn.model_selection import GridSearchCV
# prepare Train data
x_train = reg_train.drop(['user','movie','rating'], axis=1)
y_train = reg_train['rating']

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

y_train.shape
# initialize Our first XGBoost model...
first_xgb = lgbm.LGBMRegressor(silent=True,learning_rate=0.1,random_state=15)
tune(x_train,y_train,first_xgb)
```

```
0.7889161775793031
LGBMRegressor(boosting_type='gbdt', class_weight=None, colsample_bytree=1.
0,
              importance_type='split', learning_rate=0.1, max_depth=8,
              min_child_samples=20, min_child_weight=0.001, min_split_gain
=0.0,
              n_estimators=200, n_jobs=-1, num_leaves=31, objective=None,
              random_state=15, reg_alpha=0.0, reg_lambda=0.0, silent=True,
              subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
CPU times: user 3.83 s, sys: 3.72 s, total: 7.55 s
Wall time: 15.1 s
```

In [180]:

```
%%time
tuned_algo = lgbm.LGBMRegressor(boosting_type='gbdt', class_weight=None, colsample_bytre
              importance_type='split', learning_rate=0.1, max_depth=6,
              min_child_samples=20, min_child_weight=0.001, min_split_gain=0.0,
              n_estimators=1000, n_jobs=10, num_leaves=31, objective=None,
              random_state=15, reg_alpha=0.0, reg_lambda=0.0, silent=True,
              subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
train_results, test_results = run_xgboost(tuned_algo, 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

lgbm.plot_importance(tuned_algo._Booster)
plt.show()
```

```
Training the model..
Done. Time taken : 0:00:01.728635

Done

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

TEST DATA
------------------------------
RMSE :  0.5706884793878281
MAPE :  14.803759938452682
```




```
CPU times: user 22.9 s, sys: 72 ms, total: 22.9 s
Wall time: 2.67 s
```

4.4.2 Suprise BaselineModel

In [129]:

```
from surprise import BaselineOnly
```

Predicted rating : (baseline prediction)

__ __

- http://surprise.readthedocs.io/en/stable/basic_algorithms.html#surprise.prediction_algorithms.baseline_only.BaselineOnly

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

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

__Optimization function (Least Squares Problem) __

- http://surprise.readthedocs.io/en/stable/prediction_algorithms.html#baselines-estimates-configuration

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

In [193]:

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


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

```
Training the model...
Estimating biases using sgd...
Done. time taken : 0:00:00.255997

Evaluating the model with train data..
time taken : 0:00:00.213714
---------------
Train Data
---------------
RMSE : 0.9849554257056824

MAPE : 31.59424416203202

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:00.041136
---------------
Test Data
---------------
RMSE : 0.9724521102442415

MAPE : 30.787204629897452

storing the test results in test dictionary...

------------------------------------------
Total time taken to run this algorithm : 0:00:00.512986
```

4.4.3 LIGHTGBM with initial 13 features + Surprise Baseline predictor

Updating Train Data

In [194]:

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

Out[194]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	
0	808635	71	3.607185	3.0	4.0	4.0	3.0	3.0	3.0	4.0	3.0	3.75	3.75	3.7
1	898730	71	3.607185	3.0	4.0	4.0	3.0	5.0	1.0	4.0	4.0	3.00	4.00	3.1

Updating Test Data

In [196]:

```
# 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[196]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UA
0	808635	71	3.607185	3.0	4.0	4.0	3.0	3.0	3.0	4.0	3.0	3.75	3.75	3
1	941866	71	3.607185	3.0	4.0	5.0	3.0	3.0	5.0	4.5	4.5	4.50	4.50	4

In [197]:

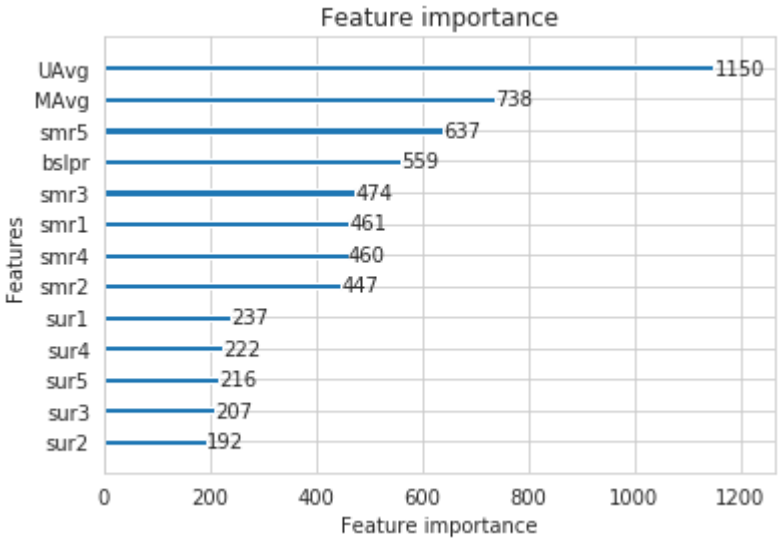
```
%%time
# prepare train data
x_train = reg_train.drop(['user', 'movie','rating'], axis=1)
y_train = reg_train['rating']

# Prepare Test data
x_test = reg_test_df.drop(['user','movie','rating'], axis=1)
y_test = reg_test_df['rating']
#hyperparameter tuning
tune(x_train,y_train,first_xgb)
```

```
0.7942986454701231
LGBMRegressor(boosting_type='gbdt', class_weight=None, colsample_bytree=1.
0,
              importance_type='split', learning_rate=0.1, max_depth=8,
              min_child_samples=20, min_child_weight=0.001, min_split_gain
=0.0,
              n_estimators=200, n_jobs=-1, num_leaves=31, objective=None,
              random_state=15, reg_alpha=0.0, reg_lambda=0.0, silent=True,
              subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
CPU times: user 4.14 s, sys: 3.34 s, total: 7.48 s
Wall time: 15.3 s
```

In [200]:

```
%%time
#predicting the model with tuned parameters
algo_new = lgbm.LGBMRegressor(boosting_type='gbdt', class_weight=None, colsample_bytree=
              importance_type='split', learning_rate=0.1, max_depth=8,
              min_child_samples=20, min_child_weight=0.001, min_split_gain=0.0,
              n_estimators=200, n_jobs=-1, num_leaves=31, objective=None,
              random_state=15, reg_alpha=0.0, reg_lambda=0.0, silent=True,
              subsample=1.0, subsample_for_bin=200000, subsample_freq=0)

train_results, test_results = run_xgboost(algo_new, x_train, y_train, x_test, y_test)

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

lgbm.plot_importance(algo_new._Booster)
plt.show()
```

```
Training the model..
Done. Time taken : 0:00:00.396355

Done

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

TEST DATA
------------------------------
RMSE :  0.7075379497247428
MAPE :  19.564318061457247
```




```
CPU times: user 5.28 s, sys: 24 ms, total: 5.31 s
Wall time: 905 ms
```

4.4.4 Surprise KNNBaseline predictor

In [131]:

```
from surprise import KNNBaseline
```

- KNN BASELINE
  - http://surprise.readthedocs.io/en/stable/knn_inspired.html#surprise.prediction_algorithms.knns.KNNBas (http://surprise.readthedocs.io/en/stable/knn_inspired.html#surprise.prediction_algorithms.knns.KNNBa

- PEARSON_BASELINE SIMILARITY
  - http://surprise.readthedocs.io/en/stable/similarities.html#surprise.similarities.pearson_baseline (http://surprise.readthedocs.io/en/stable/similarities.html#surprise.similarities.pearson_baseline)

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

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

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

- $\boldsymbol{b_{ui}}$ - *Baseline prediction* of (user,movie) rating
- $\boldsymbol{N_i^k(u)}$ - 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):

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

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

4.4.4.1 Surprise KNNBaseline with user user similarities

In [201]:

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

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

```
Training the model...
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Done. time taken : 0:00:04.953336

Evaluating the model with train data..
time taken : 0:00:10.573284
---------------
Train Data
---------------
RMSE : 0.1850277402650433

MAPE : 4.411377364338394

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:01.659460
---------------
Test Data
---------------
RMSE : 0.16599875287025806

MAPE : 3.877408095400662

storing the test results in test dictionary...

------------------------------------------------
Total time taken to run this algorithm : 0:00:17.187078
```

4.4.4.2 Surprise KNNBaseline with movie movie similarities

In [202]:

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

# '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 va
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, test

# 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 : 0:00:00.355006

Evaluating the model with train data..
time taken : 0:00:01.130263
---------------
Train Data
---------------
RMSE : 0.16596723179872885

MAPE : 3.581838014223545

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:00.283125
---------------
Test Data
---------------
RMSE : 0.15643145224029043

MAPE : 3.335072714648621

storing the test results in test dictionary...

------------------------------------------
Total time taken to run this algorithm : 0:00:01.769103
```

4.4.5 LIGHTGBM 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 [203]:

```
# 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[203]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	
0	808635	71	3.607185	3.0	4.0	4.0	3.0	3.0	3.0	4.0	3.0	3.75	3.75	3.7
1	898730	71	3.607185	3.0	4.0	4.0	3.0	5.0	1.0	4.0	4.0	3.00	4.00	3.1

__Preparing Test data __

In [204]:

```
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[204]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UA
0	808635	71	3.607185	3.0	4.0	4.0	3.0	3.0	3.0	4.0	3.0	3.75	3.75	3
1	941866	71	3.607185	3.0	4.0	5.0	3.0	3.0	5.0	4.5	4.5	4.50	4.50	4

In [206]:

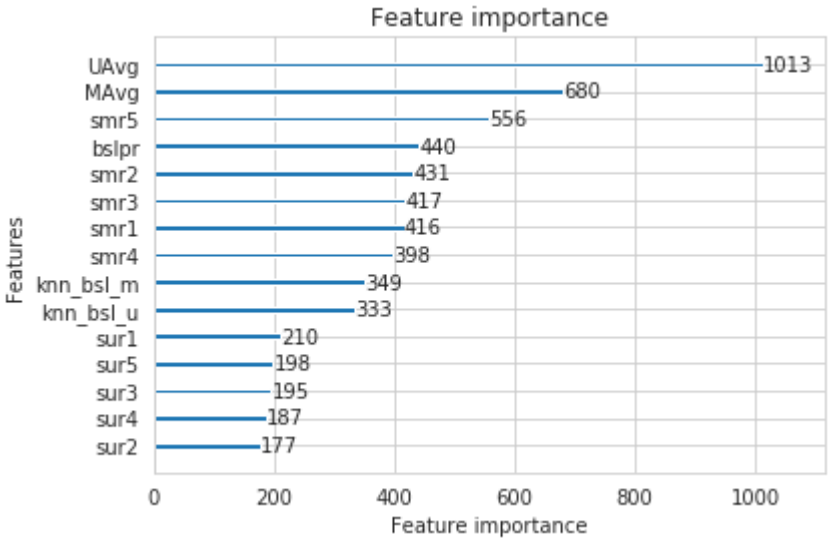
```
%%time
# prepare the train data....
x_train = reg_train.drop(['user', 'movie', 'rating'], axis=1)
y_train = reg_train['rating']

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

# declare the model
tune(x_train,y_train,first_xgb)
```

```
0.7996521861224373
LGBMRegressor(boosting_type='gbdt', class_weight=None, colsample_bytree=1.
0,
              importance_type='split', learning_rate=0.1, max_depth=8,
              min_child_samples=20, min_child_weight=0.001, min_split_gain
=0.0,
              n_estimators=200, n_jobs=-1, num_leaves=31, objective=None,
              random_state=15, reg_alpha=0.0, reg_lambda=0.0, silent=True,
              subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
CPU times: user 4.46 s, sys: 3.4 s, total: 7.86 s
Wall time: 16.7 s
```

In [208]:

```
%%time
algo_new1 = lgbm.LGBMRegressor(boosting_type='gbdt', class_weight=None, colsample_bytree
              importance_type='split', learning_rate=0.1, max_depth=8,
              min_child_samples=20, min_child_weight=0.001, min_split_gain=0.0,
              n_estimators=200, n_jobs=-1, num_leaves=31, objective=None,
              random_state=15, reg_alpha=0.0, reg_lambda=0.0, silent=True,
              subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
train_results, test_results = run_xgboost(algo_new1, 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


lgbm.plot_importance(algo_new1._Booster)
plt.show()
```

```
Training the model..
Done. Time taken : 0:00:00.442997

Done

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

TEST DATA
------------------------------
RMSE :  0.7166536850788104
MAPE :  19.83676156128557
```




```
CPU times: user 5.75 s, sys: 32 ms, total: 5.78 s
Wall time: 953 ms
```

4.4.6 Matrix Factorization Techniques

4.4.6.1 SVD Matrix Factorization User Movie intractions

In [209]:

```
from surprise import SVD
```

http://surprise.readthedocs.io/en/stable/matrix_factorization.html#surprise.prediction_algorithms.matrix_factoriza (http://surprise.readthedocs.io/en/stable/matrix_factorization.html#surprise.prediction_algorithms.matrix_factoriz

- __ Predicted Rating : __
  - $\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$
    - $\boldsymbol{q}_i$ - Representation of item(movie) in latent factor space
    - $\boldsymbol{p}_u$ - Representation of user in new latent factor space

- A BASIC MATRIX FACTORIZATION MODEL in https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf (https://datajobs.com/data-science-repo/Recommender-Systems-%5BNetflix%5D.pdf)

- **Optimization problem with user item interactions and regularization (to avoid overfitting)**
  - $\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda \left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2 \right)$

In [210]:

```
# 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 : 0:00:02.299400

Evaluating the model with train data..
time taken : 0:00:00.303641
---------------
Train Data
---------------
RMSE : 0.6695653223056354

MAPE : 20.622307740969447

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:00.073659
---------------
Test Data
---------------
RMSE : 0.6571881466926525

MAPE : 20.02575606723048

storing the test results in test dictionary...

------------------------------------------
Total time taken to run this algorithm : 0:00:02.677490
```

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

In [211]:

```
from surprise import SVDpp
```

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

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

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

- **Optimization problem with user item interactions and regularization (to avoid overfitting)**
  - $$\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda \left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2 + ||y_j||^2 \right)$$

In [212]:

```
# 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

# 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 : 0:00:18.795949

Evaluating the model with train data..
time taken : 0:00:01.206654
---------------
Train Data
---------------
RMSE : 0.5812160491261983

MAPE : 17.11058770582778

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:00.327483
---------------
Test Data
---------------
RMSE : 0.5818513515656963

MAPE : 17.079536802945626

storing the test results in test dictionary...

------------------------------------------------
Total time taken to run this algorithm : 0:00:20.331011
```

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

Preparing Train data

In [213]:

```
# 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[213]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	...	smr4	smr5	UA
0	808635	71	3.607185	3.0	4.0	4.0	3.0	3.0	3.0	4.0	...	3.75	3.75	3.750(
1	898730	71	3.607185	3.0	4.0	4.0	3.0	5.0	1.0	4.0	...	3.00	4.00	3.133

2 rows × 21 columns

__Preparing Test data __

In [214]:

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

reg_test_df.head(2)
```

Out[214]:

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	...	smr4	smr5	UAvg
0	808635	71	3.607185	3.0	4.0	4.0	3.0	3.0	3.0	4.0	...	3.75	3.75	3.75
1	941866	71	3.607185	3.0	4.0	5.0	3.0	3.0	5.0	4.5	...	4.50	4.50	4.50

2 rows × 21 columns

In [215]:

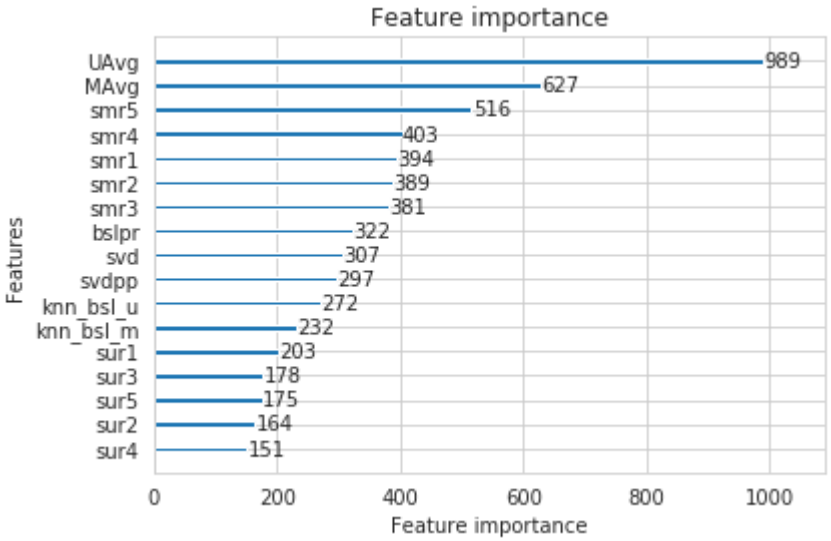
```
%%time
# prepare x_train and y_train
x_train = reg_train.drop(['user', 'movie', 'rating',], axis=1)
y_train = reg_train['rating']

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

#hyper Parameter tuning
tune(x_train,y_train,first_xgb)
```

```
0.7996921792139257
LGBMRegressor(boosting_type='gbdt', class_weight=None, colsample_bytree=1.
0,
              importance_type='split', learning_rate=0.1, max_depth=8,
              min_child_samples=20, min_child_weight=0.001, min_split_gain
=0.0,
              n_estimators=200, n_jobs=-1, num_leaves=31, objective=None,
              random_state=15, reg_alpha=0.0, reg_lambda=0.0, silent=True,
              subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
CPU times: user 4.48 s, sys: 36 ms, total: 4.52 s
Wall time: 13.7 s
```

In [216]:

```
%%time
#predicting the data using tuned parameters
algo_final = lgbm.LGBMRegressor(boosting_type='gbdt', class_weight=None, colsample_bytre
              importance_type='split', learning_rate=0.1, max_depth=8,
              min_child_samples=20, min_child_weight=0.001, min_split_gain=0.0,
              n_estimators=200, n_jobs=-1, num_leaves=31, objective=None,
              random_state=15, reg_alpha=0.0, reg_lambda=0.0, silent=True,
              subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
train_results, test_results = run_xgboost(algo_final, x_train, y_train, x_test, y_test)

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


lgbm.plot_importance(algo_final._Booster)
plt.show()
```

```
Training the model..
Done. Time taken : 0:00:00.470610

Done

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

TEST DATA
------------------------------
RMSE :  0.7211012017316449
MAPE :  19.970730962549222
```




```
CPU times: user 6.06 s, sys: 40 ms, total: 6.1 s
Wall time: 1.01 s
```

4.4.8 XgBoost with Surprise Baseline + Surprise KNNbaseline + MF Techniques

In [217]:

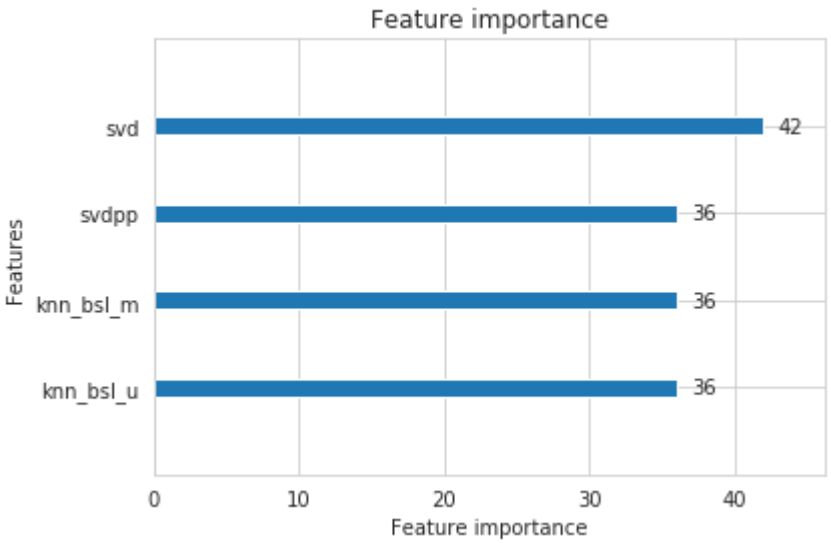
```
%%time
# prepare train data
x_train = reg_train[['knn_bsl_u', 'knn_bsl_m', 'svd', 'svdpp']]
y_train = reg_train['rating']

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

#tuning best model using hyperparameters
tune(x_train,y_train,first_xgb)
```

```
1.1033412791481692
LGBMRegressor(boosting_type='gbdt', class_weight=None, colsample_bytree=1.
0,
              importance_type='split', learning_rate=0.1, max_depth=2,
              min_child_samples=20, min_child_weight=0.001, min_split_gain
=0.0,
              n_estimators=50, n_jobs=-1, num_leaves=31, objective=None,
              random_state=15, reg_alpha=0.0, reg_lambda=0.0, silent=True,
              subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
CPU times: user 592 ms, sys: 12 ms, total: 604 ms
Wall time: 9.09 s
```

In [219]:

```
%%time
#predicting using tuned parameters
algo_final1 = lgbm.LGBMRegressor(boosting_type='gbdt', class_weight=None, colsample_bytr
              importance_type='split', learning_rate=0.1, max_depth=2,
              min_child_samples=20, min_child_weight=0.001, min_split_gain=0.0,
              n_estimators=50, n_jobs=-1, num_leaves=31, objective=None,
              random_state=15, reg_alpha=0.0, reg_lambda=0.0, silent=True,
              subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
train_results, test_results = run_xgboost(algo_final1, x_train, y_train, x_test, y_test)

# store the results in models_evaluations dictionaries
models_evaluation_train['algo_final1'] = train_results
models_evaluation_test['algo_final1'] = test_results

lgbm.plot_importance(algo_final1._Booster)
plt.show()
```

```
Training the model..
Done. Time taken : 0:00:00.042547

Done

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

TEST DATA
------------------------------
RMSE :  1.0744947763752575
MAPE :  35.29992553630213
```




```
CPU times: user 1.16 s, sys: 0 ns, total: 1.16 s
Wall time: 412 ms
```

4.5 Comparision between all models

In [221]:

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

Out[221]:

```
knn_bsl_m      0.15643145224029043
knn_bsl_u      0.16599875287025806
first_algo      0.5706884793878281
svdpp           0.5818513515656963
svd             0.6571881466926525
algo_new        0.7075379497247428
xgb_knn_bsl     0.7166536850788104
algo_final      0.7211012017316449
bsl_algo        0.9724521102442415
algo_final1     1.0744947763752575
Name: rmse, dtype: object
```

In [224]:

```
print("-"*100)
print("Total time taken to run this entire notebook ( with saved files) is :",datetime.r
```

```
----------------------------------------------------------------------------
--------------------------
Total time taken to run this entire notebook ( with saved files) is : 0:0
0:24.367784
```

```
NOTE :
1. The globalstart was not uncommented thus the total time taken might not be correct
2. The reason for consideration of lightgbm instead of xgboost was that lightgbm
outperforms xgboost
3. The quality of the metric for recommendation  can be noted in the below
summarization section
```

5. Assignment

```
1.Instead of using 10K users and 1K movies to train the above models, use 25K users
and 3K movies (or more) to train all of the above models. Report the RMSE and MAPE on
the test data using larger amount of data and provide a comparison between various
models as shown above.

NOTE: Please be patient as some of the code snippets make take many hours to compelte
execution.

2.Tune hyperparamters of all the Xgboost models above to improve the RMSE.
```

6.Conclusion:

In [225]:

```
from prettytable import PrettyTable
#summarizing metrics using pretytable
pt = PrettyTable()
pt.field_names = ["Model","RMSE","MAPE"]
pt.add_row(["LIGHTGBM with initial 13 features",0.57,14.80])
pt.add_row(["LIGHTGBM with initial 13 features + Surprise Baseline predictor",0.70,19.5
pt.add_row(["LIGHTGBM with initial 13 features + Surprise Baseline predictor + KNNBasel
pt.add_row(["LIGHTGBM with 13 features + Surprise Baseline + Surprise KNNbaseline + MF
pt.add_row(["LIGHTGBM with Surprise Baseline + Surprise KNNbaseline + MF Techniques",1.0
print(pt)
```

```
+-------------------------------------------------------------------------
----------------+------+-------+
|                                                Model
| RMSE |  MAPE |
+-------------------------------------------------------------------------
----------------+------+-------+
|                              LIGHTGBM with initial 13 features
| 0.57 |  14.8 |
|              LIGHTGBM with initial 13 features + Surprise Baseline predic
tor              | 0.7  | 19.56 |
| LIGHTGBM with initial 13 features + Surprise Baseline predictor + KNNBas
eline predictor | 0.72 | 19.84 |
|   LIGHTGBM with 13 features + Surprise Baseline + Surprise KNNbaseline +
MF Techniques   | 0.72 | 19.97 |
|          LIGHTGBM with Surprise Baseline + Surprise KNNbaseline + MF Tec
hniques          | 1.07 | 35.29 |
+-------------------------------------------------------------------------
----------------+------+-------+
```

In [226]:

```
%%javascript
// Converts integer to roman numeral
// https://github.com/kmahelona/ipython_notebook_goodies
// https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebook_toc.js
function romanize(num) {
    var lookup = {M:1000,CM:900,D:500,CD:400,C:100,XC:90,L:50,XL:40,X:10,IX:9,V:5,IV:4,I
    roman = '',
        i;
    for ( i in lookup ) {
        while ( num >= lookup[i] ) {
        roman += i;
        num -= lookup[i];
        }
    }
    return roman;
 }

// Builds a <ul> Table of Contents from all <headers> in DOM
function createTOC(){
    var toc = "";
    var level = 0;
    var levels = {}
    $('#toc').html('');

    $(":header").each(function(i){
        if (this.id=='tocheading'){return;}

        var titleText = this.innerHTML;
        var openLevel = this.tagName[1];

        if (levels[openLevel]){
        levels[openLevel] += 1;
        } else{
        levels[openLevel] = 1;
        }

        if (openLevel > level) {
        toc += (new Array(openLevel - level + 1)).join('<ul class="toc">');
        } else if (openLevel < level) {
        toc += (new Array(level - openLevel + 1)).join("</ul>");
        for (i=level;i>openLevel;i--){levels[i]=0;}
        }

        level = parseInt(openLevel);


        if (this.id==''){this.id = this.innerHTML.replace(/ /g,"-")}
        var anchor = this.id;

        toc += '<li><a style="text-decoration:none", href="#' + encodeURIComponent(ancho

    });


    if (level) {
    toc += (new Array(level + 1)).join("</ul>");
    }
```

```
    $('#toc').append(toc);

};

// Executes the createToc function
setTimeout(function(){createTOC();},100);

// Rebuild to TOC every minute
setInterval(function(){createTOC();},60000);
```