



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-1-5583846f429> (very nice blog)
- surprise library: <http://surpriselib.com/> (we use many models from this library)
- surprise library doc: http://surprise.readthedocs.io/en/stable/getting_started.html (we use many models from this library)
- installing surprise: <https://github.com/NicolasHug/Surprise#installation>
- Research paper: <http://courses.ischool.berkeley.edu/i290-dm/s11/SECURE/a1-koren.pdf> (most of our work was inspired by this paper)
- SVD Decomposition : <https://www.youtube.com/watch?v=P5mlg91as1c>

1.4 Real world/Business Objectives and constraints

Objectives:

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

Constraints:

1. Some form of interpretability.

2. Machine Learning Problem

2.1 Data

2.1.1 Data Overview

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

Data files :

- combined_data_1.txt
- combined_data_2.txt
- combined_data_3.txt
- combined_data_4.txt
- movie_titles.csv

The first line of each file [combined_data_1.txt, combined_data_2.txt, combined_data_3.txt, combined_data_4.txt] contains the movie id followed by a colon. Each 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.

```
# 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

```

Double-click (or enter) to edit

3. Exploratory Data Analysis

3.1 Preprocessing

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

```

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 appending each rating to a global file 'tr
    data = open('data.csv', mode='w')

    row = list()
    files=['data_folder/combined_data_1.txt','data_folder/combined_data_2.txt',
           'data_folder/combined_data_3.txt', 'data_folder/combined_data_4.txt']
    for file in files:
        print("Reading ratings from {}...".format(file))
        with open(file) as f:
            for line in f:
                del row[:] # you don't have to do this.
                line = line.strip()
                if line.endswith(':'):
                    # All below are ratings for this movie, until another movie appears.
                    movie_id = line.replace(':', '')
                else:
                    row = [x for x in line.split(',')]
                    row.insert(0, movie_id)
                    data.write(','.join(row))
                    data.write('\n')
            print("Done.\n")
    data.close()
print('Time taken :', datetime.now() - start)

```



Reading ratings from data_folder/combined_data_1.txt...

Done.

Reading ratings from data_folder/combined_data_2.txt...

Done.

Reading ratings from data_folder/combined_data_3.txt...

Done.

Reading ratings from data_folder/combined_data_4.txt...

Done.

Double-click (or enter) to edit

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

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

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

Sorting the dataframe by date..

Done..

df.head()



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

df.describe()['rating']



```
count    1.004805e+08
mean     3.604290e+00
```

3.1.2 Checking for NaN values

```
50%      1 000000000000
# 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 : 0

Double-click (or enter) to edit

3.1.3 Removing Duplicates

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

 There are 0 duplicate rating entries in the data..

Double-click (or enter) to edit

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

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

 Total data

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

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

```
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)

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

 Training data

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

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

```
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

Double-click (or enter) to edit

```
# 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"
```

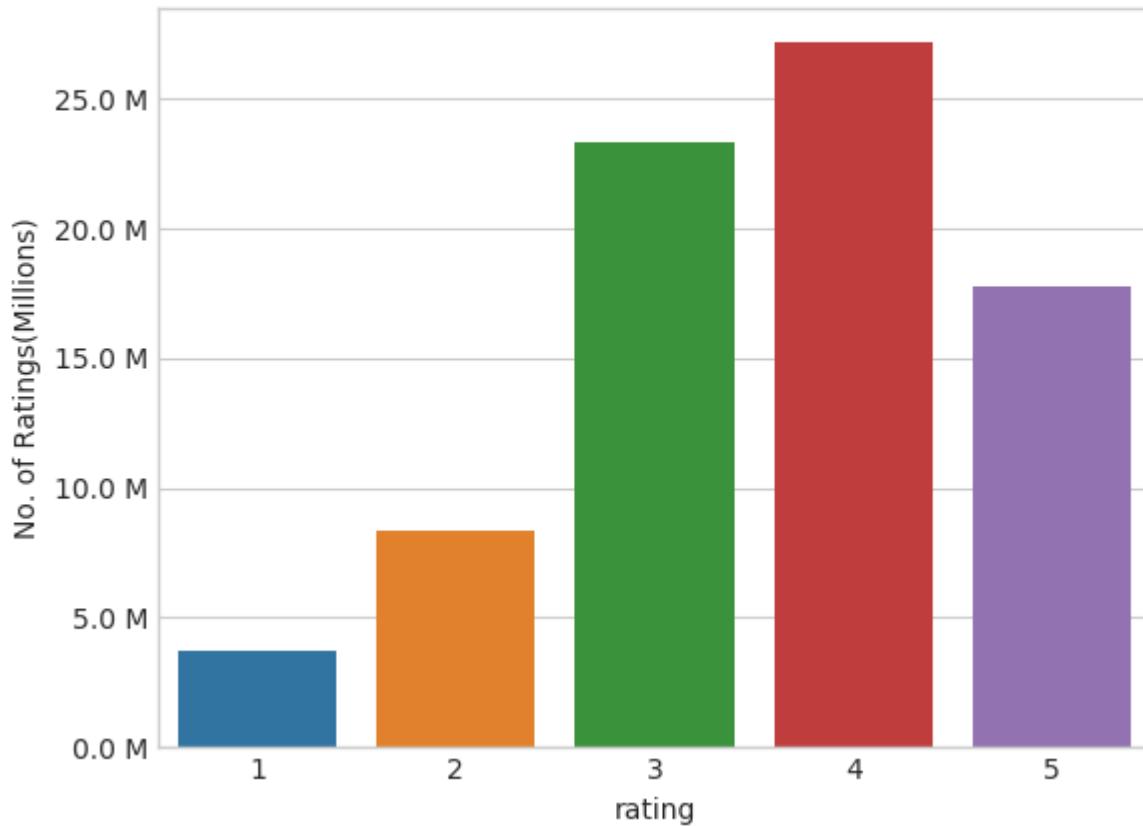
3.3.1 Distribution of ratings

```
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()
```



Distribution of ratings over Training dataset



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

```
# 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()
```



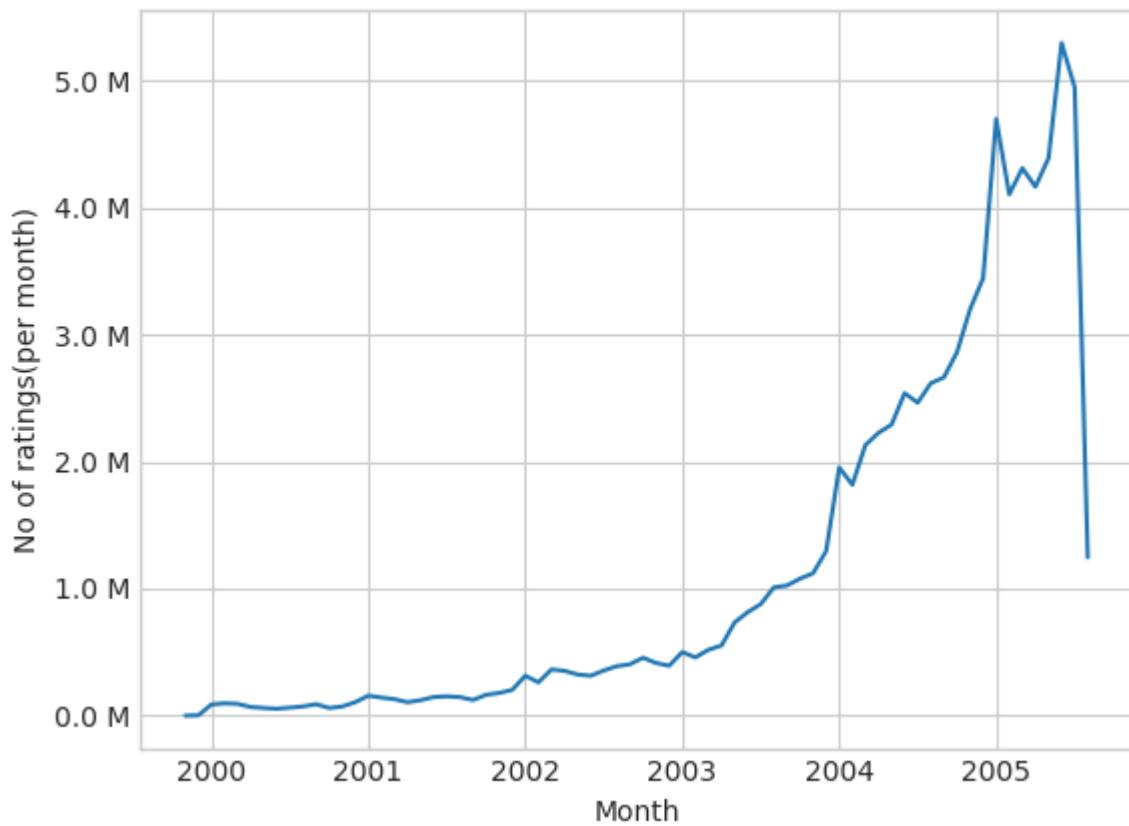
movie	user	rating	date	day_of_week
80384400	12074	2033618	4	2005-08-08

3.3.2 Number of Ratings per a month

```
80384400 12074 2033618 4 2005-08-08 Monday
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()
```



No of ratings per month (Training data)



Double-click (or enter) to edit

3.3.3 Analysis on the Ratings given by user

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



```
user
305344    17112
2439493   15896
387418    15402
1639792   9767
```

```
fig = plt.figure(figsize=plt.figaspect(.5))

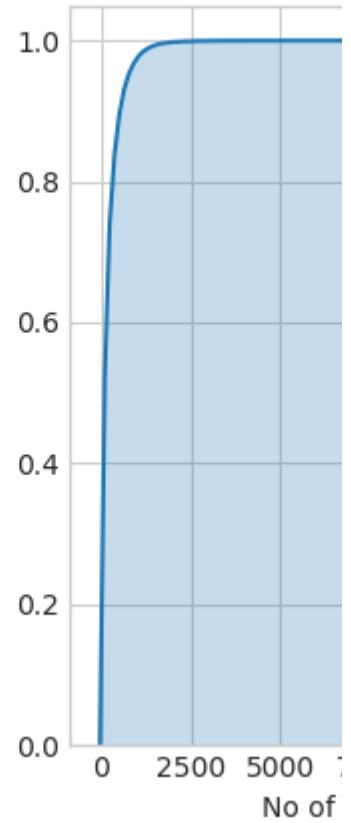
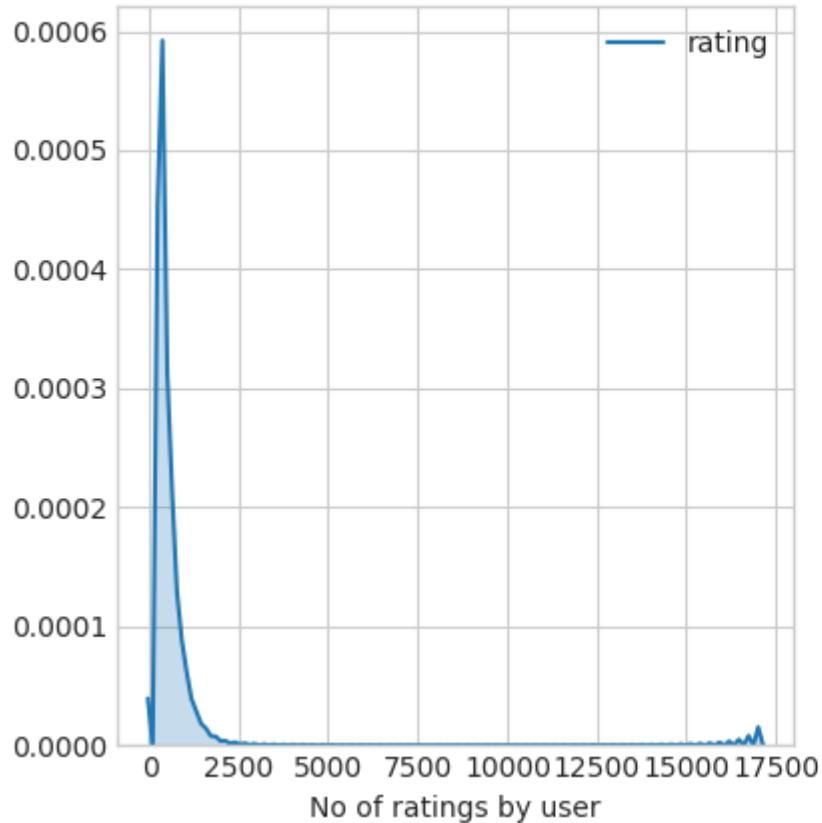
ax1 = plt.subplot(121)
sns.kdeplot(no_of_rated_movies_per_user, shade=True, ax=ax1)
plt.xlabel('No of ratings by user')
plt.title("PDF")

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()
```



PDF



```
no_of_rated_movies_per_user.describe()
```



```
count    405041.000000
mean     198.459921
std      290.793238
min      1.000000
```

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

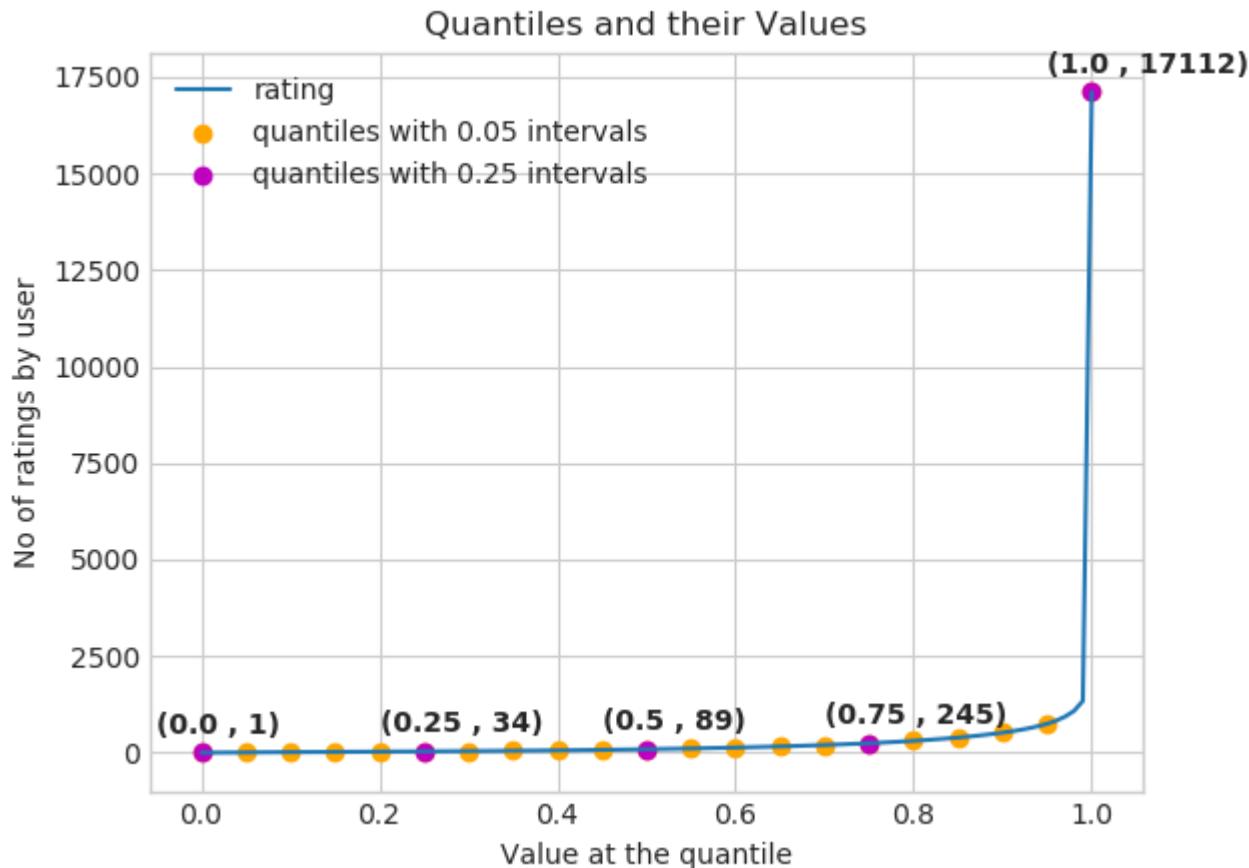
```
...  
...
```

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

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

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

plt.show()
```



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

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

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

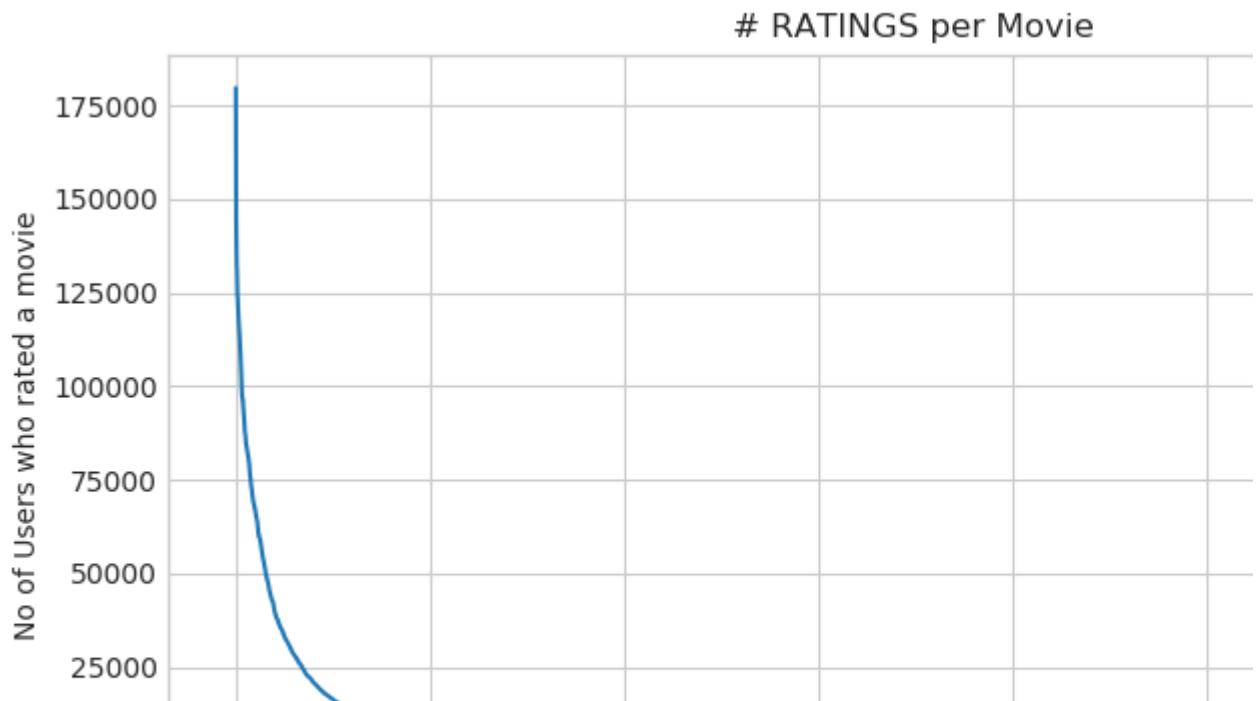
```
print('\n No of ratings at last 5 percentile : {}' .format(sum(no_of_rated_movies_per_user>=
```

👤 No of ratings at last 5 percentile : 20305

3.3.4 Analysis of ratings of a movie given by a user

```
no_of_ratings_per_movie = train_df.groupby(by='movie')['rating'].count().sort_values(ascending=False)  
  
fig = plt.figure(figsize=plt.figaspect(.5))  
ax = plt.gca()  
plt.plot(no_of_ratings_per_movie.values)  
plt.title('# RATINGS per Movie')  
plt.xlabel('Movie')  
plt.ylabel('No of Users who rated a movie')  
ax.set_xticklabels([])  
  
plt.show()
```





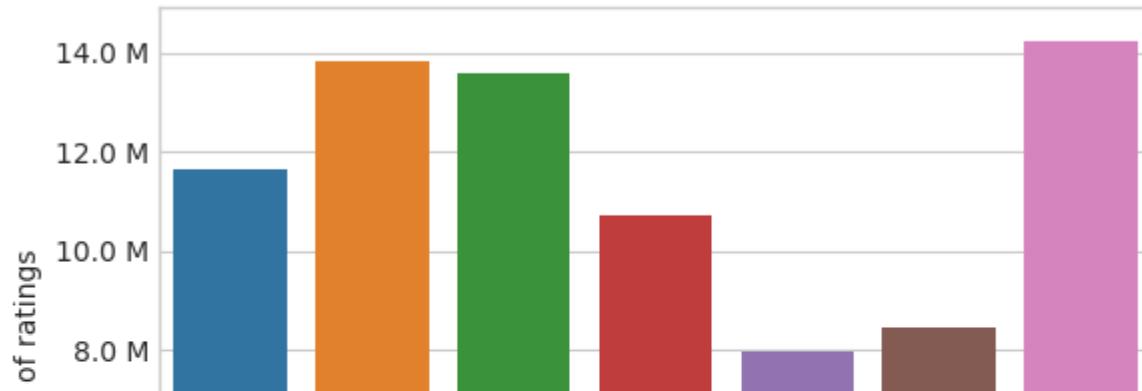
- It is very skewed.. just like number of ratings given per user.
 - There are some movies (which are very popular) which are rated by huge number of users.
 - But most of the movies (like 90%) got some hundreds of ratings.

3.3.5 Number of ratings on each day of the week

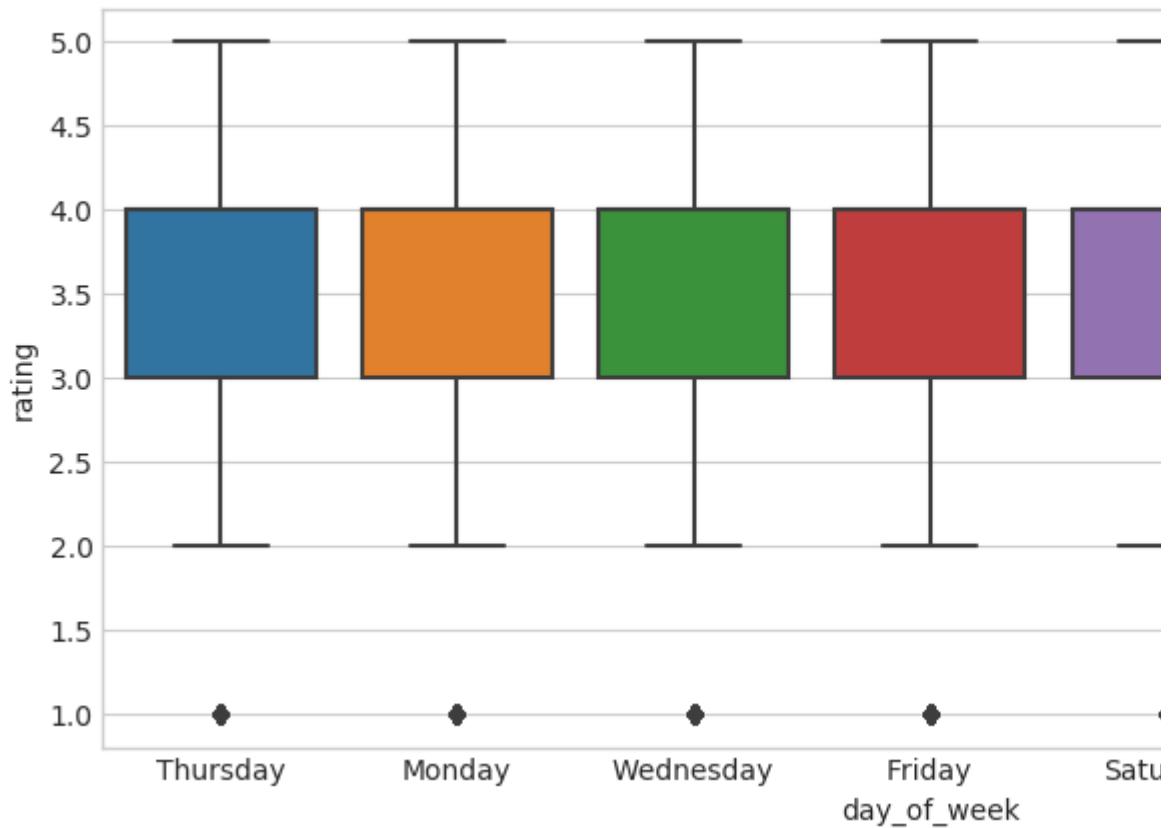
```
fig, ax = plt.subplots()
sns.countplot(x='day_of_week', data=train_df, ax=ax)
plt.title('No of ratings on each day...')
plt.ylabel('Total no of ratings')
plt.xlabel('')
ax.set_yticklabels([human(item, 'M') for item in ax.get_yticks()])
plt.show()
```



No of ratings on each day...



```
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)
```



0:01:10.003761

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



Average ratings

```
-----
day_of_week
Friday      3.585274
Monday      3.577250
Saturday    3.591791
Sunday      3.594144
Thursday    3.582463
Tuesday     3.574438
Wednesday   3.583751
Name: rating, dtype: float64
```

Double-click (or enter) to edit

3.3.6 Creating sparse matrix from data frame



3.3.6.1 Creating sparse matrix from train data frame

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

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

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



It is present in your pwd, getting it from disk....
DONE..
0:00:04.463750

The Sparsity of Train Sparse Matrix

```
us,mv = train_sparse_matrix.shape
elem = train_sparse_matrix.count_nonzero()
```

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

 Sparsity Of Train matrix : 99.8292709259195 %

3.3.6.2 Creating sparse matrix from test data frame

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

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

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

 It is present in your pwd, getting it from disk....
DONE..
0:00:01.172240

The Sparsity of Test data Matrix

```
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

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

def get_average_ratings(sparse_matrix, of_users):

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

    # ".A1" is for converting Column_Matrix to 1-D numpy array
    sum_of_ratings = sparse_matrix.sum(axis=ax).A1
    # Boolean matrix of ratings ( whether a user rated that movie or not)
    is_rated = sparse_matrix!=0
```

```
# no of ratings that each user OR movie..
no_of_ratings = is_rated.sum(axis=ax).A1

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

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

3.3.7.1 finding global average of all movie ratings

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

 {'global': 3.582890686321557}

3.3.7.2 finding average rating per user

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

 Average rating of user 10 : 3.3781094527363185

3.3.7.3 finding average rating per movie

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

 Average rating of movie 15 : 3.3038461538461537

Double-click (or enter) to edit

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

```
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')

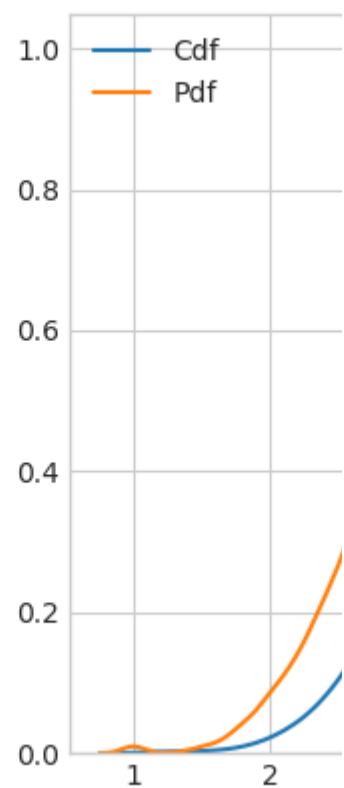
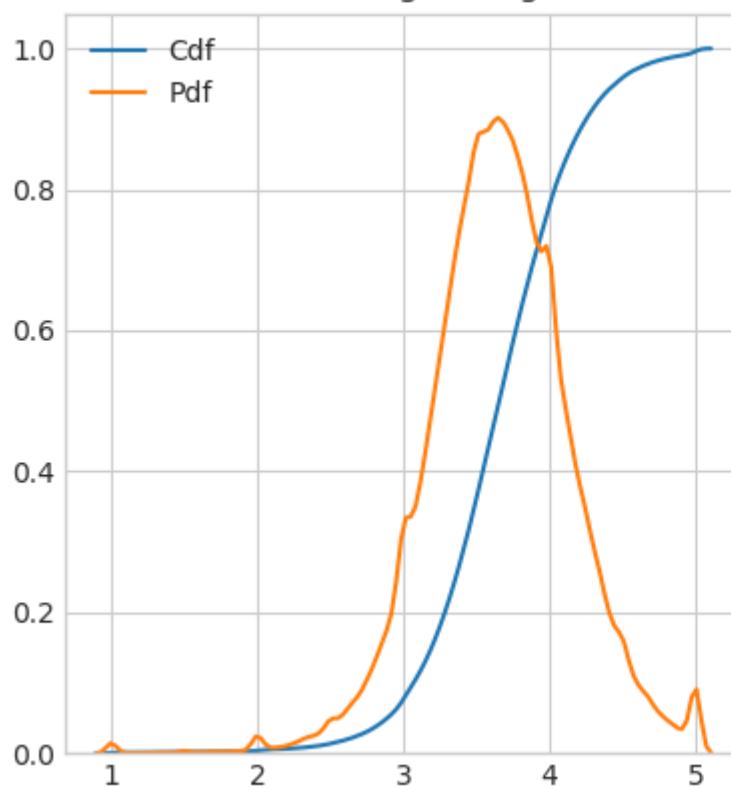
ax2.set_title('Movies-Avg-Rating')
# get the list of movie_average_ratings from the dictionary..
movie_averages = [rat for rat in train_averages['movie'].values()]
sns.distplot(movie_averages, ax=ax2, hist=False,
             kde_kws=dict(cumulative=True), label='Cdf')
sns.distplot(movie_averages, ax=ax2,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

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

```



Avg Ratings per User and per Movie



0:00:35.003443

3.3.8 Cold Start problem

3.3.8.1 Cold Start problem with Users

```

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

print('\nTotal number of Users :', total_users)
print('\nNumber of Users in Train data :', users_train)
print("\nNo of Users that didn't appear in train data: {}({} %) \n ".format(new_users,
                                                                           np.round((new_users/total_users)*100)))

```



Total number of Users : 480189

Number of Users in Train data : 405041

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

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

3.3.8.2 Cold Start problem with Movies

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

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



Total number of Movies : 17770

Number of Users in Train data : 17424

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

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

Double-click (or enter) to edit

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. users being large.

- 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)

```

from sklearn.metrics.pairwise import cosine_similarity

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

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

    start = datetime.now()
    temp = 0

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

        # get the similarity row for this user with all other users
        sim = cosine_similarity(sparse_matrix.getrow(row), sparse_matrix).ravel()
        # We will get only the top 'top' most similar users and ignore rest of them..
        top_sim_ind = sim.argsort()[-top:]
        top_sim_val = sim[top_sim_ind]

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

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

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

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

```

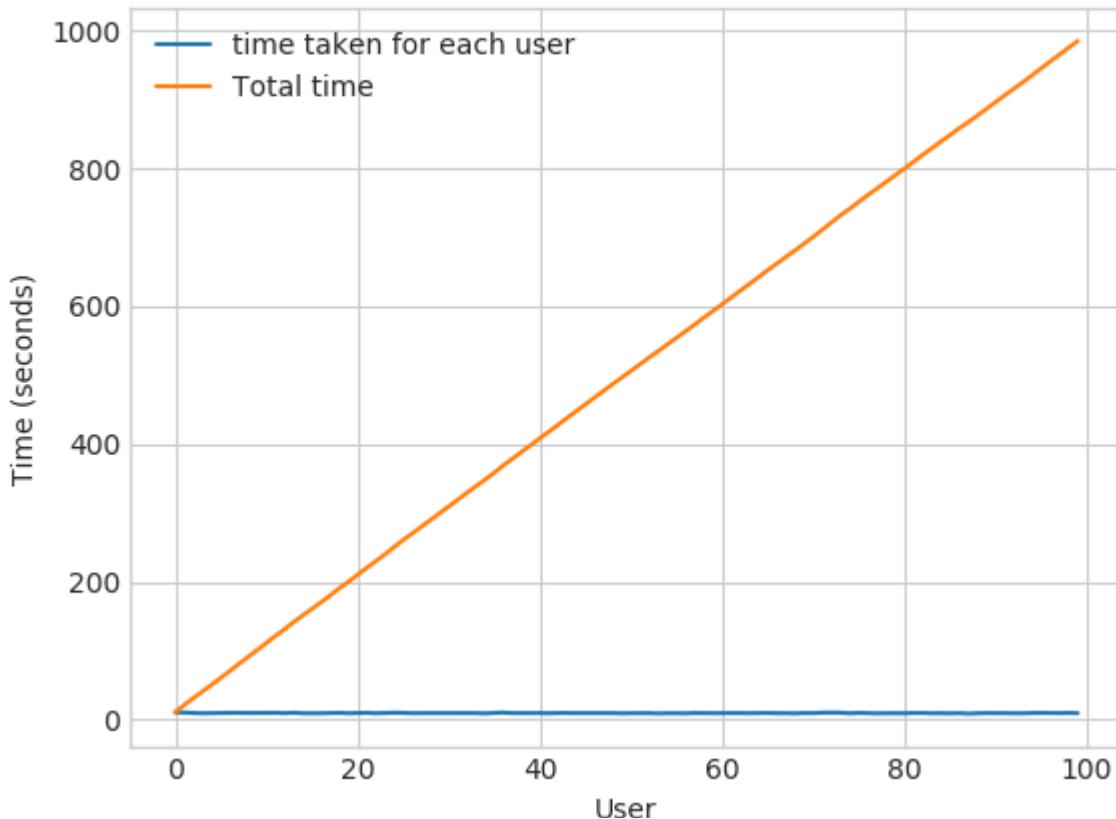
```

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

```



```
Computing top 100 similarities for each user..
computing done for 20 users [ time elapsed : 0:03:20.300488 ]
computing done for 40 users [ time elapsed : 0:06:38.518391 ]
computing done for 60 users [ time elapsed : 0:09:53.143126 ]
computing done for 80 users [ time elapsed : 0:13:10.080447 ]
computing done for 100 users [ time elapsed : 0:16:24.711032 ]
Creating Sparse matrix from the computed similarities
```



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

Double-click (or enter) to edit

- We have **405,041 users** in our training set and computing similarities between them..(**17K dimensional vector..**) is time consuming..
- From above plot, It took roughly **8.88 sec** for computing similar users for **one user**
- We have **405,041 users** with us in training set.
- $405041 \times 8.88 = 3596764.08 \text{ sec} = 59946.068 \text{ min} = 999.101133333 \text{ hours} = 41.629213889 \text{ days}$
 - Even if we run on 4 cores parallelly (a typical system now a days), It will still take almost **10 and 1/2 days**.

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

```

from datetime import datetime
from sklearn.decomposition import TruncatedSVD

start = datetime.now()

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

print(datetime.now()-start)

```

 0:29:07.069783

Here,

- $\sum \leftarrow (\text{netflix_svd.singular_values}_\cdot)$
- $V^T \leftarrow (\text{netflix_svd.components}_\cdot)$
- U is not returned. instead **Projection_of_X** onto the new vectorspace is returned.
- It uses **randomized svd** internally, which returns **All 3 of them saperately**. Use that instead..

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

```

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

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

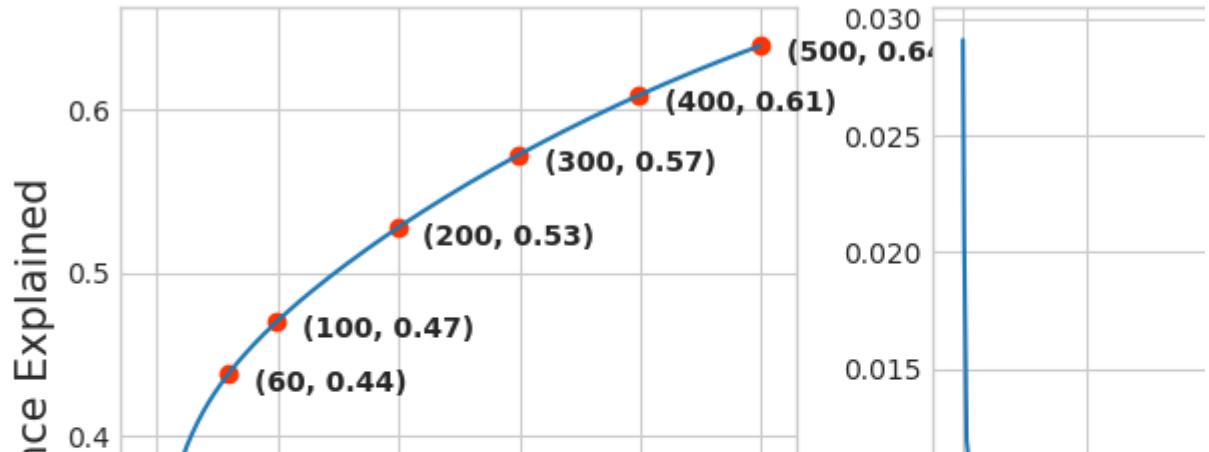
change_in_expl_var = [expl_var[i+1] - expl_var[i] for i in range(len(expl_var)-1)]
ax2.plot(change_in_expl_var)

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()

```





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

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

I think 500 dimensions is good enough

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

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

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

 0:00:45.670265

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

 (numpy.ndarray, (2649430, 500))

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

```
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')
```

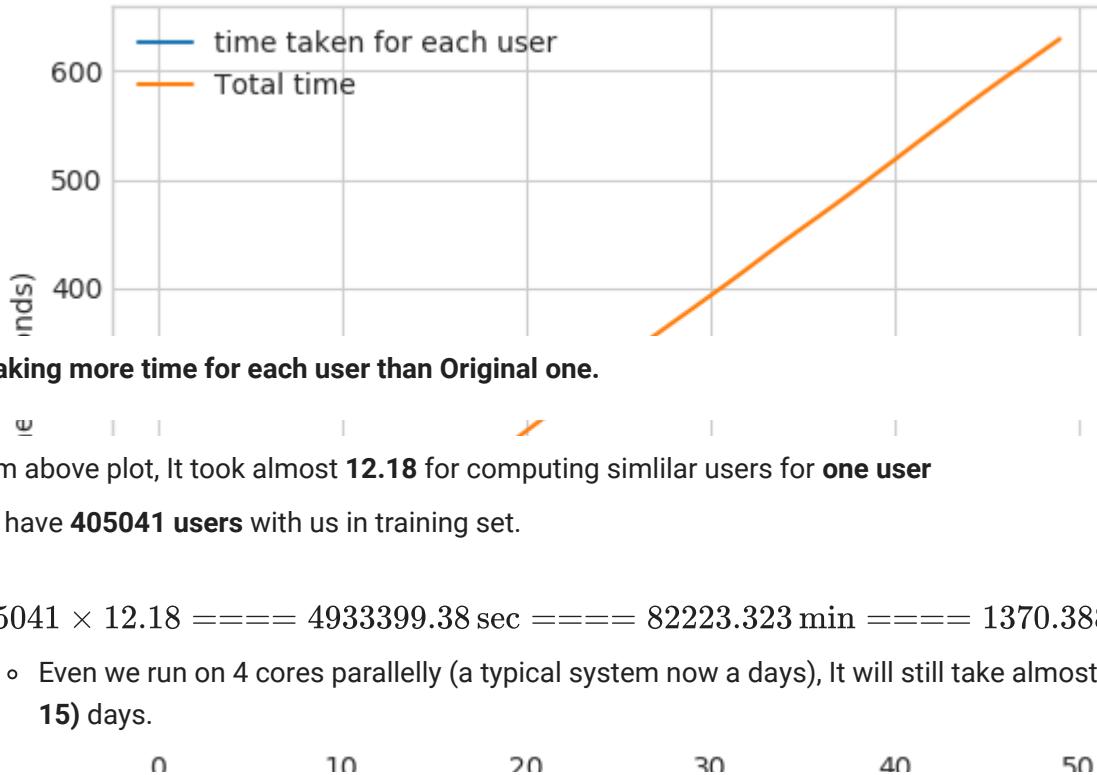
```
trunc_sparse_matrix.shape
```

 (2649430, 500)

```
start = datetime.now()
trunc_u_u_sim_matrix, _ = compute_user_similarity(trunc_sparse_matrix, compute_for_few=True,
                                                 verb_for_n_rows=10)
print("-"*50)
print("time:", datetime.now()-start)
```



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



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

- from above plot, It took almost **12.18** for computing similar users for **one user**
- We have **405041 users** with us in training set.
- $405041 \times 12.18 = 4933399.38 \text{ sec} = 82223.323 \text{ min} = 1370.388716667 \text{ hours}$
 - Even we run on 4 cores parallelly (a typical system now a days), It will still take almost (**14 - 15**) days.

• Why did this happen...??

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

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

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

-An alternative is to compute similar users for a particular user, whenever required (**ie., Run time**)

- We maintain a binary Vector for users, which tells us whether we already computed or not..
- *****If not*** :**
 - Compute top (let's just say, 1000) most similar users for this given user, and add this to our datastructure, so that we can just access it(similar users) without recomputing it again.
 -
 - *****If It is already Computed***:**
 - Just get it directly from our datastructure, which has that information.

- In production time, We might have to recompute similarities, if it is computed a long time ago. Because user preferences changes over time. If we could maintain some kind of Timer, which when expires, we have to update it (recompute it).
- ***Which datastructure to use:***
- It is purely implementation dependant.
- One simple method is to maintain a **Dictionary Of Dictionaries**.
-
- **key :** _userid_
- __value__: _Again a dictionary_
 - __key__ : _Similar User_
 - __value__: _Similarity Value_

3.4.2 Computing Movie-Movie Similarity matrix

```

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
 0:10:02.736054

m_m_sim_sparse.shape

 (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.

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

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]
```

 0:00:33.411700

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

Double-click (or enter) to edit

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....

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

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

movie_titles.head()
```



```
Tokenization took: 4.50 ms
Type conversion took: 165.72 ms
Parser memory cleanup took: 0.01 ms
```

year_of_release	title
-----------------	-------

movie_id

Similar Movies for 'Vampire Journals'

```
mv_id = 67

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

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

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



Movie -----> Vampire Journals

It has 270 Ratings from users.

We have 17284 movies which are similar to this and we will get only top most..

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

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

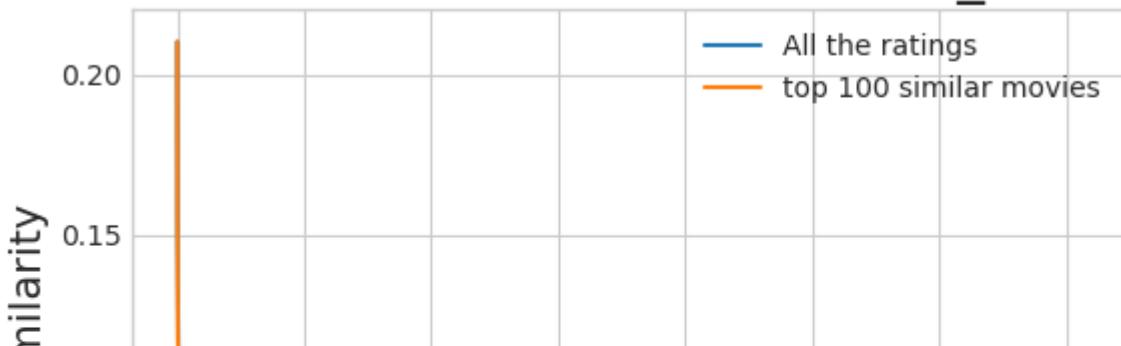
similarities[similar_indices]

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

```
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()
```



Similar Movies of 67(movie_id)



Double-click (or enter) to edit

Double-click (or enter) to edit

Top 10 similar movies

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

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

Double-click (or enter) to edit

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

Double-click (or enter) to edit

Double-click (or enter) to edit

4. Machine Learning Models



```
!pip install kaggle  
from google.colab import files  
files.upload()
```

```
↳ Requirement already satisfied: kaggle in /usr/local/lib/python3.6/dist-packages (1.5.3)  
Requirement already satisfied: urllib3<1.25,>=1.21.1 in /usr/local/lib/python3.6/dist-  
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.6/dist-packages (fr  
Requirement already satisfied: certifi in /usr/local/lib/python3.6/dist-packages (from  
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.6/dist-packag  
Requirement already satisfied: requests in /usr/local/lib/python3.6/dist-packages (fro  
Requirement already satisfied: tqdm in /usr/local/lib/python3.6/dist-packages (from ka  
Requirement already satisfied: python-slugify in /usr/local/lib/python3.6/dist-package  
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /usr/local/lib/python3.6/dist-  
Requirement already satisfied: idna<2.7,>=2.5 in /usr/local/lib/python3.6/dist-package  
Requirement already satisfied: text-unidecode==1.2 in /usr/local/lib/python3.6/dist-pa  
Choose Files kaggle.json  
• kaggle.json(application/json) - 67 bytes, last modified: 2/27/2019 - 100% done  
Saving kaggle.json to kaggle.json  
{'kaggle.json': b'{"username":"pankajkarki","key":"563115ce1ea9892ab835dfbe5b8acba1"}'
```

```
!mkdir -p ~/.kaggle  
!cp kaggle.json ~/.kaggle/  
  
# This permissions change avoids a warning on Kaggle tool startup.  
!chmod 600 ~/.kaggle/kaggle.json  
  
!kaggle datasets download -d pankajkarki/netflix  
!ls
```

```
↳ Downloading netflix.zip to /content  
  0% 0.00/2.13M [00:00<?, ?B/s]  
100% 2.13M/2.13M [00:00<00:00, 70.8MB/s]  
kaggle.json  netflix.zip  sample_data
```

```
!unzip netflix.zip  
  
↳ Archive: netflix.zip  
  inflating: sample_train_sparse_matrix.npz  
  inflating: reg_train.csv  
  inflating: reg_test.csv  
  inflating: sample_test_sparse_matrix.npz
```

```

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

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

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

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

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

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

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

    return sample_sparse_matrix

```

4.1 Sampling Data

4.1.1 Build sample train data from the train data

```

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=25000
                                                          path = path)

print(datetime.now() - start)

```

→ It is present in your pwd, getting it from disk....
 DONE..
 0:00:00.054409

4.1.2 Build sample test data from the test data

```

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=9000, no_movies=500,
                                                          path = "sample/small/sample_test_sparse_matrix.npz")
print(datetime.now() - start)

```

→ It is present in your pwd, getting it from disk....
 DONE..
 0:00:00.045440

Double-click (or enter) to edit

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

```
sample_train_averages = dict()
```

4.2.1 Finding Global Average of all movie ratings

```

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

```

→ {'global': 3.581679377504138}

4.2.2 Finding Average rating per User

```

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

```

→ Average rating of user 1515220 : 3.9655172413793105

4.2.3 Finding Average rating per Movie

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

→ Average rating of movie 15153 : 2.6458333333333335

Double-click (or enter) to edit

4.3 Featurizing data

```
print('\n No of ratings in Our Sampled train matrix is : {}'.format(sample_train_sparse_matrix.shape[0]))
print('\n No of ratings in Our Sampled test matrix is : {}'.format(sample_test_sparse_matrix.shape[0]))
```

→ No of ratings in Our Sampled train matrix is : 129286

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

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

#####
# 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..'.format(len(sample_train_ratings)))
    with open('reg_train.csv', mode='w') as reg_data_file:
        count = 0
        for (user, movie, rating) in zip(sample_train_users, sample_train_movies, sample_train_ratings):
            st = datetime.now()
            # print(user, movie)
            #----- Ratings of "movie" by similar users of "user" -----
            # compute the similar Users of the "user"
            user_sim = cosine_similarity(sample_train_sparse_matrix[user], sample_train_sparse_matrix)
            top_sim_users = user_sim.argsort()[:-1][1:] # we are ignoring 'The User' from it
            # get the ratings of most similar users for this movie
            top_ratings = sample_train_sparse_matrix[top_sim_users, movie].toarray().ravel()
            # we will make it's length "5" by adding movie averages to .
            top_sim_users_ratings = list(top_ratings[top_ratings != 0][:5])
            top_sim_users_ratings.extend([sample_train_averages['movie'][movie]]*(5 - len(top_sim_users_ratings)))
            # print(top_sim_users_ratings, end=" ")
            reg_data_file.write(str(count) + ',' + str(user) + ',' + str(movie) + ',' + str(rating) + ',' + str(top_sim_users_ratings) + '\n')
            count += 1
            if count % 10000 == 0:
                print(count)
```

```

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

```

File already exists you don't have to prepare again...
0:00:00.001911

Reading from the file to make a Train_dataframe

```

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

```

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5
0	53406	33	3.581679	4.0	5.0	5.0	4.0	1.0	5.0	2.0	5.0	3.0	1.0
1	99540	33	3.581679	5.0	5.0	5.0	4.0	5.0	3.0	4.0	4.0	3.0	5.0
2	99865	33	3.581679	5.0	5.0	4.0	5.0	3.0	5.0	4.0	4.0	5.0	4.0
3	101620	33	3.581679	2.0	3.0	5.0	5.0	4.0	4.0	3.0	3.0	4.0	5.0
4	112974	33	3.581679	5.0	5.0	5.0	5.0	5.0	3.0	5.0	5.0	5.0	3.0

- **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.

Double-click (or enter) to edit

4.3.1.2 Featurizing test data

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

sample_train_averages['global'].
```

3.581679377504138

```
start = datetime.now()

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

    print('preparing {} tuples for the dataset..\n'.format(len(sample_test_ratings)))
    with open('reg_test.csv', mode='w') as reg_data_file:
        count = 0
        for (user, movie, rating) in zip(sample_test_users, sample_test_movies, sample_test_ratings):
            st = datetime.now()
            #----- Ratings of "movie" by similar users of "user" -----
            #print(user, movie)
            try:
                # compute the similar Users of the "user"
                user_sim = cosine_similarity(sample_train_sparse_matrix[user], sample_train_sparse_matrix)
                top_sim_users = user_sim.argsort()[:-1][1:] # we are ignoring 'The User' from the list
                # get the ratings of most similar users for this movie
                top_ratings = sample_train_sparse_matrix[top_sim_users, movie].toarray().ravel()
                # we will make it's length "5" by adding movie averages to .
                top_sim_users_ratings = list(top_ratings[top_ratings != 0][:5])
                top_sim_users_ratings.extend([sample_train_averages['movie'][movie]]*(5 - len(top_sim_users_ratings)))
                # print(top_sim_users_ratings, end="--")

            except (IndexError, KeyError):
                # It is a new User or new Movie or there are no ratings for given user for top
                ##### Cold Start Problem #####
                top_sim_users_ratings.extend([sample_train_averages['global']]*(5 - len(top_sim_users_ratings)))
            except:
                print(user, movie)
```

```

# we just want KeyErrors to be resolved. Not every Exception...
raise

#----- Ratings by "user" to similar movies of "movie" -----
try:
    # compute the similar movies of the "movie"
    movie_sim = cosine_similarity(sample_train_sparse_matrix[:, movie].T, sample_t
    top_sim_movies = movie_sim.argsort()[:-1][1:] # we are ignoring 'The User' fi
    # get the ratings of most similar movie rated by this user..
    top_ratings = sample_train_sparse_matrix[user, top_sim_movies].toarray().ravel()
    # we will make it's length "5" by adding user averages to.
    top_sim_movies_ratings = list(top_ratings[top_ratings != 0][:5])
    top_sim_movies_ratings.extend([sample_train_averages['user'][user]]*(5-len(top_si
    #print(top_sim_movies_ratings)
except (IndexError, KeyError):
    #print(top_sim_movies_ratings, end=" : -- ")
    top_sim_movies_ratings.extend([sample_train_averages['global']]*(5-len(top_si
    #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)
# finally, 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)

```

→ It is already created...

Reading from the file to make a test dataframe

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

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

- **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 simiilar movies rated by this movie..)
- **UAvg** : User AVerage rating
- **MAvg** : Average rating of this movie
- **rating** : Rating of this movie by this user.

Double-click (or enter) to edit

4.3.2 Transforming data for Surprise models

```
!pip install surprise
from surprise import Reader, Dataset
```

⇨

Collecting surprise

Downloading <https://files.pythonhosted.org/packages/61/de/e5cba8682201fcf9c3719a6fdd>

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 separate format for TRAIN and TEST data, which will be useful for training the models like SVD, KNNBaseline...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

```
Installing collected packages: scikit-surprise, surprise
```

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

# create the traindata from the dataframe...
train_data = Dataset.load_from_df(reg_train[['user', 'movie', 'rating']], reader)

# build the trainset from traindata.., It is of dataset format from surprise library..
trainset = train_data.build_full_trainset()
```

4.3.2.2 Transforming test data

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

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

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

4.4 Applying Machine Learning models

Double-click (or enter) to edit

- 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)

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

models_evaluation_train, models_evaluation_test
```

⇒

Double-click (or enter) to edit

Utility functions for running regression models

```
# 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

```

# it is just to makesure that all of our algorithms should produce same results
# everytime they run...

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

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

    return actual, pred

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

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

    return rmse, mape*100

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

    return train_dict, test_dict

    It returns two dictionaries, one for train and the other is for test
    Each of them have 3 key-value pairs, which specify 'rmse', 'mape', and 'predicted'
    ...

    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 : {}'.format(datetime.now()-st))

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

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

    #store them in the train dictionary

```

```

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

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

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

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

# return two dictionaries train and test
return train, test

```

Double-click (or enter) to edit

4.4.1 XGBoost with initial 13 features

```

from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import TimeSeriesSplit
import xgboost as xgb

# Hyperparameters
parameters = {'max_depth':[1,2,3],
              'learning_rate':[0.001,0.01,0.1],
              'n_estimators':[100,300,500,700]}

# 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']

start = datetime.now()
# initialize Our first XGBoost model...
first_xgb = xgb.XGBRegressor(nthread=-1)

# Cross validation
gsv = GridSearchCV(first_xgb,
                    param_grid = parameters,
                    scoring="neg_mean_squared_error",

```

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

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

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

➡

Fitting 5 folds for each of 36 candidates, totalling 180 fits

Double-click (or enter) to edit

```
[Parallel(n_jobs=-1): done 180 out of 180 | elapsed: 19.9min finished]
```

```
xgb_bsl = xgb.XGBRegressor(max_depth=3, learning_rate = 0.1, n_estimators=100, nthread=-1)
```

```
↳ XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
    colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
    max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
    n_jobs=1, nthread=-1, objective='reg:linear', random_state=0,
    reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
    silent=True, subsample=1)
```

```
-----
```

```
train_results, test_results = run_xgboost(xgb_bsl, x_train, y_train, x_test, y_test)
```

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

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

```
↳ Training the model..
```

```
Done. Time taken : 0:00:06.609408
```

```
Done
```

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

```
Evaluating Test data
```

```
TEST DATA
```

```
-----
```

```
RMSE : 1.0761851474385373
MAPE : 34.504887593204884
```

```
-0.717589 (0.026650) with {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
```

4.4.2 Surprise BaselineModel

```
-----
```

```
from surprise import BaselineOnly
```

Predicted_rating : (baseline prediction)

```
-
```

```
http://surprise.readthedocs.io/en/stable/basic_algorithms.html#surprise.predict
ion_algorithms.baseline_only.BaselineOnly
```

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

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

Optimization function (Least Squares Problem)

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

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

```
# options are to specify.., how to compute those user and item biases
bsl_options = {'method': 'sgd',
               'reg': 0.01,
               'learning_rate': 0.001,
               'n_epochs': 120
              }
bsl_algo = BaselineOnly(bsl_options=bsl_options)
# run this algorithm.., It will return the train and test results..
bsl_train_results, bsl_test_results = run_surprise(bsl_algo, trainset, testset, verbose=True)

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

```
↳ Training the model...
Estimating biases using sgd...
Done. time taken : 0:00:06.571262

Evaluating the model with train data..
time taken : 0:00:01.249900
-----
Train Data
-----
RMSE : 0.8883104307239651

MAPE : 27.274344731601268

adding train results in the dictionary..

Evaluating for test data...
time taken : 0:00:00.211147
-----
Test Data
-----
RMSE : 1.0725953793894922

MAPE : 35.01055341888157

storing the test results in test dictionary..

-----
Total time taken to run this algorithm : 0:00:08.034151
```

Double-click (or enter) to edit

4.4.3 XGBoost with initial 13 features + Surprise Baseline predictor

Updating Train Data

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

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	...
0	53406	33	3.581679	4.0	5.0	5.0	4.0	1.0	5.0	2.0	5.0	3.0	1.0	...
1	99540	33	3.581679	5.0	5.0	5.0	4.0	5.0	3.0	4.0	4.0	3.0	5.0	...

Updating Test Data

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

reg_test_df.head(2)
```

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	...
0	808635	71	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	...
1	941866	71	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	...

```
import xgboost as xgb
# prepare train data
x_train = reg_train.drop(['user', 'movie', 'rating'], axis=1)
y_train = reg_train['rating']

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

# initialize Our first XGBoost model...
start = datetime.now()

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

# Cross validation
gsv = GridSearchCV(xgb,
                    param_grid = parameters,
                    scoring="neg_mean_squared_error",
                    cv = TimeSeriesSplit(n_splits=5),
                    n_jobs = -1,
                    verbose = 1)
gsv_result = gsv.fit(x_train, y_train)

# Summarizing results
print("Best: %f using %s" % (gsv_result.best_score_, gsv_result.best_params_))
print()
```

```

means = gsv_result.cv_results_['mean_test_score']
stds = gsv_result.cv_results_['std_test_score']
params = gsv_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

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

```

⌚ Fitting 5 folds for each of 36 candidates, totalling 180 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.

[Parallel(n_jobs=-1)]: Done 46 tasks | elapsed: 4.3min

[Parallel(n_jobs=-1)]: Done 180 out of 180 | elapsed: 22.7min finished

Best: -0.718201 using {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}

-9.003600 (0.421070) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 10}

-6.396660 (0.323360) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 30}

-4.635431 (0.245576) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 50}

-3.444588 (0.195216) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 70}

-8.968829 (0.374920) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 10}

-6.330400 (0.267078) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 30}

-4.550069 (0.191141) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 50}

-3.349752 (0.140934) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 70}

-8.950769 (0.371915) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 10}

-6.284471 (0.244525) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 30}

-4.494928 (0.168086) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 50}

-3.290015 (0.117240) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 70}

-2.322729 (0.136005) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 100}

-0.891127 (0.051646) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 300}

-0.800698 (0.043965) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 500}

-0.767167 (0.038827) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 700}

-2.224923 (0.096179) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 100}

-0.800935 (0.035338) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 300}

-0.739654 (0.031972) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 500}

-0.726500 (0.029573) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 700}

-2.157529 (0.065650) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100}

-0.767979 (0.029763) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 300}

-0.724202 (0.028836) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 500}

-0.719040 (0.027658) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 700}

-0.741297 (0.033748) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 100}

-0.721238 (0.027466) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 300}

-0.721127 (0.027470) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 500}

-0.721381 (0.027617) with: {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 700}

-0.720922 (0.027313) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 100}

-0.718334 (0.026914) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 300}

-0.718794 (0.027844) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 500}

-0.719570 (0.028418) with: {'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 700}

-0.718201 (0.027065) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}

-0.718827 (0.028910) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 300}

-0.720730 (0.030690) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 500}

-0.722689 (0.032462) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 700}

Time Taken: 0:22:48.697097

Double-click (or enter) to edit

```
import xgboost as xgb
xgb_bsl = xgb.XGBRegressor(max_depth=3, learning_rate = 0.1, n_estimators=100, nthread=-1)
xgb_bsl
```

↳ XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1, colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0, max_depth=3, min_child_weight=1, missing=None, n_estimators=100, n_jobs=1, nthread=-1, objective='reg:linear', random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None, silent=True, subsample=1)

```
train_results, test_results = run_xgboost(xgb_bsl, x_train, y_train, x_test, y_test)
```

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

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

↳ Training the model..
Done. Time taken : 0:00:07.398671

Done

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

TEST DATA

```
-----  
RMSE : 1.0762300237782054  
MAPE : 34.5017727794818
```

Double-click (or enter) to edit

4.4.4 Surprise KNNBaseline predictor

```
from surprise import KNNBaseline
```

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

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

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

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

- **b_{ui}** - Baseline prediction of (user,movie) rating
- **$N_i^k(u)$** - Set of **K similar** users (neighbours) of **user (u)** who rated **movie(i)**
- **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)

Double-click (or enter) to edit

- **Predicted rating (based on Item Item similarity):**

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

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

4.4.4.1 Surprise KNNBaseline with user user similarities

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

knn_bsl_u = KNNBaseline(k=40, sim_options = sim_options, bsl_options = bsl_options)
knn_bsl_u_train_results, knn_bsl_u_test_results = run_surprise(knn_bsl_u, trainset, testset, 'user')

# 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 : 0:00:35.523655
```

```
Evaluating the model with train data..
time taken : 0:02:13.628979
```

```
-----
Train Data
-----
```

```
RMSE : 0.33642097416508826
```

```
MAPE : 9.145093375416348
```

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

```
Evaluating for test data...
time taken : 0:00:00.088311
```

```
-----
Test Data
-----
```

```
RMSE : 1.0726493739667242
```

```
MAPE : 35.02094499698424
```

4.4.4.2 Surprise KNNBaseline with movie movie similarities

```
# we specify , how to compute similarities and what to consider with sim_options to our algos
# 'user_based' : False => this considers the similarities of movies instead of users

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



```
knn_bsl_m = KNNBaseline(k=40, sim_options = sim_options, bsl_options = bsl_options)
knn_bsl_m_train_results, knn_bsl_m_test_results = run_surprise(knn_bsl_m, trainset, testset, 'KNN')

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



```
Training the model...
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Done. time taken : 0:00:01.581487
```

```
Evaluating the model with train data..
time taken : 0:00:10.509544
```

```
-----
Train Data
-----
```

```
RMSE : 0.32584796251610554
```

```
MAPE : 8.447062581998374
```

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

```
Evaluating for test data...
time taken : 0:00:00.084750
```

```
-----
Test Data
-----
```

```
RMSE : 0.32584796251610554
```

Double-click (or enter) to edit

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

- ◦ ▪ First we will run XGBoost with predictions from both KNN's (that uses User_User and Item_Item similarities along with our previous features.
- ◦ ▪ Then we will run XGBoost with just predictions from both knn models and predictions from our baseline model.

Preparing Train data

```
# 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)
```

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5
0	53406	33	3.581679	4.0	5.0	5.0	4.0	1.0	5.0	2.0	5.0	3.0	1.0
1	99540	33	3.581679	5.0	5.0	5.0	4.0	5.0	3.0	4.0	4.0	3.0	5.0

Preparing Test data

```
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)
```

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1
0	808635	71	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679
1	941866	71	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679	3.581679

```
import xgboost as xgb
# 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']

start = datetime.now()

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

# Cross validation
gsv = GridSearchCV(model,
                    param_grid = parameters,
                    scoring="neg_mean_squared_error",
                    cv = TimeSeriesSplit(n_splits=5),
                    n_jobs = -1,
                    verbose = 1)
gsv_result = gsv.fit(x_train, y_train)

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

→

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  46 tasks      | elapsed:  5.4min
[Parallel(n_jobs=-1)]: Done 180 out of 180 | elapsed: 28.6min finished
Best: -0.718291 using {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
```

```
-9.003600 (0.421070) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 10
-6.396660 (0.323360) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 30
-4.635431 (0.245576) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 50
-3.444588 (0.195216) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 70
-8.968829 (0.374920) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 10
-6.330400 (0.267078) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 30
-4.550069 (0.191141) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 50
-3.349752 (0.140934) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 70
-8.950769 (0.371915) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 10
-6.284471 (0.244525) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 30
-4.494928 (0.168086) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 50
-3.290015 (0.117240) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 70
-2.322729 (0.136005) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 100
-0.891127 (0.051646) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 300
-0.800698 (0.043965) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 500
-0.767167 (0.038827) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 700
-2.224923 (0.096179) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 100
-0.800935 (0.035338) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 300
-0.739654 (0.031972) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 500
-0.726516 (0.029594) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 700
-0.718291 (0.027260) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
```

```
import xgboost as xgb
xgb_knn_bsl = xgb.XGBRegressor(max_depth=3, learning_rate = 0.1, n_estimators=100, nthread=-1)
xgb_knn_bsl
```

```
↳ XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
    colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
    max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
    n_jobs=1, nthread=-1, objective='reg:linear', random_state=0,
    reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
    silent=True, subsample=1)
```

```
-0.718291 (0.027260) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
train_results, test_results = run_xgboost(xgb_knn_bsl, x_train, y_train, x_test, y_test)
```

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

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

```
↳
```

Training the model..
Done. Time taken : 0:00:09.333497

Done

Evaluating the model with TRAIN data...

4.4.6 Matrix Factorization Techniques

4.4.6.1 SVD Matrix Factorization User Movie interactions

```
from surprise import SVD
```

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

- Predicted Rating :
 - - $\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$
 - q_i - Representation of item(movie) in latent factor space
 - p_u - Representation of user in new latent factor space
- A BASIC MATRIX FACTORIZATION MODEL in [https://datajobs.com/data-science-repo/Recommender-Systems-\[Netflix\].pdf](https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf)
- Optimization problem with user item interactions and regularization (to avoid overfitting)
 - - $\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda (b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2)$

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

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



Training the model...

```
Processing epoch 0
Processing epoch 1
Processing epoch 2
Processing epoch 3
Processing epoch 4
Processing epoch 5
Processing epoch 6
Processing epoch 7
Processing epoch 8
Processing epoch 9
Processing epoch 10
Processing epoch 11
Processing epoch 12
Processing epoch 13
Processing epoch 14
Processing epoch 15
Processing epoch 16
Processing epoch 17
Processing epoch 18
Processing epoch 19
Done. time taken : 0:00:09.778854
```

Evaluating the model with train data..

```
time taken : 0:00:01.560933
```

Train Data

```
RMSE : 0.6574721240954099
```

```
MAPE : 19.704901088660474
```

adding train results in the dictionary..

Evaluating for test data...

```
time taken : 0:00:00.081963
```

Test Data

```
RMSE : 1.0726046873826458
```

Double-click (or enter) to edit

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

```
from surprise import SVDpp
```

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

- **Predicted Rating :**

◦

- $\hat{r}_{ui} = u + b_u + b_i + \sigma^T \left(n^{-\frac{1}{2}} |I_{ui}|^{-\frac{1}{2}} \sum_{j \in I_{ui}} y_j \right)$
- I_u -- the set of all items rated by user u
- y_j -- Our new set of item factors that capture implicit ratings.
- **Optimization problem with user item interactions and regularization (to avoid overfitting)**
 -
 - $\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda (b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2 + \|y_j\|^2)$

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

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



```
Training the model...
processing epoch 0
processing epoch 1
processing epoch 2
processing epoch 3
processing epoch 4
processing epoch 5
processing epoch 6
processing epoch 7
processing epoch 8
processing epoch 9
processing epoch 10
processing epoch 11
```

Double-click (or enter) to edit

```
processing epoch 12
```

Double-click (or enter) to edit

```
processing epoch 13
```

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

```
evaluating the model with train data..
```

Preparing Train data

```
train data

# 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)
```

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	...	smr4	smr5
0	53406	33	3.581679	4.0	5.0	5.0	4.0	1.0	5.0	2.0	...	3.0	1.0
1	99540	33	3.581679	5.0	5.0	5.0	4.0	5.0	3.0	4.0	...	3.0	5.0

2 rows × 21 columns

```
MADE IN 2017012010007
```

Preparing Test data

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

reg_test_df.head(2)
```

```
→
```

user movie Gavg sur1 sur2 sur3 sur4 sur5 smr1

Double-click (or enter) to edit

1 041966 71 3 591670 3 591670 3 591670 3 591670 3 591670 3 591670 3 591670 3 591670 3

```
import xgboost as xgb
# 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']
```

```
start = datetime.now()
```

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

Cross validation

```
gsv = GridSearchCV(model,
                    param_grid = parameters,
                    scoring="neg_mean_squared_error",
                    cv = TimeSeriesSplit(n_splits=5),
                    n_jobs = -1,
                    verbose = 1)
gsv_result = gsv.fit(x_train, y_train)
```

Summarizing results

```
print("Best: %f using %s" % (gsv_result.best_score_, gsv_result.best_params_))
print()
```

```
means = gsv_result.cv_results_['mean_test_score']
stds = gsv_result.cv_results_['std_test_score']
params = gsv_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

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



```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  46 tasks      | elapsed:  6.5min
[Parallel(n_jobs=-1)]: Done 180 out of 180 | elapsed: 34.1min finished
Best: -0.718429 using {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
```

```
-9.003600 (0.421070) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 10
-6.396660 (0.323360) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 30
-4.635431 (0.245576) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 50
-3.444588 (0.195216) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 70
-8.968829 (0.374920) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 10
-6.330400 (0.267078) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 30
-4.550069 (0.191141) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 50
-3.349752 (0.140934) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 70
-8.950769 (0.371915) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 10
-6.284471 (0.244525) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 30
-4.494928 (0.168086) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 50
-3.290015 (0.117240) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 70
-2.322729 (0.136005) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 100
-0.891127 (0.051646) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 300
-0.800698 (0.043965) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 500
-0.767167 (0.038827) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 700
-2.224923 (0.096179) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 100
-0.800935 (0.035338) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 300
-0.739654 (0.031972) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 500
```

```
xgb_final = xgb.XGBRegressor(max_depth=3, learning_rate = 0.1, n_estimators=100, nthread=-1)
xgb_final
```

```
↳ XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
    colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
    max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
    n_jobs=1, nthread=-1, objective='reg:linear', random_state=0,
    reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
    silent=True, subsample=1)
```

```
0.700124 (0.007606) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
train_results, test_results = run_xgboost(xgb_final, x_train, y_train, x_test, y_test)
```

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

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

```
↳
```

Training the model..
Done. Time taken : 0:00:11.354137

4.4.8 XgBoost with Surprise Baseline + Surprise KNNbaseline + MF Techniques

```

TEST DATA

# 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']

start = datetime.now()

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

# Cross validation
gsv = GridSearchCV(model,
                    param_grid = parameters,
                    scoring="neg_mean_squared_error",
                    cv = TimeSeriesSplit(n_splits=5),
                    n_jobs = -1,
                    verbose = 1)
gsv_result = gsv.fit(x_train, y_train)

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

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

```



```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  46 tasks      | elapsed:  3.0min
[Parallel(n_jobs=-1)]: Done 180 out of 180 | elapsed: 16.1min finished
Best: -1.164610 using {'learning_rate': 0.1, 'max_depth': 1, 'n_estimators': 100}
```

```
-9.029998 (0.425032) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 10
-6.473006 (0.352038) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 30
-4.752577 (0.292264) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 50
-3.594032 (0.243208) with: {'learning_rate': 0.001, 'max_depth': 1, 'n_estimators': 70
-9.029755 (0.425032) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 10
-6.472546 (0.352162) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 30
-4.752052 (0.292632) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 50
-3.593538 (0.243638) with: {'learning_rate': 0.001, 'max_depth': 2, 'n_estimators': 70
-9.029730 (0.425077) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 10
-6.472351 (0.352617) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 30
-4.751680 (0.293461) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 50
-3.593233 (0.244425) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 70
-2.511971 (0.184904) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 100
-1.200392 (0.064240) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 300
-1.166743 (0.061822) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 500
-1.164822 (0.062102) with: {'learning_rate': 0.01, 'max_depth': 1, 'n_estimators': 700
-2.511541 (0.185349) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 100
-1.200485 (0.064306) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 300
-1.167093 (0.061770) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 500
-1.165347 (0.062068) with: {'learning_rate': 0.01, 'max_depth': 2, 'n_estimators': 700
-2.511381 (0.186006) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100
-1.200670 (0.061457) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 200
```

```
xgb_all_models = xgb.XGBRegressor(max_depth=1, learning_rate = 0.01, n_estimators=100, nthread=-1)
xgb_all_models
```

```
↳ XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
               colsample_bytree=1, gamma=0, learning_rate=0.01, max_delta_step=0,
               max_depth=1, min_child_weight=1, missing=None, n_estimators=100,
               n_jobs=1, nthread=-1, objective='reg:linear', random_state=0,
               reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
               silent=True, subsample=1)

-1.166189 (0.062513) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
train_results, test_results = run_xgboost(xgb_all_models, x_train, y_train, x_test, y_test)

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

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

```
↳
```

```
Training the model..
Done. Time taken : 0:00:02.292256
```

```
Done
```

4.5 Comparision between all models

```
# 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().
```

```
→ bsl_algo      1.0725953793894922
    svd          1.0726046873826458
    knn_bsl_u    1.0726493739667242
    knn_bsl_m    1.072758832653683
    svdpp        1.0728491944183447
    xgb_bsl      1.0762300237782054
    xgb_knn_bsl  1.0763318643343687
    xgb_final    1.07637090548753
    xgb_all_models 1.5597123428477524
Name: rmse, dtype: object
```

Procedure

1. We have sampled the data because of less computational power. I sampled (25000,3000) training dataset and (9000,1500) testing dataset.
2. After sampling the data we featurized the data for regression and for surprise library.
3. After that we have applied XGBRegressor each time with different set of features generated after featurization with tuned hyperparameter.
4. At last we will compare the model performance using RMSE.

Conclusion

1. There is very small difference in 'RMSE' score, And this can be improved by using the whole dataset.
2. bsl_algo model is best among all the models we tried.

