Netflix Movie Recommendation System

Business Problem

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 (https://www.netflixprize.com/rules.html)

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.)

Sources

- https://www.netflixprize.com/rules.html (https://www.netflixprize.com/rules.html)
- https://www.kaggle.com/netflix-inc/netflix-prize-data (https://www.kaggle.com/netflix-inc/netflix-prize-data)
- Netflix blog: https://medium.com/netflix-techblog/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429 (https://medium.com/netflix-techblog/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429) (very nice blog)
- surprise library: http://surpriselib.com/ (http://surpriselib.com/) (we use many models from this library)
- surprise library doc: http://surprise.readthedocs.io/en/stable/getting_started.html (http://surprise.readthedocs.io/en/stable/getting_started.html) (we use many models from this library)
- installing surprise: https://github.com/NicolasHug/Surprise#installation (https://github.com/NicolasHug/Surprise#installation)

• Research paper: http://courses.ischool.berkeley.edu/i290-dm/s11/SECURE/a1-koren.pdf (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 (https://www.youtube.com/watch?v=P5mlg91as1c)

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. There is no low latency requirement as the recommended movies can be precomputed earlier.

Type of Data:

• There are 17770 unique movie IDs.
• There are 480189 unique user IDs.
• There are ratings. Ratings are on a five star (integral) scale from 1 to 5.

Data Overview

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 customerID, rating from a customer and its date.

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
```

Mapping the real world problem to a Machine Learning Problem

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

Performance metric

Mean Absolute Percentage Error

Root Mean Square Error

Machine Learning Objective and Constraints

1. Try to Minimize RMSE
2. Provide some form of interpretability

In [76]:
```
from datetime import datetime
import pandas as pd
import numpy as np
import seaborn as sns
sns.set_style("whitegrid")
import os
import random
import matplotlib
import matplotlib.pyplot as plt
from scipy import sparse
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.metrics import mean_squared_error


import xgboost as xgb
from surprise import Reader, Dataset
from surprise import BaselineOnly
from surprise import KNNBaseline
from surprise import SVD
from surprise import SVDpp
from surprise.model_selection import GridSearchCV
```

1. Reading and Storing Data

Data Pre-processing

In [14]:
```
if not os.path.isfile("../Data/NetflixRatings.csv"):
#This line: "os.path.isfile("../Data/NetflixRatings.csv")" simply checks that is there a file with the name "NetflixRatin
#in the folder "/Data/". If the file is present then it return true else false
    startTime = datetime.now()
    data = open("../Data/NetflixRatings.csv", mode = "w") #this line simply creates the file with the name "NetflixRating
    #write mode in the folder "Data".
#    files = ['../Data/combined_data_1.txt','../Data/combined_data_2.txt', '../Data/combined_data_3.txt', '../Data/combi
    files = ['../Data/combined_data_2.txt', '../Data/combined_data_4.txt']
    for file in files:
        print("Reading from file: "+str(file)+"...")
        with open(file) as f:  #you can think of this command "with open(file) as f" as similar to 'if' statement or a so
            #loop statement. This command says that as long as this file is opened, perform the underneath operation.
            for line in f:
                line = line.strip() #line.strip() clears all the leading and trailing spaces from the string, as here eac
                #that we are reading from a file is a string.
                #Note first line consist of a movie id followed by a semi-colon, then second line contains custID,rating,
                #then third line agains contains custID,rating,date which belong to that movie ID and so on. The format o
                #is exactly same as shown above with the heading "Example Data Point". Check out above.
                if line.endswith(":"):
                    movieID = line.replace(":", "") #this will remove the trailing semi-colon and return us the leading m
                else:
                    #here, in the below code we have first created an empty list with the name "row "so that we can inser
                    #at the first position and rest customerID, rating and date in second position. After that we have se
                    #four namely movieID, custID, rating and date with comma and converted a single string by joining the
                    #then finally written them to our output ".csv" file.
                    row = []
                    row = [x for x in line.split(",")] #custID, rating and date are separated by comma
                    row.insert(0, movieID)
                    data.write(",".join(row))
                    data.write("\n")
        print("Reading of file: "+str(file)+" is completed\n")
    data.close()
    print("Total time taken for execution of this code = "+str(datetime.now() - startTime))
```

```
Reading from file: ../Data/combined_data_2.txt...
Reading of file: ../Data/combined_data_2.txt is completed

Reading from file: ../Data/combined_data_4.txt...
Reading of file: ../Data/combined_data_4.txt is completed
```

```
Total time taken for execution of this code = 0:03:48.924208
```

In [15]:
```
# creating data frame from our output csv file.
if not os.path.isfile("../Data/NetflixData.pkl"):
    startTime = datetime.now()
    Final_Data = pd.read_csv("../Data/NetflixRatings.csv", sep=",", names = ["MovieID","CustID", "Ratings", "Date"])
    Final_Data["Date"] = pd.to_datetime(Final_Data["Date"])
    Final_Data.sort_values(by = "Date", inplace = True)
    print("Time taken for execution of above code = "+str(datetime.now() - startTime))
```

```
Time taken for execution of above code = 0:01:11.269949
```

In [16]:
```
# storing pandas dataframe as a picklefile for later use
if not os.path.isfile("../Data/NetflixData.pkl"):
    Final_Data.to_pickle("../Data/NetflixData.pkl")
else:
    Final_Data = pd.read_pickle("../Data/NetflixData.pkl")
```

In [17]:
```
Final_Data.head()
```

Out[17]:

	MovieID	CustID	Ratings	Date
49557332	17064	510180	2	1999-11-11
46370047	16465	510180	3	1999-11-11
22463125	8357	510180	4	1999-11-11
35237815	14660	510180	2	1999-11-11
21262258	8079	510180	2	1999-11-11

In [18]:
```
Final_Data.describe()["Ratings"]
```

Out[18]:
```
count    5.382511e+07
mean     3.606058e+00
std      1.082326e+00
min      1.000000e+00
25%      3.000000e+00
50%      4.000000e+00
75%      4.000000e+00
max      5.000000e+00
Name: Ratings, dtype: float64
```

Checking for NaN

In [19]:
```
print("Number of NaN values = "+str(Final_Data.isnull().sum()))
```

```
Number of NaN values = MovieID    0
CustID     0
Ratings    0
Date       0
dtype: int64
```

Removing Duplicates

In [20]:
```
duplicates = Final_Data.duplicated(["MovieID","CustID", "Ratings"])
print("Number of duplicate rows = "+str(duplicates.sum()))
```

```
Number of duplicate rows = 0
```

Basic Statistics

In [22]:
```
print("Total Data:")
print("Total number of movie ratings = "+str(Final_Data.shape[0]))
print("Number of unique users = "+str(len(np.unique(Final_Data["CustID"]))))
print("Number of unique movies = "+str(len(np.unique(Final_Data["MovieID"]))))
```

```
Total Data:
Total number of movie ratings = 53825114
Number of unique users = 478723
Number of unique movies = 9114
```

Spliting data into Train and Test(80:20)

In [2]:
```
if not os.path.isfile("../Data/TrainData.pkl"):
    Final_Data.iloc[:int(Final_Data.shape[0]*0.80)].to_pickle("../Data/TrainData.pkl")
    Train_Data = pd.read_pickle("../Data/TrainData.pkl")
    Train_Data.reset_index(drop = True, inplace = True)
else:
    Train_Data = pd.read_pickle("../Data/TrainData.pkl")
    Train_Data.reset_index(drop = True, inplace = True)

if not os.path.isfile("../Data/TestData.pkl"):
    Final_Data.iloc[int(Final_Data.shape[0]*0.80):].to_pickle("../Data/TestData.pkl")
    Test_Data = pd.read_pickle("../Data/TestData.pkl")
    Test_Data.reset_index(drop = True, inplace = True)
else:
    Test_Data = pd.read_pickle("../Data/TestData.pkl")
    Test_Data.reset_index(drop = True, inplace = True)
```

Basic Statistics in Train data

In [3]:
```
Train_Data.head()
```

Out[3]:

	MovieID	CustID	Ratings	Date
0	17064	510180	2	1999-11-11
1	16465	510180	3	1999-11-11
2	8357	510180	4	1999-11-11
3	14660	510180	2	1999-11-11
4	8079	510180	2	1999-11-11

In [24]:
```
print("Total Train Data:")
print("Total number of movie ratings in train data = "+str(Train_Data.shape[0]))
print("Number of unique users in train data = "+str(len(np.unique(Train_Data["CustID"]))))
print("Number of unique movies in train data = "+str(len(np.unique(Train_Data["MovieID"]))))
print("Highest value of a User ID = "+str(max(Train_Data["CustID"].values)))
print("Highest value of a Movie ID = "+str(max(Train_Data["MovieID"].values)))
```

```
Total Train Data:
Total number of movie ratings in train data = 43060091
Number of unique users in train data = 401901
Number of unique movies in train data = 8931
Highest value of a User ID = 2649429
Highest value of a Movie ID = 17770
```

Basic Statistics in Test data

In [5]:
```
Test_Data.head()
```

Out[5]:

	MovieID	CustID	Ratings	Date
0	17405	1557557	4	2005-08-09
1	13462	2017421	4	2005-08-09
2	6475	934053	4	2005-08-09
3	6007	1156578	5	2005-08-09
4	5085	2311323	4	2005-08-09

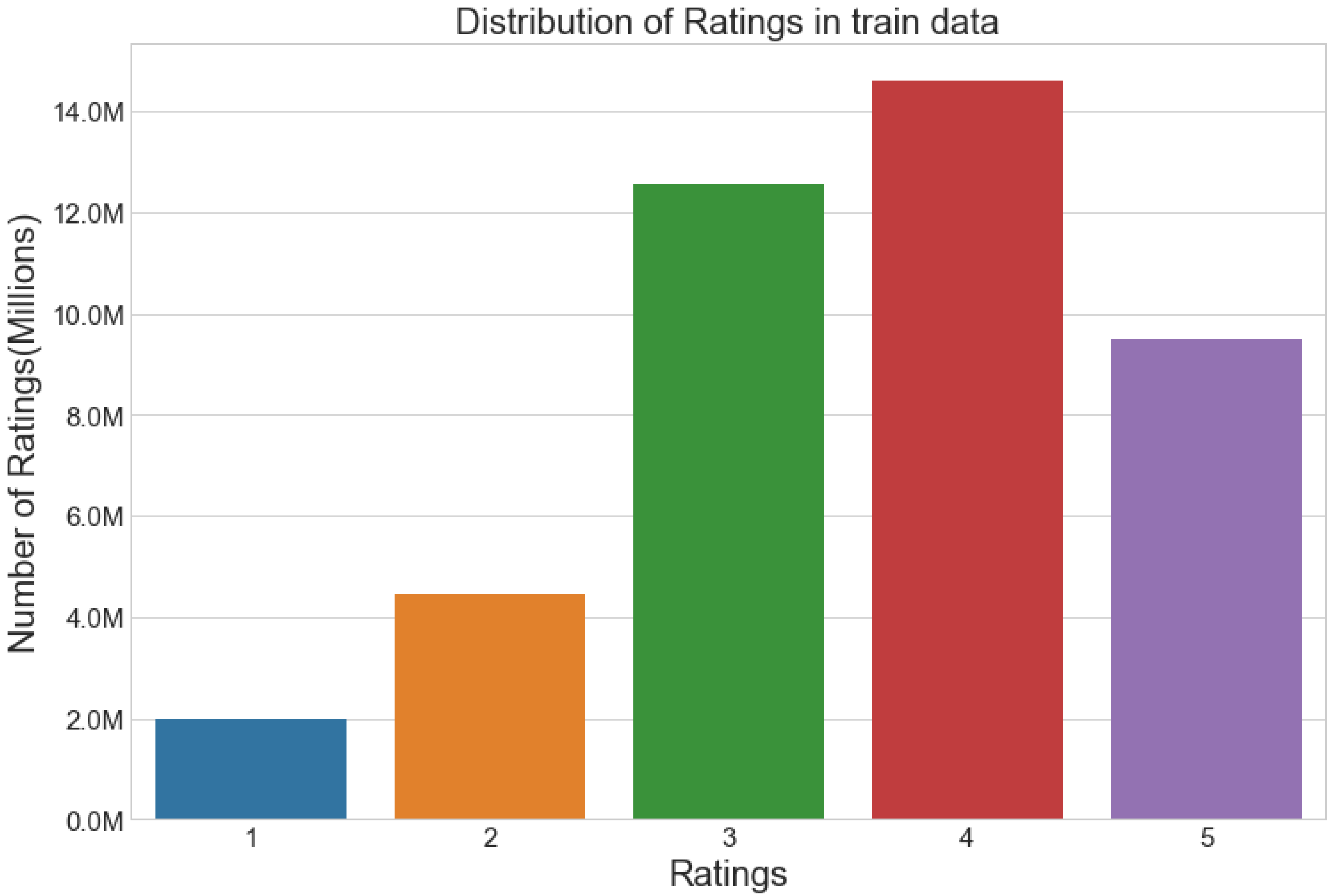
In [25]:
```
print("Total Test Data:")
print("Total number of movie ratings in Test data = "+str(Test_Data.shape[0]))
print("Number of unique users in Test data = "+str(len(np.unique(Test_Data["CustID"]))))
print("Number of unique movies in Test data = "+str(len(np.unique(Test_Data["MovieID"]))))
print("Highest value of a User ID = "+str(max(Test_Data["CustID"].values)))
print("Highest value of a Movie ID = "+str(max(Test_Data["MovieID"].values)))
```

```
Total Test Data:
Total number of movie ratings in Test data = 10765023
Number of unique users in Test data = 327355
Number of unique movies in Test data = 9107
Highest value of a User ID = 2649429
Highest value of a Movie ID = 17770
```

2. Exploratory Data Analysis on Train Data

In [7]:
```
def changingLabels(number):
    return str(number/10**6) + "M"
```

In [115]:
```
plt.figure(figsize = (12, 8))
ax = sns.countplot(x="Ratings", data=Train_Data)

ax.set_yticklabels([changingLabels(num) for num in ax.get_yticks()])

plt.tick_params(labelsize = 15)
plt.title("Distribution of Ratings in train data", fontsize = 20)
plt.xlabel("Ratings", fontsize = 20)
plt.ylabel("Number of Ratings(Millions)", fontsize = 20)
plt.show()
```



In [10]:
```
Train_Data["DayOfWeek"] = Train_Data.Date.dt.weekday_name
```

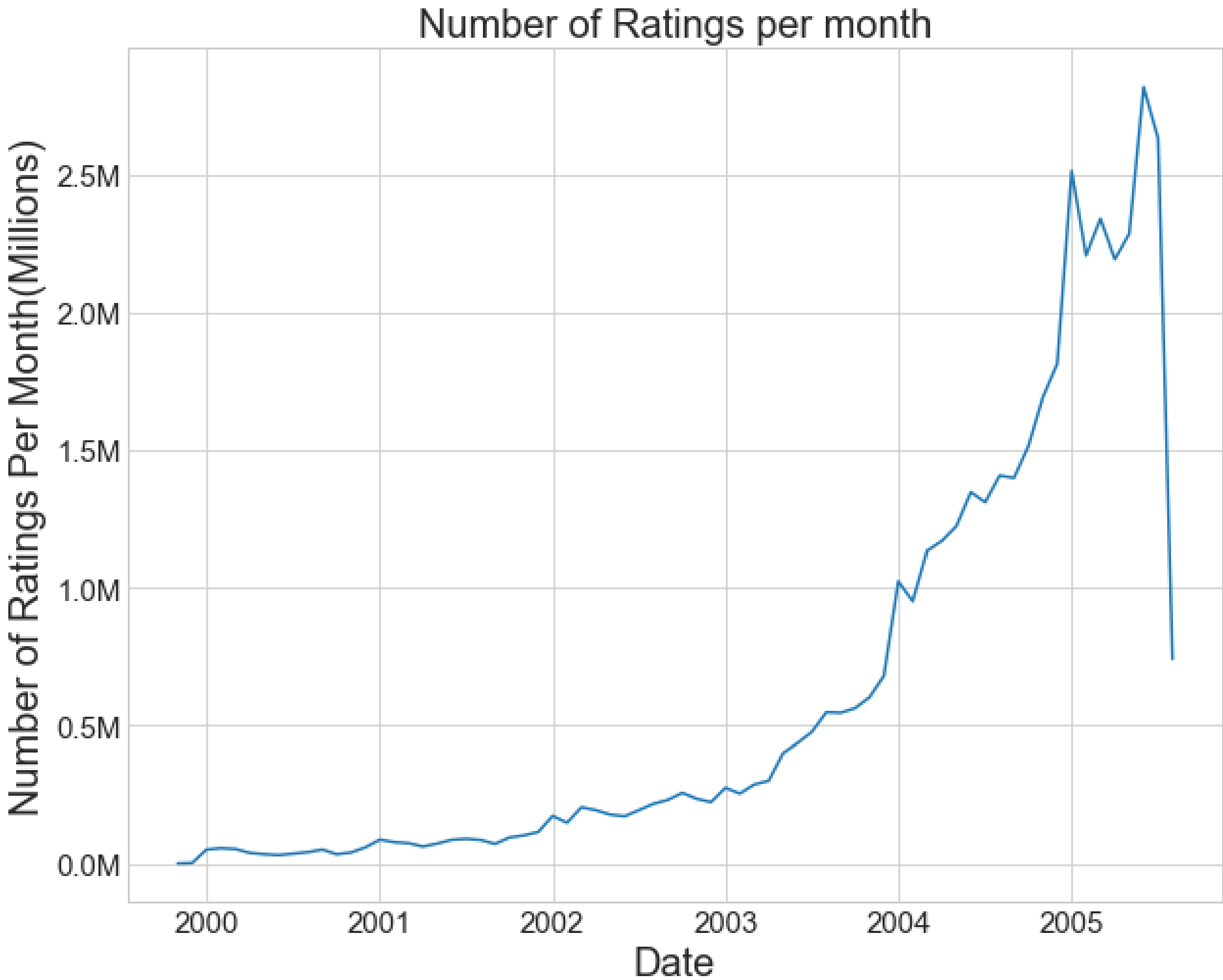
In [11]:
```
Train_Data.tail()
```

Out[11]:

	MovieID	CustID	Ratings	Date	DayOfWeek
43060086	8370	2570992	3	2005-08-09	Tuesday
43060087	17324	60769	4	2005-08-09	Tuesday
43060088	17174	1831297	4	2005-08-09	Tuesday
43060089	5765	1779412	4	2005-08-09	Tuesday
43060090	16922	1367773	5	2005-08-09	Tuesday

Number of Ratings per month

In [28]:
```
plt.figure(figsize = (10,8))
ax = Train_Data.resample("M", on = "Date")["Ratings"].count().plot()
#this above resample() function is a sort of group-by operation.Resample() function can work with dates. It can take mont
#days and years values independently. Here, in parameter we have given "M" which means it will group all the rows Monthly
#"Date" which is already present in the DataFrame. Now after grouping the rows month wise, we have just counted the ratin
#which are grouped by months and plotted them. So, below plot shows that how many ratings are there per month.
#In resample(), we can also give "6M" for grouping the rows every 6-Monthly, we can also give "Y" for grouping
#the rows yearly, we can also give "D" for grouping the rows by day.
#Resample() is a function which is designed to work with time and dates.
#This "Train_Data.resample("M", on = "Date")["Ratings"].count()" returns a pandas series where keys are Dates and values
#counts of ratings grouped by months.You can even check it and print it. Then we are plotting it, where it automatically
#Dates--which are keys on--x-axis and counts--which are values on--y-axis.
ax.set_yticklabels([changingLabels(num) for num in ax.get_yticks()])
ax.set_title("Number of Ratings per month", fontsize = 20)
ax.set_xlabel("Date", fontsize = 20)
ax.set_ylabel("Number of Ratings Per Month(Millions)", fontsize = 20)
plt.tick_params(labelsize = 15)
plt.show()
```



In [104]:
```
#Train_Data.resample("M", on = "Date")["Ratings"].count()
```

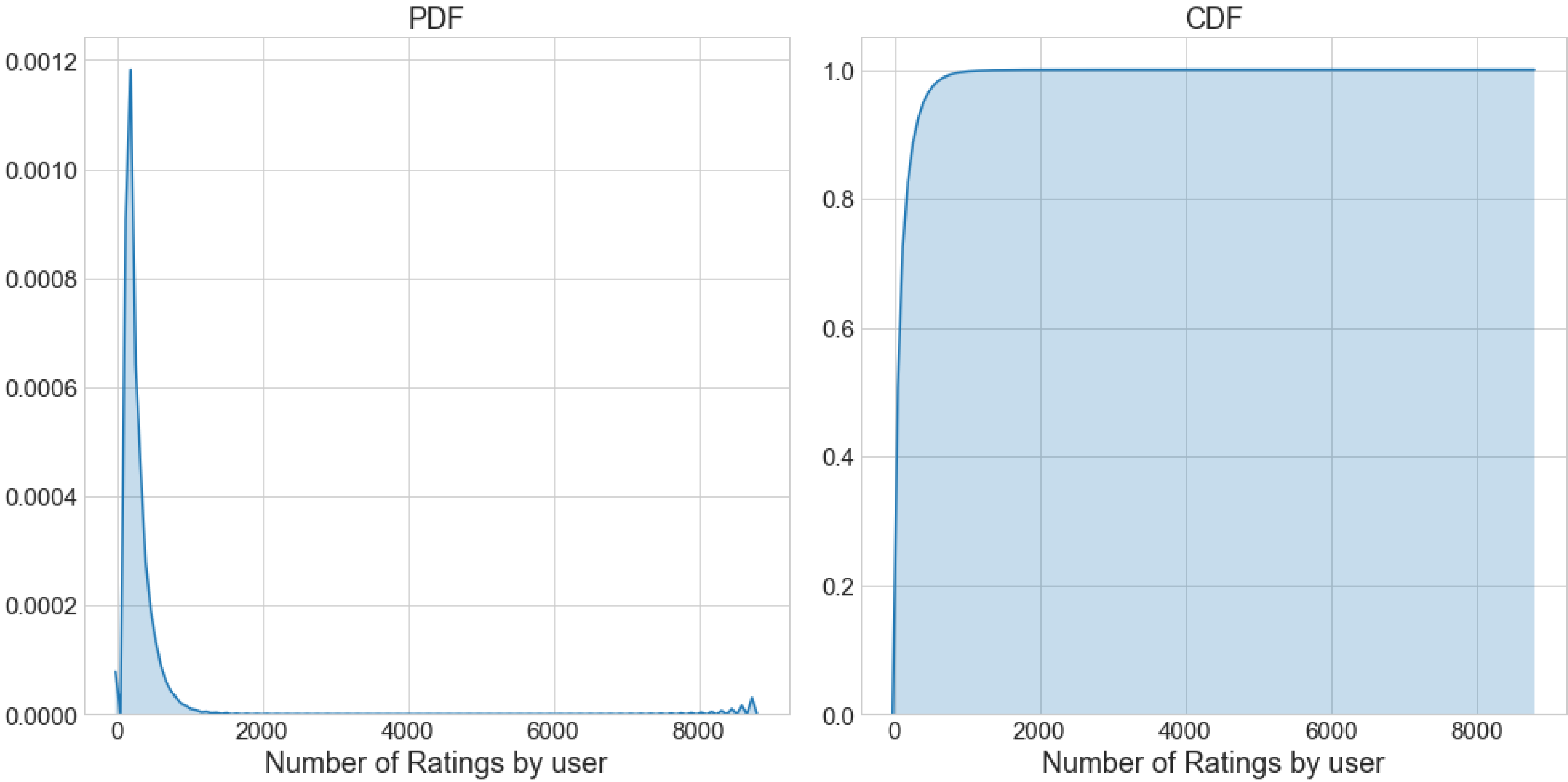
Analysis of Ratings given by user

In [4]:
```
no_of_rated_movies_per_user = Train_Data.groupby(by = "CustID")["Ratings"].count().sort_values(ascending = False)
```

In [86]:
```
no_of_rated_movies_per_user.head()
```

Out[86]:
```
CustID
305344     8779
2439493    8126
387418     7884
1639792    4983
1461435    4846
Name: Ratings, dtype: int64
```

In [124]:
```
fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize=(14,7))

sns.kdeplot(no_of_rated_movies_per_user.values, shade = True, ax = axes[0])
axes[0].set_title("PDF", fontsize = 18)
axes[0].set_xlabel("Number of Ratings by user", fontsize = 18)
axes[0].tick_params(labelsize = 15)

sns.kdeplot(no_of_rated_movies_per_user.values, shade = True, cumulative = True, ax = axes[1])
axes[1].set_title("CDF", fontsize = 18)
axes[1].set_xlabel("Number of Ratings by user", fontsize = 18)
axes[1].tick_params(labelsize = 15)

fig.subplots_adjust(wspace=2)
plt.tight_layout()
plt.show()
```



->Above PDF graph shows that almost all of the users give very few ratings. There are very few users who;s ratings count is high.
->Similarly, above CDF graph shows that almost 99% of users give very few ratings.

In [126]:
```
print("Information about movie ratings grouped by users:")
no_of_rated_movies_per_user.describe()
```

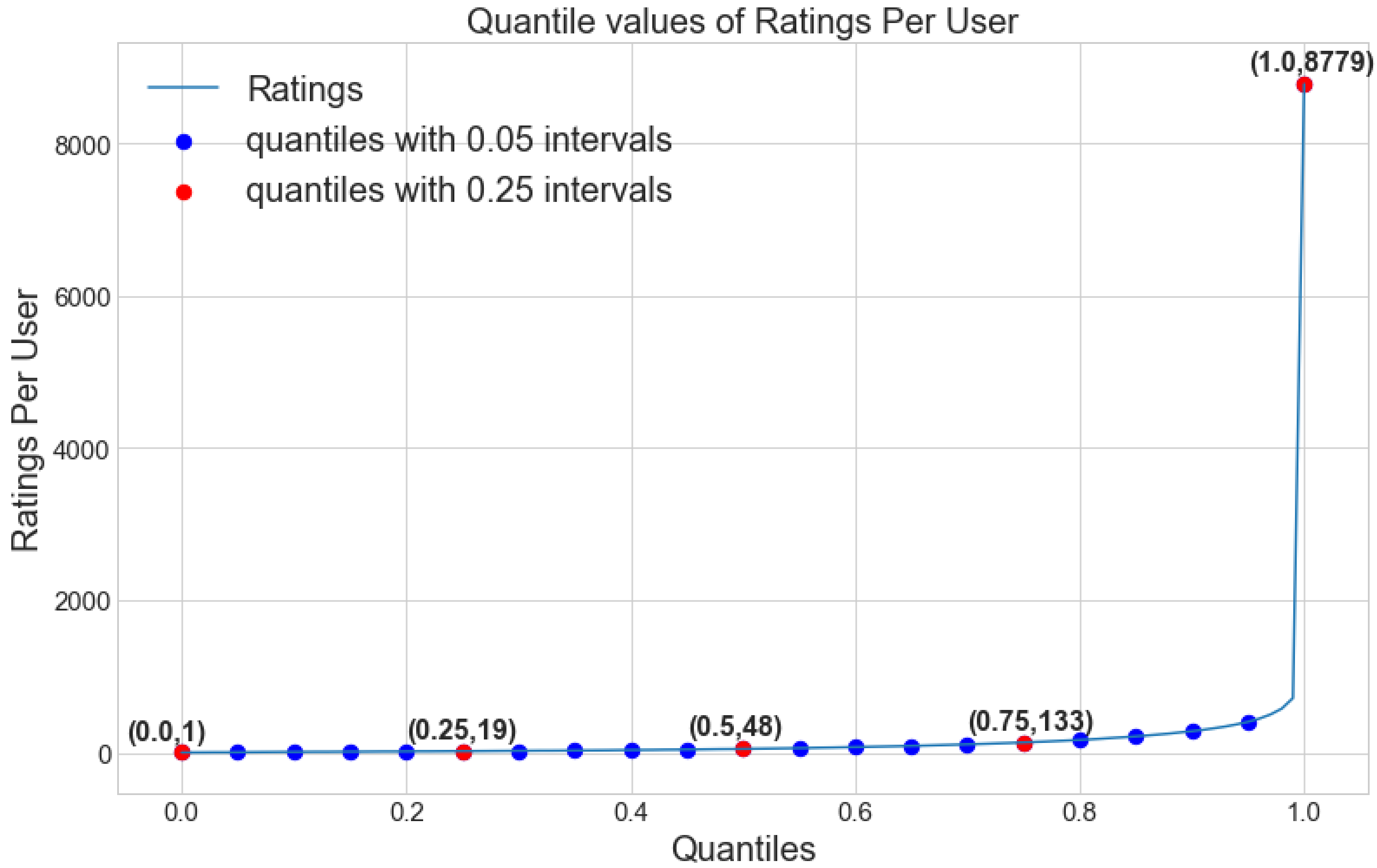
```
Information about movie ratings grouped by users:
```

Out[126]:
```
count    401901.00000
mean        107.14104
std         155.05350
min           1.00000
25%          19.00000
50%          48.00000
75%         133.00000
max        8779.00000
Name: Ratings, dtype: float64
```

In [130]:
```
# no_of_rated_movies_per_user.describe()["75%"]
```

In [161]:
```
quantiles = no_of_rated_movies_per_user.quantile(np.arange(0,1.01,0.01))
```

In [210]:
```
fig = plt.figure(figsize = (10, 6))

axes = fig.add_axes([0.1,0.1,1,1])
axes.set_title("Quantile values of Ratings Per User", fontsize = 20)
axes.set_xlabel("Quantiles", fontsize = 20)
axes.set_ylabel("Ratings Per User", fontsize = 20)
axes.plot(quantiles)

plt.scatter(x = quantiles.index[::5], y = quantiles.values[::5], c = "blue", s = 70, label="quantiles with 0.05 intervals
plt.scatter(x = quantiles.index[::25], y = quantiles.values[::25], c = "red", s = 70, label="quantiles with 0.25 interval
plt.legend(loc='upper left', fontsize = 20)

for x, y in zip(quantiles.index[::25], quantiles.values[::25]):
    plt.annotate(s = '({},{})'.format(x, y), xy = (x, y), fontweight='bold', fontsize = 16, xytext=(x-0.05, y+180))

axes.tick_params(labelsize = 15)
```


Quantile values of Ratings Per User
Ratings
quantiles with 0.05 intervals
quantiles with 0.25 intervals
(1.0,8779)
(0.0,1)
(0.25,19)
(0.5,48)
(0.75,133)
8000
6000
4000
2000
0
Ratings Per User
0.0
0.2
0.4
0.6
0.8
1.0
Quantiles

In [231]:
```
quantiles[::5]
```

Out[231]:
```
0.00       1
0.05       4
0.10       8
0.15      12
0.20      15
0.25      19
0.30      23
0.35      27
0.40      33
0.45      40
0.50      48
0.55      59
0.60      72
0.65      88
0.70     108
0.75     133
0.80     166
0.85     213
0.90     281
0.95     404
1.00    8779
Name: Ratings, dtype: int64
```

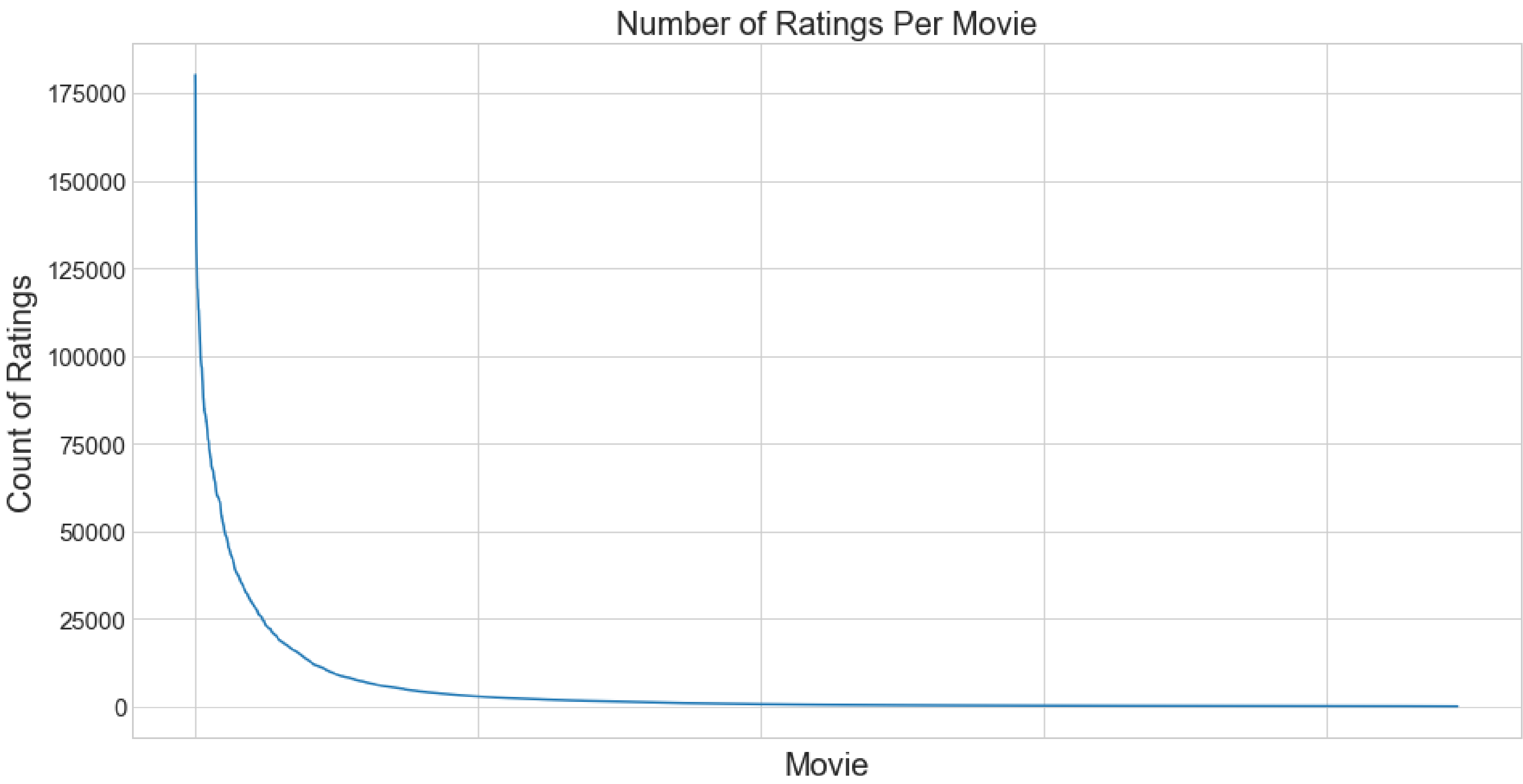
In [233]:
```
print("Total number of ratings below 75th percentile = "+str(sum(no_of_rated_movies_per_user.values<=133)))
print("Total number of ratings above 75th percentile = "+str(sum(no_of_rated_movies_per_user.values>133)))
```

```
Total number of ratings below 75th percentile = 301857
Total number of ratings above 75th percentile = 100044
```

Analysis of Ratings Per Movie

In [234]:
```
no_of_ratings_per_movie = Train_Data.groupby(by = "MovieID")["Ratings"].count().sort_values(ascending = False)
```

In [248]:
```
fig = plt.figure(figsize = (12, 6))
axes = fig.add_axes([0.1,0.1,1,1])
plt.title("Number of Ratings Per Movie", fontsize = 20)
plt.xlabel("Movie", fontsize = 20)
plt.ylabel("Count of Ratings", fontsize = 20)
plt.plot(no_of_ratings_per_movie.values)
plt.tick_params(labelsize = 15)
axes.set_xticklabels([])
plt.show()
```

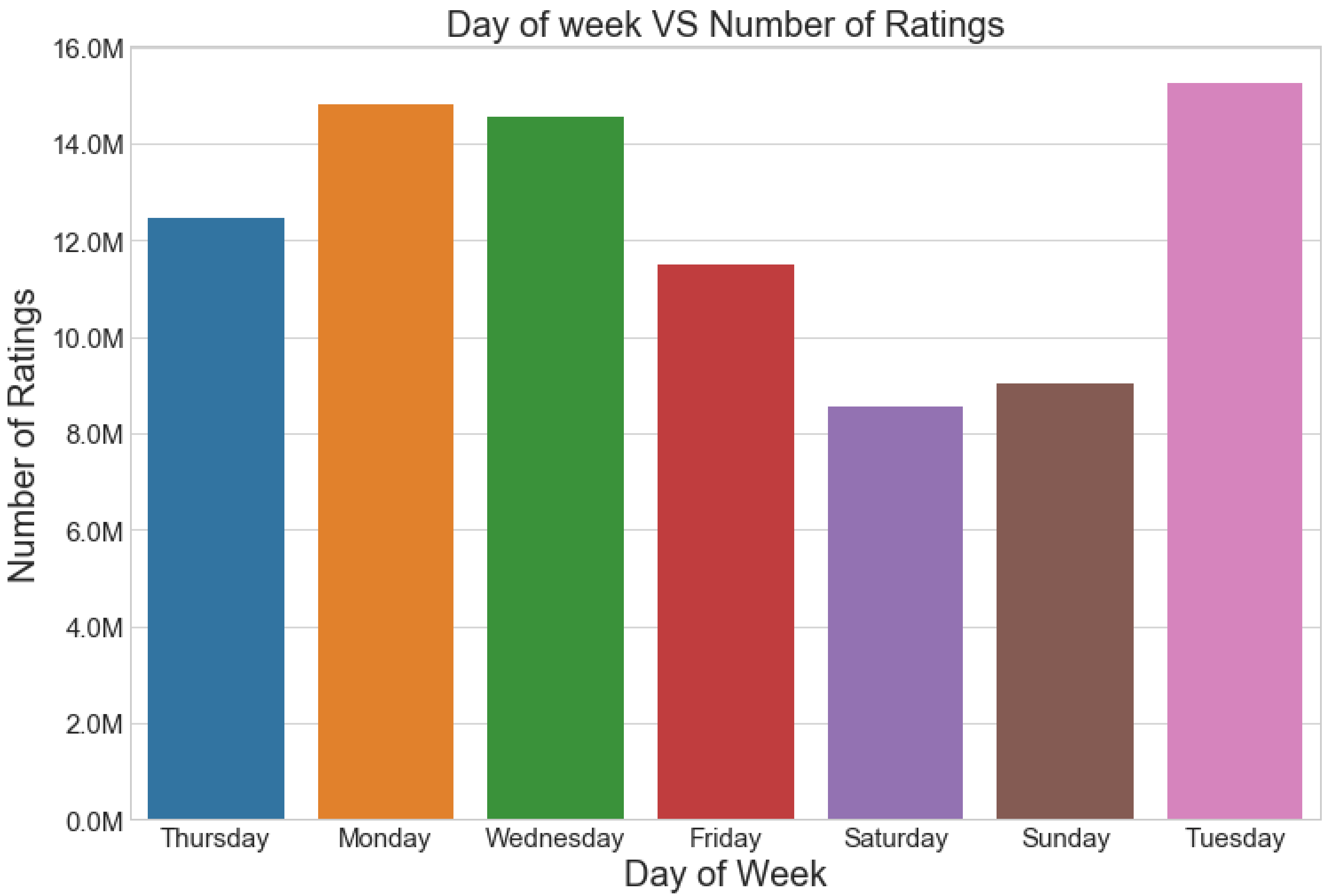


It is very skewed

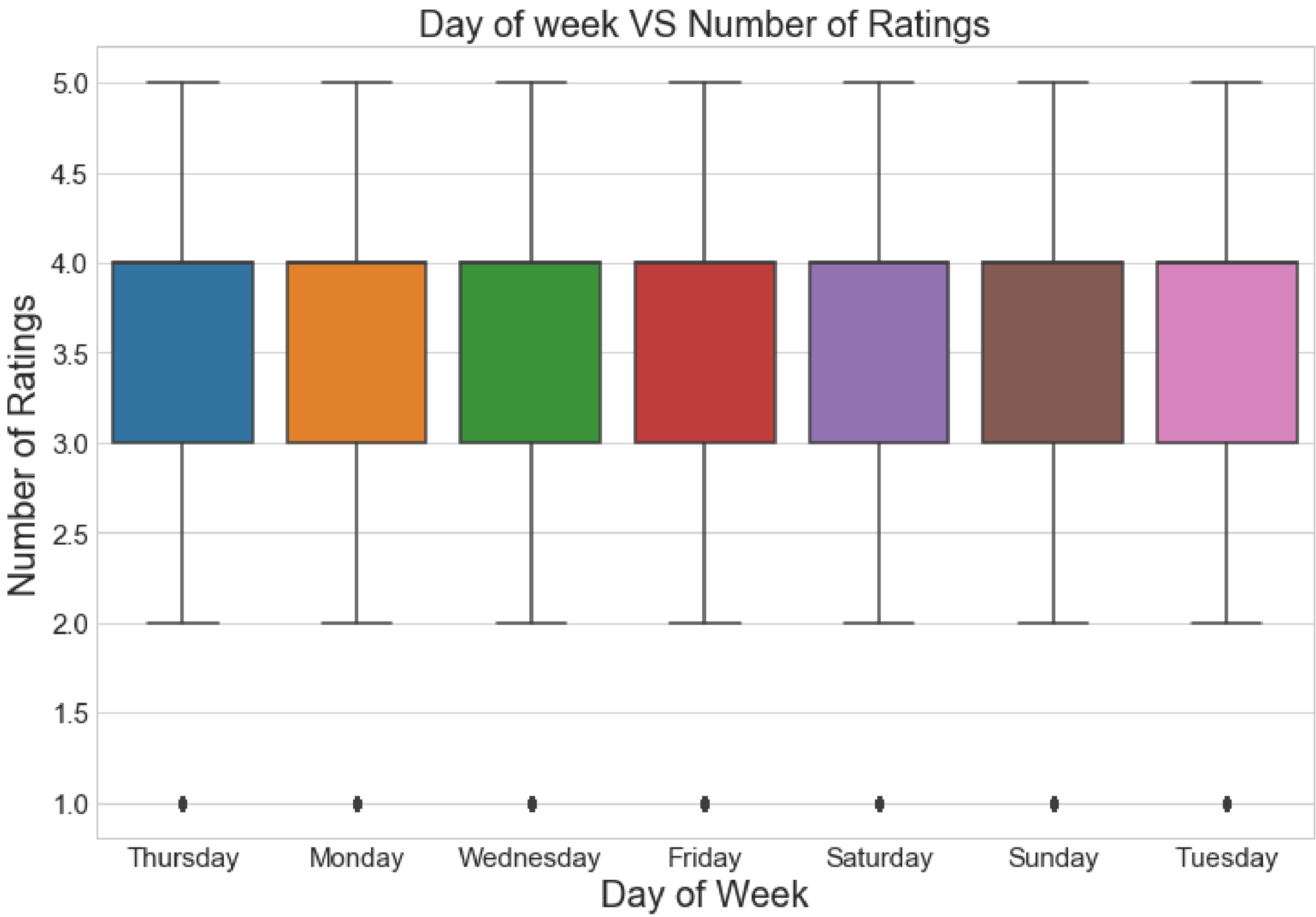
It clearly shows that there are some movies which are very popular and were rated by many users as comapared to other movies

Analysis of Movie Ratings on Day of Week

In [250]:
```
fig = plt.figure(figsize = (12, 8))

axes = sns.countplot(x = "DayOfWeek", data = Train_Data)
axes.set_title("Day of week VS Number of Ratings", fontsize = 20)
axes.set_xlabel("Day of Week", fontsize = 20)
axes.set_ylabel("Number of Ratings", fontsize = 20)
axes.set_yticklabels([changingLabels(num) for num in ax.get_yticks()])
axes.tick_params(labelsize = 15)

plt.show()
```



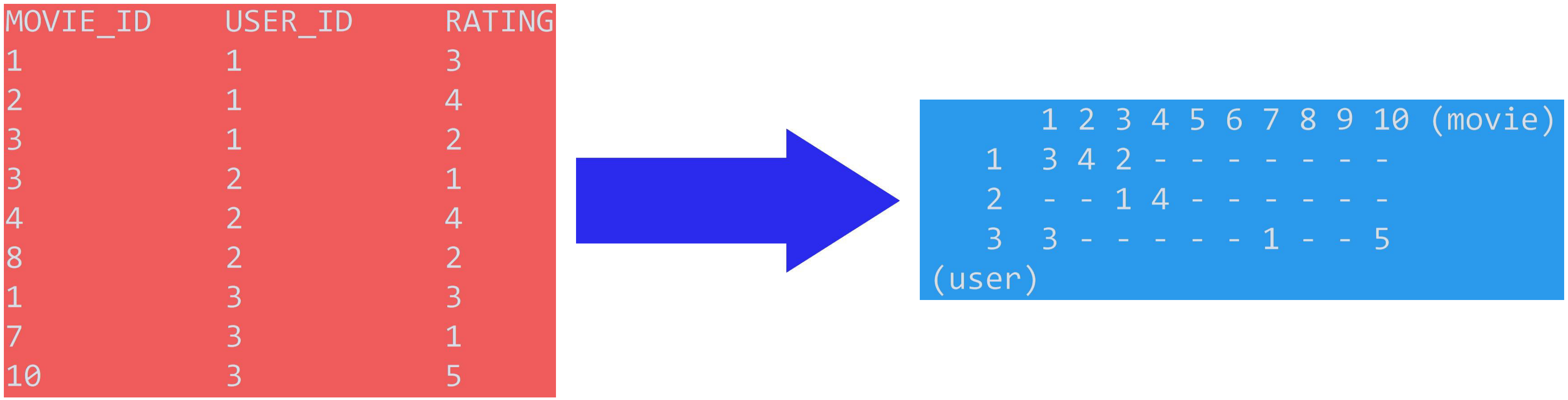
In [15]:
```
fig = plt.figure(figsize = (12, 8))

axes = sns.boxplot(x = "DayOfWeek", y = "Ratings", data = Train_Data)
axes.set_title("Day of week VS Number of Ratings", fontsize = 20)
axes.set_xlabel("Day of Week", fontsize = 20)
axes.set_ylabel("Number of Ratings", fontsize = 20)
axes.tick_params(labelsize = 15)

plt.show()
```



In [14]:
```python
average_ratings_dayofweek = Train_Data.groupby(by = "DayOfWeek")["Ratings"].mean()
print("Average Ratings on Day of Weeks")
print(average_ratings_dayofweek)
```

```
Average Ratings on Day of Weeks
DayOfWeek
Friday       3.589555
Monday       3.577235
Saturday     3.595120
Sunday       3.596637
Thursday     3.583570
Tuesday      3.574852
Wednesday    3.585002
Name: Ratings, dtype: float64
```

3. Creating USER-ITEM sparse matrix from data frame



In [3]:
```
startTime = datetime.now()
print("Creating USER_ITEM sparse matrix for train Data")
if os.path.isfile("../Data/TrainUISparseData.npz"):
    print("Sparse Data is already present in your disk, no need to create further. Loading Sparse Matrix")
    TrainUISparseData = sparse.load_npz("../Data/TrainUISparseData.npz")
    print("Shape of Train Sparse matrix = "+str(TrainUISparseData.shape))

else:
    print("We are creating sparse data")
    TrainUISparseData = sparse.csr_matrix((Train_Data.Ratings, (Train_Data.CustID, Train_Data.MovieID)))
    print("Creation done. Shape of sparse matrix = "+str(TrainUISparseData.shape))
    print("Saving it into disk for furthur usage.")
    sparse.save_npz("../Data/TrainUISparseData.npz", TrainUISparseData)
    print("Done\n")

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

```
Creating USER_ITEM sparse matrix for train Data
Sparse Data is already present in your disk, no need to create further. Loading Sparse Matrix
Shape of Train Sparse matrix = (2649430, 17771)
0:00:02.746893
```

In [4]:
```
startTime = datetime.now()
print("Creating USER_ITEM sparse matrix for test Data")
if os.path.isfile("../Data/TestUISparseData.npz"):
    print("Sparse Data is already present in your disk, no need to create further. Loading Sparse Matrix")
    TestUISparseData = sparse.load_npz("../Data/TestUISparseData.npz")
    print("Shape of Test Sparse Matrix = "+str(TestUISparseData.shape))
else:
    print("We are creating sparse data")
    TestUISparseData = sparse.csr_matrix((Test_Data.Ratings, (Test_Data.CustID, Test_Data.MovieID)))
    print("Creation done. Shape of sparse matrix = "+str(TestUISparseData.shape))
    print("Saving it into disk for furthur usage.")
    sparse.save_npz("../Data/TestUISparseData.npz", TestUISparseData)
    print("Done\n")

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

```
Creating USER_ITEM sparse matrix for test Data
Sparse Data is already present in your disk, no need to create further. Loading Sparse Matrix
Shape of Test Sparse Matrix = (2649430, 17771)
0:00:00.979906
```

In [ ]:
```
#If you can see above that the shape of both train and test sparse matrices are same, furthermore, how come this shape of
#matrix has arrived:
#Shape of sparse matrix depends on highest value of User ID and highest value of Movie ID.
#Now the user whose user ID is highest is present in both train data and test data. Similarly, the movie whose movie ID i
#highest is present in both train data and test data. Hence, shape of both train and test sparse matrices are same.
```

In [26]:
```
rows,cols = TrainUISparseData.shape
presentElements = TrainUISparseData.count_nonzero()

print("Sparsity Of Train matrix : {}% ".format((1-(presentElements/(rows*cols)))*100))
```

```
Sparsity Of Train matrix : 99.90854433187319%
```

In [27]:
```
rows,cols = TestUISparseData.shape
presentElements = TestUISparseData.count_nonzero()

print("Sparsity Of Test matrix : {}% ".format((1-(presentElements/(rows*cols)))*100))
```

```
Sparsity Of Test matrix : 99.97713608243731%
```

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

In [5]:
```
def getAverageRatings(sparseMatrix, if_user):
    ax = 1 if if_user else 0
    #axis = 1 means rows and axis = 0 means columns
    sumOfRatings = sparseMatrix.sum(axis = ax).A1  #this will give an array of sum of all the ratings of user if axis = 1
    #sum of all the ratings of movies if axis = 0
    noOfRatings = (sparseMatrix!=0).sum(axis = ax).A1  #this will give a boolean True or False array, and True means 1 an
    #means 0, and further we are summing it to get the count of all the non-zero cells means length of non-zero cells
    rows, cols = sparseMatrix.shape
    averageRatings = {i: sumOfRatings[i]/noOfRatings[i] for i in range(rows if if_user else cols) if noOfRatings[i]!=0}
    return averageRatings
```

Global Average Rating

In [57]:
```
Global_Average_Rating = TrainUISparseData.sum()/TrainUISparseData.count_nonzero()
print("Global Average Rating {}".format(Global_Average_Rating))
```

```
Global Average Rating 3.5844935859517806
```

Average Rating Per User

In [58]:
```
AvgRatingUser = getAverageRatings(TrainUISparseData, True)
```

In [62]:
```
print("Average rating of user 25 = {}".format(AvgRatingUser[25]))
```

```
Average rating of user 25 = 3.0
```

Average Rating Per Movie

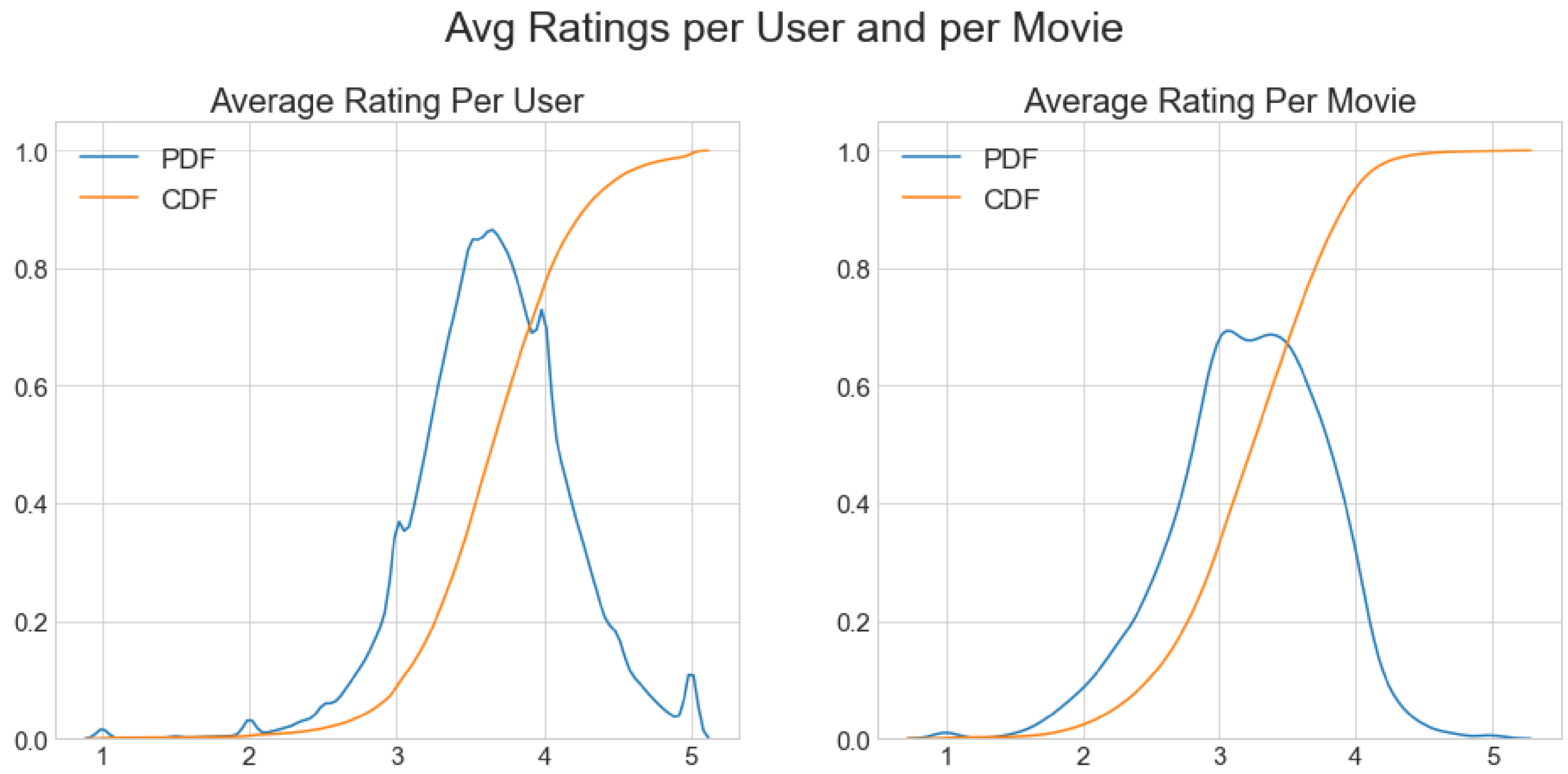
In [63]:
```
AvgRatingMovie = getAverageRatings(TrainUISparseData, False)
```

In [119]:
```
print("Average rating of movie 4500 = {}".format(AvgRatingMovie[4500]))
```

```
Average rating of movie 4500 = 3.28
```

PDF and CDF of Average Ratings of Users and Movies

In [108]:
```
fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (16, 7))
fig.suptitle('Avg Ratings per User and per Movie', fontsize=25)

user_average = [rats for rats in AvgRatingUser.values()]
sns.distplot(user_average, hist = False, ax = axes[0], label = "PDF")
sns.kdeplot(user_average, cumulative = True, ax = axes[0], label = "CDF")
axes[0].set_title("Average Rating Per User", fontsize=20)
axes[0].tick_params(labelsize = 15)
axes[0].legend(loc='upper left', fontsize = 17)

movie_average = [ratm for ratm in AvgRatingMovie.values()]
sns.distplot(movie_average, hist = False, ax = axes[1], label = "PDF")
sns.kdeplot(movie_average, cumulative = True, ax = axes[1], label = "CDF")
axes[1].set_title("Average Rating Per Movie", fontsize=20)
axes[1].tick_params(labelsize = 15)
axes[1].legend(loc='upper left', fontsize = 17)

plt.subplots_adjust(wspace=0.2, top=0.85)
plt.show()
```



Cold Start Problem

Cold Start Problem with Users

In [110]:
```
total_users = len(np.unique(Final_Data["CustID"]))
train_users = len(AvgRatingUser)
uncommonUsers = total_users - train_users

print("Total number of Users = {}".format(total_users))
print("Number of Users in train data= {}".format(train_users))
print("Number of Users not present in train data = {}({}%)".format(uncommonUsers, np.round((uncommonUsers/total_users)*10
```

```
Total number of Users = 478723
Number of Users in train data= 401901
Number of Users not present in train data = 76822(16.0%)
```

Cold Start Problem with Movies

In [112]:
```
total_movies = len(np.unique(Final_Data["MovieID"]))
train_movies = len(AvgRatingMovie)
uncommonMovies = total_movies - train_movies

print("Total number of Movies = {}".format(total_movies))
print("Number of Movies in train data= {}".format(train_movies))
print("Number of Movies not present in train data = {}({}%)".format(uncommonMovies, np.round((uncommonMovies/total_movies
```

```
Total number of Movies = 9114
Number of Movies in train data= 8931
Number of Movies not present in train data = 183(2.0%)
```

4. Computing Similarity Matrices

Computing User-User Similarity Matrix

Calculating User User Similarity_Matrix is **not very easy**(*unless you have huge Computing Power and lots of time*)

In [293]:
```
row_index, col_index = TrainUISparseData.nonzero()
rows = np.unique(row_index)
for i in rows[:100]:
    print(i)
```

```
6
7
10
25
33
42
59
79
83
87
94
97
131
134
142
149
158
168
169
178
183
188
189
192
195
199
201
242
247
248
261
265
266
267
268
283
```

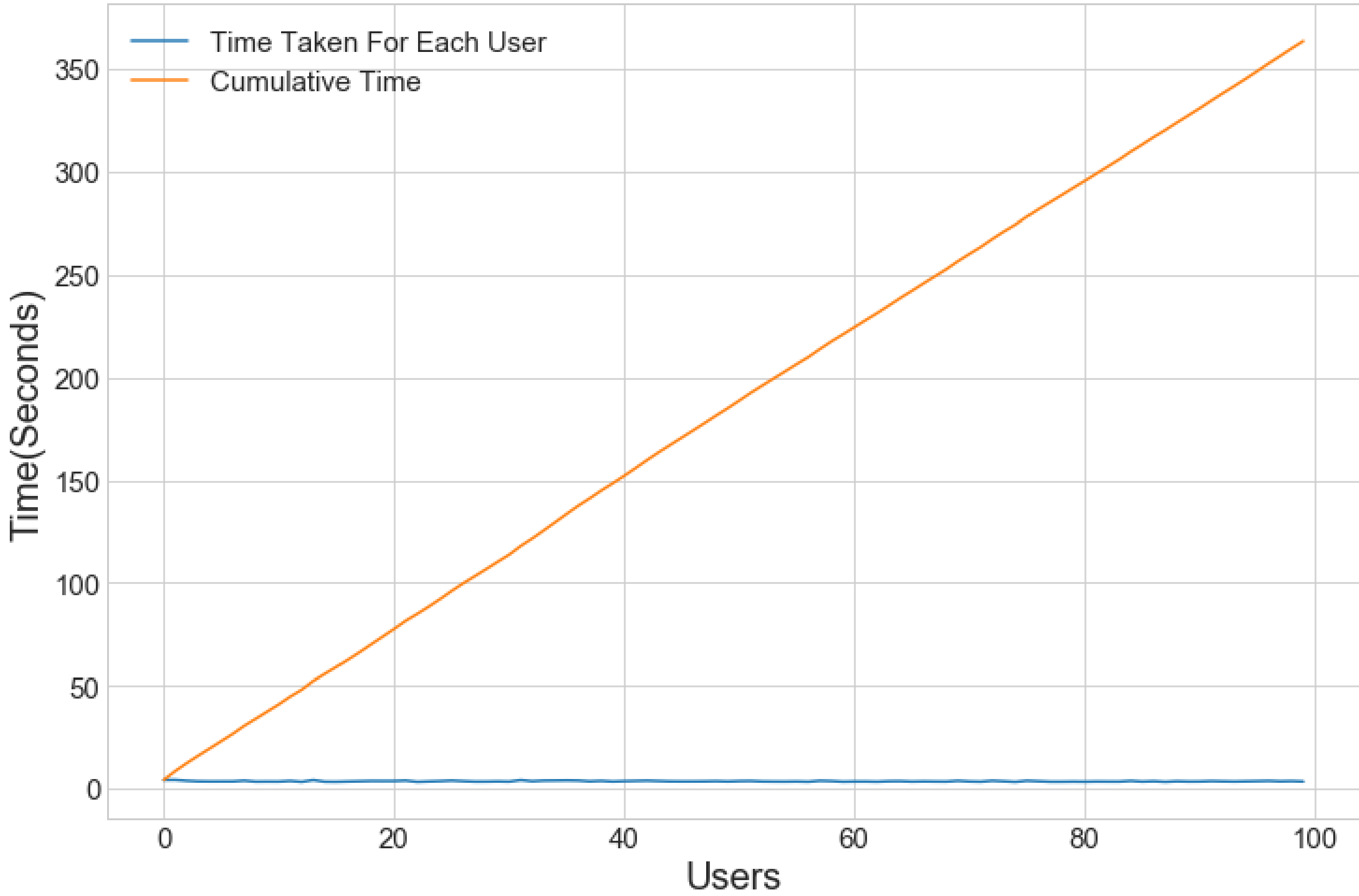
```
291
296
298
299
301
302
304
305
307
308
310
312
314
330
331
333
352
363
368
369
379
383
384
385
392
413
416
424
437
439
440
442
453
462
470
471
477
478
479
481
485
490
```

```
491
492
495
508
515
517
527
529
536
540
544
546
550
561
576
585
592
596
602
609
614
616
```

In [306]:
```
#Here, we are calculating user-user similarity matrix only for first 100 users in our sparse matrix. And we are calculati
#top 100 most similar users with them.
def getUser_UserSimilarity(sparseMatrix, top = 100):
    startTimestamp20 = datetime.now()

    row_index, col_index = sparseMatrix.nonzero()  #this will give indices of rows in "row_index" and indices of columns
    #"col_index" where there is a non-zero value exist.
    rows = np.unique(row_index)
    similarMatrix = np.zeros(61700).reshape(617,100)    # 617*100 = 61700. As we are building similarity matrix only
    #for top 100 most similar users.
    timeTaken = []
    howManyDone = 0
    for row in rows[:top]:
        howManyDone += 1
        startTimestamp = datetime.now().timestamp()  #it will give seconds elapsed
        sim = cosine_similarity(sparseMatrix.getrow(row), sparseMatrix).ravel()
        top100_similar_indices = sim.argsort()[-top:]
        top100_similar = sim[top100_similar_indices]
        similarMatrix[row] = top100_similar
        timeforOne = datetime.now().timestamp() - startTimestamp
        timeTaken.append(timeforOne)
        if howManyDone % 20 == 0:
            print("Time elapsed for {} users = {}sec".format(howManyDone, (datetime.now() - startTimestamp20)))
    print("Average Time taken to compute similarity matrix for 1 user = "+str(sum(timeTaken)/len(timeTaken))+"seconds")

    fig = plt.figure(figsize = (12,8))
    plt.plot(timeTaken, label = 'Time Taken For Each User')
    plt.plot(np.cumsum(timeTaken), label='Cumulative Time')
    plt.legend(loc='upper left', fontsize = 15)
    plt.xlabel('Users', fontsize = 20)
    plt.ylabel('Time(Seconds)', fontsize = 20)
    plt.tick_params(labelsize = 15)
    plt.show()

    return similarMatrix
```

In [307]:
```
simMatrix = getUser_UserSimilarity(TrainUISparseData, 100)
```

```
Time elapsed for 20 users = 0:01:15.836766sec
Time elapsed for 40 users = 0:02:30.449323sec
Time elapsed for 60 users = 0:03:42.918229sec
Time elapsed for 80 users = 0:04:54.074407sec
Time elapsed for 100 users = 0:06:05.538711sec
Average Time taken to compute similarity matrix for 1 user = 3.635262870788574seconds
```



We have **401901 Users** in our training data.

Average time taken to compute similarity matrix for one user is **3.635 sec.**

For 401901 users:

*401901*3.635 == 1460910.135sec == 405.808hours == 17Days*

Computation of user-user similarity matrix is impossible if computational power is limited. On the other hand, if we try to reduce the dimension say by truncated SVD then it would take even more time because truncated SVD creates dense matrix and amount of multiplication for creation of user-user similarity matrix would increase dramatically.

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

We maintain a binary Vector for users, which tells us whether we already computed similarity for this user or not..

OR

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>

Computing Movie-Movie Similarity Matrix

In [6]:
```
start = datetime.now()

if not os.path.isfile("../Data/m_m_similarity.npz"):
    print("Movie-Movie Similarity file does not exist in your disk. Creating Movie-Movie Similarity Matrix...")

    m_m_similarity = cosine_similarity(TrainUISparseData.T, dense_output = False)
    print("Done")
    print("Dimension of Matrix = {}".format(m_m_similarity.shape))
    print("Storing the Movie Similarity matrix on disk for further usage")
    sparse.save_npz("../Data/m_m_similarity.npz", m_m_similarity)
else:
    print("File exists in the disk. Loading the file...")
    m_m_similarity = sparse.load_npz("../Data/m_m_similarity.npz")
    print("Dimension of Matrix = {}".format(m_m_similarity.shape))

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

```
File exists in the disk. Loading the file...
Dimension of Matrix = (17771, 17771)
0:00:09.533895
```

Does Movie-Movie Similarity Works?

Let's pick random movie and check it's top 10 most similar movies.

In [46]:
```
movie_ids = np.unique(m_m_similarity.nonzero())
```

In [52]:
```
similar_movies_dict = dict()
for movie in movie_ids:
    smlr = np.argsort(-m_m_similarity[movie].toarray().ravel())[1:100]
    similar_movies_dict[movie] = smlr
```

In [54]:
```
movie_titles_df = pd.read_csv("../Data/movie_titles.csv",sep = ",", header = None, names=['MovieID', 'Year_of_Release', '
```

In [188]:
```
movie_titles_df.head()
```

Out[188]:

	Year_of_Release	Movie_Title
MovieID		
1	2003.0	Dinosaur Planet
2	2004.0	Isle of Man TT 2004 Review
3	1997.0	Character
4	1994.0	Paula Abdul's Get Up & Dance
5	2004.0	The Rise and Fall of ECW

Similar Movies to: **Godzilla's Revenge**

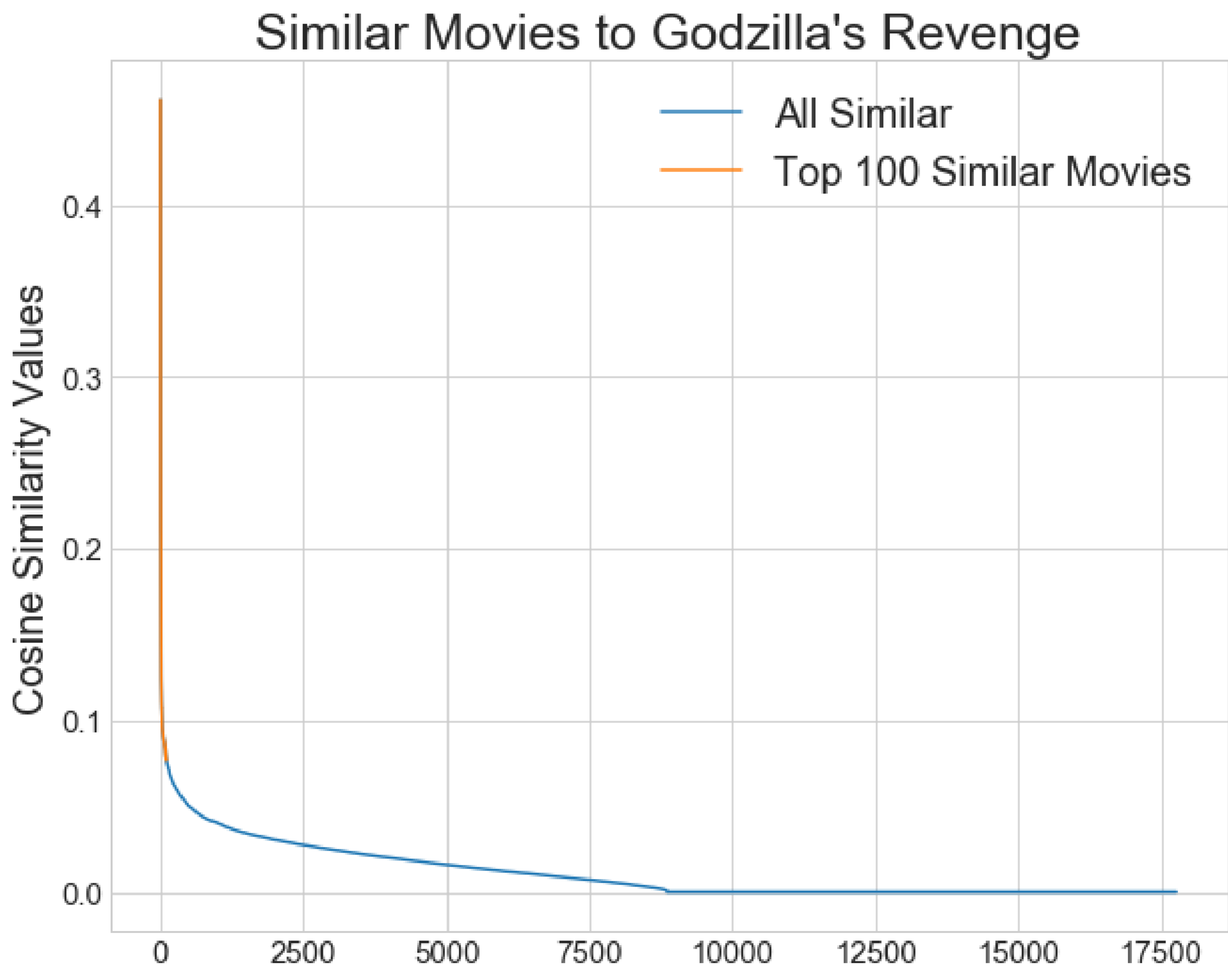
In [89]:
```
movieID_GR = 17765

print("Name of the movie -------> "+str(movie_titles_df.loc[movieID_GR][1]))

print("Number of ratings by users for movie {} is {}".format(movie_titles_df.loc[movieID_GR][1], TrainUISparseData[:,movi

print("Number of similar movies to {} is {}".format(movie_titles_df.loc[movieID_GR][1], m_m_similarity[movieID_GR].count_
```

```
Name of the movie -------> Godzilla's Revenge
Number of ratings by users for movie Godzilla's Revenge is 285
Number of similar movies to Godzilla's Revenge is 8863
```

In [ ]:
```
# Meaning of "[:,17765]" means get all the values of column "17765".
# "getnnz()" give count of explicitly-stored values (nonzeros).
```

In [111]:
```
all_similar = sorted(m_m_similarity[movieID_GR].toarray().ravel(), reverse = True)[1:]

similar_100 = all_similar[:101]
```

In [119]:
```
plt.figure(figsize = (10, 8))
plt.plot(all_similar, label = "All Similar")
plt.plot(similar_100, label = "Top 100 Similar Movies")
plt.title("Similar Movies to Godzilla's Revenge", fontsize = 25)
plt.ylabel("Cosine Similarity Values", fontsize = 20)
plt.tick_params(labelsize = 15)
plt.legend(fontsize = 20)
plt.show()
```



Top 10 Similar Movies to: **Godzilla's Revenge**

In [190]:
```
movie_titles_df.loc[similar_movies_dict[movieID_GR][:10]]
```

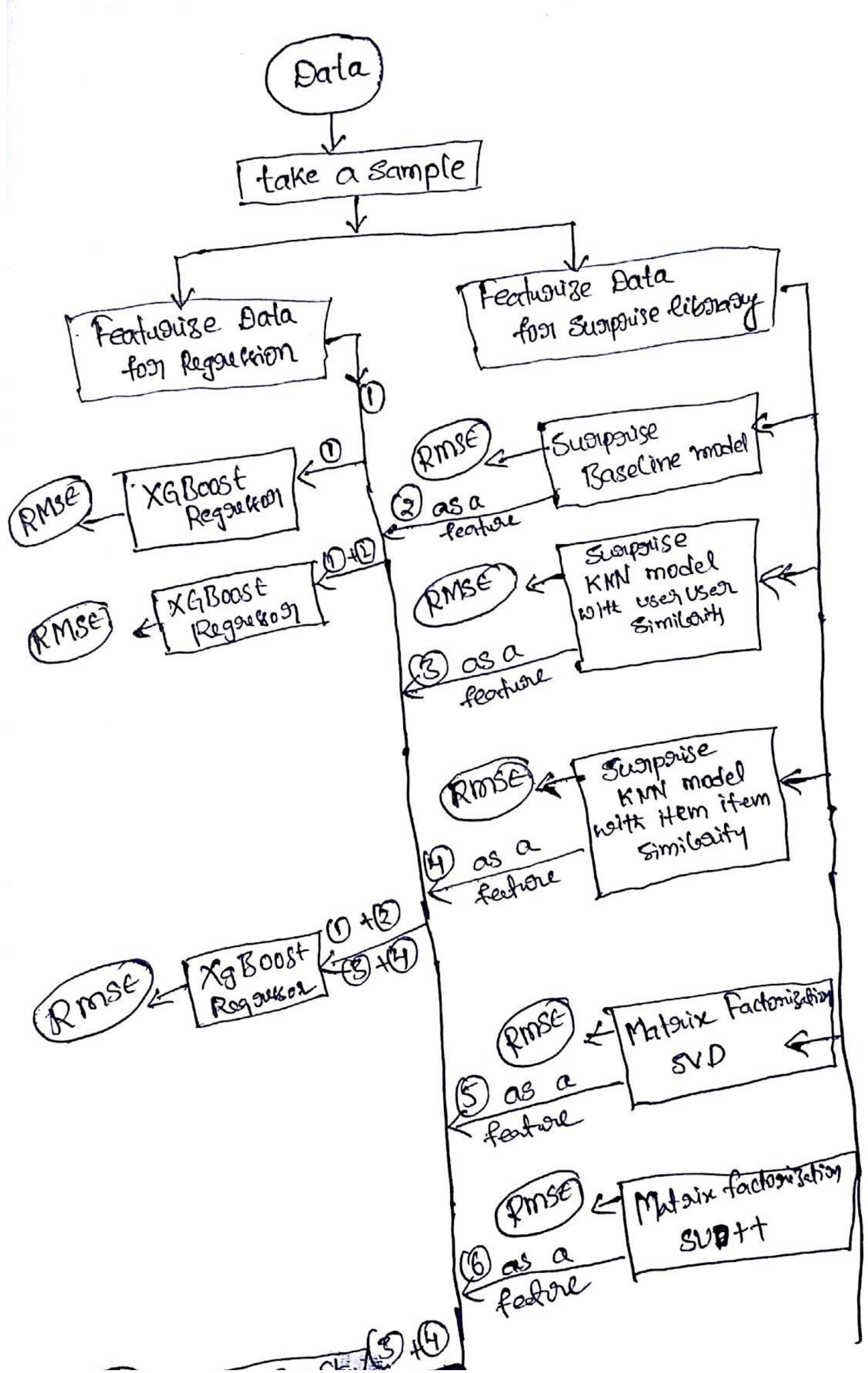
Out[190]:

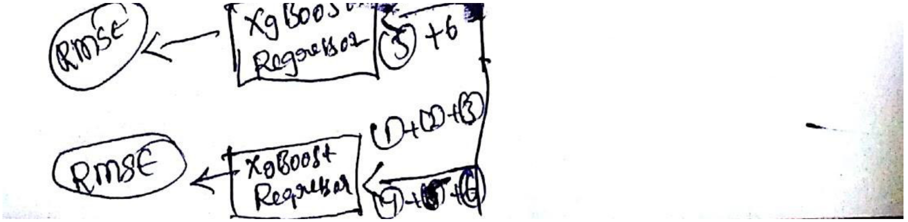
MovieID	Year_of_Release	Movie_Title
15810	1964.0	Godzilla vs. Mothra
5907	1956.0	Godzilla: King of the Monsters
14623	1971.0	Godzilla vs. Hedorah
8233	1968.0	Destroy All Monsters
17746	1991.0	Godzilla & Mothra: Battle for Earth / Vs. King...
15123	1995.0	Godzilla vs. Destroyah / Godzilla vs. Space Go...
8601	1997.0	Rebirth of Mothra 1 & 2: Double Feature
8656	1993.0	Godzilla vs. Mechagodzilla II
7140	2003.0	Godzilla: Tokyo S.O.S.
7228	1996.0	Gamera 2: Attack of Legion

It seems that Movie-Movie similarity is working perfectly.

5. Machine Learning Models

Strategy is as follows:


Data
take a Sample
Featurize Data for Regression
Featurize Data for Surprise library
1
XGBoost Regressor
RMSE
1+2
XGBoost Regressor
RMSE
Surprise Baseline model
RMSE
2 as a feature
Surprise KNN model with user user similarity
RMSE
3 as a feature
Surprise KNN model with item item similarity
RMSE
4 as a feature
1+2
3+4
XgBoost Regressor
RMSE
Matrix Factorization SVD
RMSE
5 as a feature
Matrix factorization SVD++
RMSE
6 as a feature
3+4



In [26]:
```
def get_sample_sparse_matrix(sparseMatrix, n_users, n_movies):
    startTime = datetime.now()
    users, movies, ratings = sparse.find(sparseMatrix)
    uniq_users = np.unique(users)
    uniq_movies = np.unique(movies)
    np.random.seed(15)   #this will give same random number everytime, without replacement
    userS = np.random.choice(uniq_users, n_users, replace = False)
    movieS = np.random.choice(uniq_movies, n_movies, replace = False)
    mask = np.logical_and(np.isin(users, userS), np.isin(movies, movieS))
    sparse_sample = sparse.csr_matrix((ratings[mask], (users[mask], movies[mask])),
                                                 shape = (max(userS)+1, max(movieS)+1))
    print("Sparse Matrix creation done. Saving it for later use.")
    sparse.save_npz(path, sparse_sample)
    print("Done")
    print("Shape of Sparse Sampled Matrix = "+str(sparse_sample.shape))

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

Creating Sample Sparse Matrix for Train Data

In [7]:
```
path = "../Data/TrainUISparseData_Sample.npz"
if not os.path.isfile(path):
    print("Sample sparse matrix is not present in the disk. We are creating it...")
    train_sample_sparse = get_sample_sparse_matrix(TrainUISparseData, 4000, 400)
else:
    print("File is already present in the disk. Loading the file...")
    train_sample_sparse = sparse.load_npz(path)
    print("File loading done.")
    print("Shape of Train Sample Sparse Matrix = "+str(train_sample_sparse.shape))
```

```
File is already present in the disk. Loading the file...
File loading done.
Shape of Train Sample Sparse Matrix = (2649117, 17764)
```

Creating Sample Sparse Matrix for Test Data

In [8]:
```
path = "../Data/TestUISparseData_Sample.npz"
if not os.path.isfile(path):
    print("Sample sparse matrix is not present in the disk. We are creating it...")
    test_sample_sparse = get_sample_sparse_matrix(TestUISparseData, 2000, 200)
else:
    print("File is already present in the disk. Loading the file...")
    test_sample_sparse = sparse.load_npz(path)
    print("File loading done.")
    print("Shape of Test Sample Sparse Matrix = "+str(test_sample_sparse.shape))
```

```
File is already present in the disk. Loading the file...
File loading done.
Shape of Test Sample Sparse Matrix = (2647588, 17689)
```

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

In [56]:
```
print("Global average of all movies ratings in Train Sample Sparse is {}".format(np.round((train_sample_sparse.sum()/trai
```

```
Global average of all movies ratings in Train Sample Sparse is 3.58
```

Finding Average of all movie ratings

In [121]:
```
globalAvgMovies = getAverageRatings(train_sample_sparse, False)
print("Average move rating for movie 14890 is {}".format(globalAvgMovies[14890]))
```

```
Average move rating for movie 14890 is 3.2870967741935484
```

Finding Average rating per User

In [122]:
```
globalAvgUsers = getAverageRatings(train_sample_sparse, True)
print("Average user rating for user 16879 is {}".format(globalAvgMovies[16879]))
```

```
Average user rating for user 16879 is 3.738095238095238
```

Featurizing data

In [10]:
```
print("No of ratings in Our Sampled train matrix is : {}".format(train_sample_sparse.count_nonzero()))
print("No of ratings in Our Sampled test matrix is : {}".format(test_sample_sparse.count_nonzero()))
```

```
No of ratings in Our Sampled train matrix is : 19214
No of ratings in Our Sampled test matrix is : 1150
```

Featurizing data for regression problem

Featurizing Train Data

In [210]:
```
sample_train_users, sample_train_movies, sample_train_ratings = sparse.find(train_sample_sparse)
```

In [280]:
```
if os.path.isfile("../Data/Train_Regression.csv"):
    print("File is already present in your disk. You do not have to prepare it again.")
else:
    startTime = datetime.now()
    print("Preparing Train csv file for {} rows".format(len(sample_train_ratings)))
    with open("../Data/Train_Regression.csv", mode = "w") as data:
        count = 0
        for user, movie, rating in zip(sample_train_users, sample_train_movies, sample_train_ratings):
            row = list()
            row.append(user)  #appending user ID
            row.append(movie) #appending movie ID
            row.append(train_sample_sparse.sum()/train_sample_sparse.count_nonzero()) #appending global average rating

#-----------------------------------Ratings given to "movie" by top 5 similar users with "user"--------------------#
            similar_users = cosine_similarity(train_sample_sparse[user], train_sample_sparse).ravel()
            similar_users_indices = np.argsort(-similar_users)[1:]
            similar_users_ratings = train_sample_sparse[similar_users_indices, movie].toarray().ravel()
            top_similar_user_ratings = list(similar_users_ratings[similar_users_ratings != 0][:5])
            top_similar_user_ratings.extend([globalAvgMovies[movie]]*(5-len(top_similar_user_ratings)))
            #above line means that if top 5 ratings are not available then rest of the ratings will be filled by "movie"
            #rating. Let say only 3 out of 5 ratings are available then rest 2 will be "movie" average rating.
            row.extend(top_similar_user_ratings)

 #----------------------------------Ratings given by "user" to top 5 similar movies with "movie"------------------#
            similar_movies = cosine_similarity(train_sample_sparse[:,movie].T, train_sample_sparse.T).ravel()
            similar_movies_indices = np.argsort(-similar_movies)[1:]
            similar_movies_ratings = train_sample_sparse[user, similar_movies_indices].toarray().ravel()
            top_similar_movie_ratings = list(similar_movies_ratings[similar_movies_ratings != 0][:5])
            top_similar_movie_ratings.extend([globalAvgUsers[user]]*(5-len(top_similar_movie_ratings)))
            #above line means that if top 5 ratings are not available then rest of the ratings will be filled by "user" a
            #rating. Let say only 3 out of 5 ratings are available then rest 2 will be "user" average rating.
            row.extend(top_similar_movie_ratings)

 #----------------------------------Appending "user" average, "movie" average & rating of "user""movie"-----------#
            row.append(globalAvgUsers[user])
            row.append(globalAvgMovies[movie])
            row.append(rating)

#-----------------------------------Converting rows and appending them as comma separated values to csv file------#
            data.write(",".join(map(str, row)))
            data.write("\n")
```

```
            count += 1
            if count % 2000 == 0:
                print("Done for {}. Time elapsed: {}".format(count, (datetime.now() - startTime)))

    print("Total Time for {} rows = {}".format(len(sample_train_ratings), (datetime.now() - startTime)))
```

```
Preparing Train csv file for 19214 rows
Done for 2000. Time elapsed: 0:14:17.429226
Done for 4000. Time elapsed: 0:25:51.882984
Done for 6000. Time elapsed: 0:37:21.039996
Done for 8000. Time elapsed: 0:49:03.121577
Done for 10000. Time elapsed: 1:00:25.030957
Done for 12000. Time elapsed: 1:11:50.660054
Done for 14000. Time elapsed: 1:24:15.366893
Done for 16000. Time elapsed: 1:36:31.156832
Done for 18000. Time elapsed: 1:48:18.891065
Total Time for 19214 rows = 1:55:33.782934
```

In [2]:

```
Train_Reg = pd.read_csv("../Data/Train_Regression.csv", names = ["User_ID", "Movie_ID", "Global_Average", "SUR1", "SUR2",
Train_Reg.head()
```

Out[2]:

	User_ID	Movie_ID	Global_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	User_Average	Movie_Average	Ratir
0	180921	4512	3.582804	3.0	2.0	1.0	2.0	1.0	4.0	3.0	4.0	2.0	2.0	2.900000	2.5	
1	210185	4512	3.582804	2.0	1.0	3.0	3.0	4.0	3.0	3.0	3.0	4.0	4.0	3.388889	2.5	
2	218038	4512	3.582804	2.0	3.0	3.0	2.0	4.0	4.0	4.0	4.0	3.0	5.0	4.250000	2.5	
3	221936	4512	3.582804	4.0	2.0	2.0	1.0	2.0	3.0	4.0	4.0	5.0	3.0	3.458333	2.5	
4	370736	4512	3.582804	2.0	4.0	1.0	2.0	2.0	4.0	4.0	4.0	4.0	5.0	4.038462	2.5	

In [3]:

```
print("Number of nan Values = "+str(Train_Reg.isnull().sum().sum()))
```

```
Number of nan Values = 0
```

User_ID: ID of a this User

Movie_ID: ID of a this Movie

Global_Average: Global Average Rating

Ratings given to this Movie by top 5 similar users with this User: (SUR1, SUR2, SUR3, SUR4, SUR5)

Ratings given by this User to top 5 similar movies with this Movie: (SMR1, SMR2, SMR3, SMR4, SMR5)

User_Average: Average Rating of this User

Movie_Average: Average Rating of this Movie

Rating: Rating given by this User to this Movie

In [4]:
```
print("Shape of Train DataFrame = {}".format(Train_Reg.shape))
```

```
Shape of Train DataFrame = (19214, 16)
```

Featurizing Test Data

In [274]:
```
sample_test_users, sample_test_movies, sample_test_ratings = sparse.find(test_sample_sparse)
```

In [275]:
```
if os.path.isfile("../Data/Test_Regression.csv"):
    print("File is already present in your disk. You do not have to prepare it again.")
else:
    startTime = datetime.now()
    print("Preparing Test csv file for {} rows".format(len(sample_test_ratings)))
    with open("../Data/Test_Regression.csv", mode = "w") as data:
        count = 0
        for user, movie, rating in zip(sample_test_users, sample_test_movies, sample_test_ratings):
            row = list()
            row.append(user)  #appending user ID
            row.append(movie) #appending movie ID
            row.append(train_sample_sparse.sum()/train_sample_sparse.count_nonzero()) #appending global average rating

#-----------------------------Ratings given to "movie" by top 5 similar users with "user"-------------------------#
            try:
                similar_users = cosine_similarity(train_sample_sparse[user], train_sample_sparse).ravel()
                similar_users_indices = np.argsort(-similar_users)[1:]
                similar_users_ratings = train_sample_sparse[similar_users_indices, movie].toarray().ravel()
                top_similar_user_ratings = list(similar_users_ratings[similar_users_ratings != 0][:5])
                top_similar_user_ratings.extend([globalAvgMovies[movie]]*(5-len(top_similar_user_ratings)))
                #above line means that if top 5 ratings are not available then rest of the ratings will be filled by "mov
                #average rating. Let say only 3 out of 5 ratings are available then rest 2 will be "movie" average rating
                row.extend(top_similar_user_ratings)
            #########Cold Start Problem, for a new user or a new movie#########
            except(IndexError, KeyError):
                global_average_train_rating = [train_sample_sparse.sum()/train_sample_sparse.count_nonzero()]*5
                row.extend(global_average_train_rating)
            except:
                raise

 #-----------------------------Ratings given by "user" to top 5 similar movies with "movie"-----------------------#
            try:
                similar_movies = cosine_similarity(train_sample_sparse[:,movie].T, train_sample_sparse.T).ravel()
                similar_movies_indices = np.argsort(-similar_movies)[1:]
                similar_movies_ratings = train_sample_sparse[user, similar_movies_indices].toarray().ravel()
                top_similar_movie_ratings = list(similar_movies_ratings[similar_movies_ratings != 0][:5])
                top_similar_movie_ratings.extend([globalAvgUsers[user]]*(5-len(top_similar_movie_ratings)))
                #above line means that if top 5 ratings are not available then rest of the ratings will be filled by "use
                #average rating. Let say only 3 out of 5 ratings are available then rest 2 will be "user" average rating.
                row.extend(top_similar_movie_ratings)
            #########Cold Start Problem, for a new user or a new movie#########
```

```
            except(IndexError, KeyError):
                global_average_train_rating = [train_sample_sparse.sum()/train_sample_sparse.count_nonzero()]*5
                row.extend(global_average_train_rating)
            except:
                raise

 #-----------------------------Appending "user" average, "movie" average & rating of "user""movie"----------------#
            try:
                row.append(globalAvgUsers[user])
            except (KeyError):
                global_average_train_rating = train_sample_sparse.sum()/train_sample_sparse.count_nonzero()
                row.append(global_average_train_rating)
            except:
                raise

            try:
                row.append(globalAvgMovies[movie])
            except(KeyError):
                global_average_train_rating = train_sample_sparse.sum()/train_sample_sparse.count_nonzero()
                row.append(global_average_train_rating)
            except:
                raise

            row.append(rating)

#------------------------------Converting rows and appending them as comma separated values to csv file-----------#
            data.write(",".join(map(str, row)))
            data.write("\n")

            count += 1
            if count % 100 == 0:
                print("Done for {}. Time elapsed: {}".format(count, (datetime.now() - startTime)))

    print("Total Time for {} rows = {}".format(len(sample_test_ratings), (datetime.now() - startTime)))
```

```
Preparing Test csv file for 1150 rows
Done for 100. Time elapsed: 0:00:57.690535
Done for 200. Time elapsed: 0:01:55.658291
Done for 300. Time elapsed: 0:02:51.644355
Done for 400. Time elapsed: 0:03:48.542774
Done for 500. Time elapsed: 0:04:46.203274
Done for 600. Time elapsed: 0:05:43.748850
```

```
Done for 700. Time elapsed: 0:06:40.060096
Done for 800. Time elapsed: 0:07:36.876978
Done for 900. Time elapsed: 0:08:35.474421
Done for 1000. Time elapsed: 0:09:35.487426
Done for 1100. Time elapsed: 0:10:33.057698
Total Time for 1150 rows = 0:11:01.636286
```

In [5]:

```
Test_Reg = pd.read_csv("../Data/Test_Regression.csv", names = ["User_ID", "Movie_ID", "Global_Average", "SUR1", "SUR2", "
Test_Reg.head()
```

Out[5]:

	User_ID	Movie_ID	Global_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	User_A
0	464626	4614	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
1	1815614	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
2	2298717	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
3	2532402	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
4	2027	4798	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5

In [6]:

```
print("Number of nan Values = "+str(Test_Reg.isnull().sum().sum()))
```

```
Number of nan Values = 0
```

User_ID: ID of a this User

Movie_ID: ID of a this Movie

Global_Average: Global Average Rating

Ratings given to this Movie by top 5 similar users with this User: (SUR1, SUR2, SUR3, SUR4, SUR5)

Ratings given by this User to top 5 similar movies with this Movie: (SMR1, SMR2, SMR3, SMR4, SMR5)

User_Average: Average Rating of this User

Movie_Average: Average Rating of this Movie

Rating: Rating given by this User to this Movie

In [7]:
```
print("Shape of Test DataFrame = {}".format(Test_Reg.shape))
```

```
Shape of Test DataFrame = (1150, 16)
```

Transforming Data for Surprise Models

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

In [8]:
```
Train_Reg[['User_ID', 'Movie_ID', 'Rating']].head(5)
```

Out[8]:

	User_ID	Movie_ID	Rating
0	180921	4512	1
1	210185	4512	2
2	218038	4512	4
3	221936	4512	3
4	370736	4512	4

In [9]:
```
reader = Reader(rating_scale=(1, 5))

data = Dataset.load_from_df(Train_Reg[['User_ID', 'Movie_ID', 'Rating']], reader)

trainset = data.build_full_trainset()
```

Transforming Test Data

- For test data we just have to define a tuple (user, item, rating).
- You can check out this link: https://github.com/NicolasHug/Surprise/commit/86cf44529ca0bbb97759b81d1716ff547b950812 (https://github.com/NicolasHug/Surprise/commit/86cf44529ca0bbb97759b81d1716ff547b950812)
- Above link is a github of surprise library. Check methods "def all_ratings(self)" and "def build_testset(self)" from line 177 to 201(If they modify the file then line number may differ, but you can always check aforementioned two methods).
- "def build_testset(self)" method returns a list of tuples of (user, item, rating).

In [10]:
```
testset = list(zip(Test_Reg["User_ID"].values, Test_Reg["Movie_ID"].values, Test_Reg["Rating"].values))
```

In [11]:
```
testset[:5]
```

Out[11]:
```
[(464626, 4614, 3),
 (1815614, 4627, 3),
 (2298717, 4627, 5),
 (2532402, 4627, 4),
 (2027, 4798, 5)]
```

Applying Machine Learning Models

We have two Error Metrics.

-> RMSE: Root Mean Square Error: RMSE is the error of each point which is squared. Then mean is calculated. Finally root of that mean is taken as final value.

-> MAPE: Mean Absolute Percentage Error: The mean absolute percentage error (MAPE), also known as mean absolute percentage deviation (MAPD), is a measure of prediction accuracy of a forecasting method.

$$\mathrm{M} = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right|,$$

where At is the actual value and Ft is the forecast value. The difference between At and Ft is divided by the actual value At again. The absolute value in this calculation is summed for every forecasted point in time and divided by the number of fitted points n. Multiplying by 100% makes it a percentage error.

We can also use other regression models. But we are using exclusively XGBoost as it is typically fairly powerful in practice.

In [12]:
```
error_table = pd.DataFrame(columns = ["Model", "Train RMSE", "Train MAPE", "Test RMSE", "Test MAPE"])
model_train_evaluation = dict()
model_test_evaluation = dict()
```

In [13]:
```
def make_table(model_name, rmse_train, mape_train, rmse_test, mape_test):
    global error_table
    #All variable assignments in a function store the value in the local symbol table; whereas variable references first
    #in the local symbol table, then in the global symbol table, and then in the table of built-in names. Thus, global va
    #cannot be directly assigned a value within a function (unless named in a global statement),
    #although they may be referenced.
    error_table = error_table.append(pd.DataFrame([[model_name, rmse_train, mape_train, rmse_test, mape_test]], columns =
    error_table.reset_index(drop = True, inplace = True)
```

Utility Functions for Regression Models

In [14]:
```
def error_metrics(y_true, y_pred):
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    mape = np.mean(abs((y_true - y_pred)/y_true))*100
    return rmse, mape
```

In [15]:
```
def train_test_xgboost(x_train, x_test, y_train, y_test, model_name):
    startTime = datetime.now()
    train_result = dict()
    test_result = dict()

    clf = xgb.XGBRegressor(n_estimators = 100, silent = False, n_jobs  = 10)
    clf.fit(x_train, y_train)

    print("-"*50)
    print("TRAIN DATA")
    y_pred_train = clf.predict(x_train)
    rmse_train, mape_train = error_metrics(y_train, y_pred_train)
    print("RMSE = {}".format(rmse_train))
    print("MAPE = {}".format(mape_train))
    print("-"*50)
    train_result = {"RMSE": rmse_train, "MAPE": mape_train, "Prediction": y_pred_train}

    print("TEST DATA")
    y_pred_test = clf.predict(x_test)
    rmse_test, mape_test = error_metrics(y_test, y_pred_test)
    print("RMSE = {}".format(rmse_test))
    print("MAPE = {}".format(mape_test))
    print("-"*50)
    test_result = {"RMSE": rmse_test, "MAPE": mape_test, "Prediction": y_pred_test}

    print("Time Taken = "+str(datetime.now() - startTime))

    plot_importance(xgb, clf)

    make_table(model_name, rmse_train, mape_train, rmse_test, mape_test)

    return train_result, test_result
```

In [16]:
```
def plot_importance(model, clf):
    fig = plt.figure(figsize = (8, 6))
    ax = fig.add_axes([0,0,1,1])
    model.plot_importance(clf, ax = ax, height = 0.3)
    plt.xlabel("F Score", fontsize = 20)
    plt.ylabel("Features", fontsize = 20)
    plt.title("Feature Importance", fontsize = 20)
    plt.tick_params(labelsize = 15)

    plt.show()
```

Utility Functions for Surprise Models

In [17]:
```
def get_ratings(predictions):
    actual = np.array([pred.r_ui for pred in predictions])
    predicted = np.array([pred.est for pred in predictions])
    return actual, predicted
#in surprise prediction of every data point is returned as dictionary like this:
#"user: 196        item: 302        r_ui = 4.00   est = 4.06   {'actual_k': 40, 'was_impossible': False}"
#In this dictionary, "r_ui" is a key for actual rating and "est" is a key for predicted rating
```

In [18]:
```
def get_error(predictions):
    actual, predicted = get_ratings(predictions)
    rmse = np.sqrt(mean_squared_error(actual, predicted))
    mape = np.mean(abs((actual - predicted)/actual))*100
    return rmse, mape
```

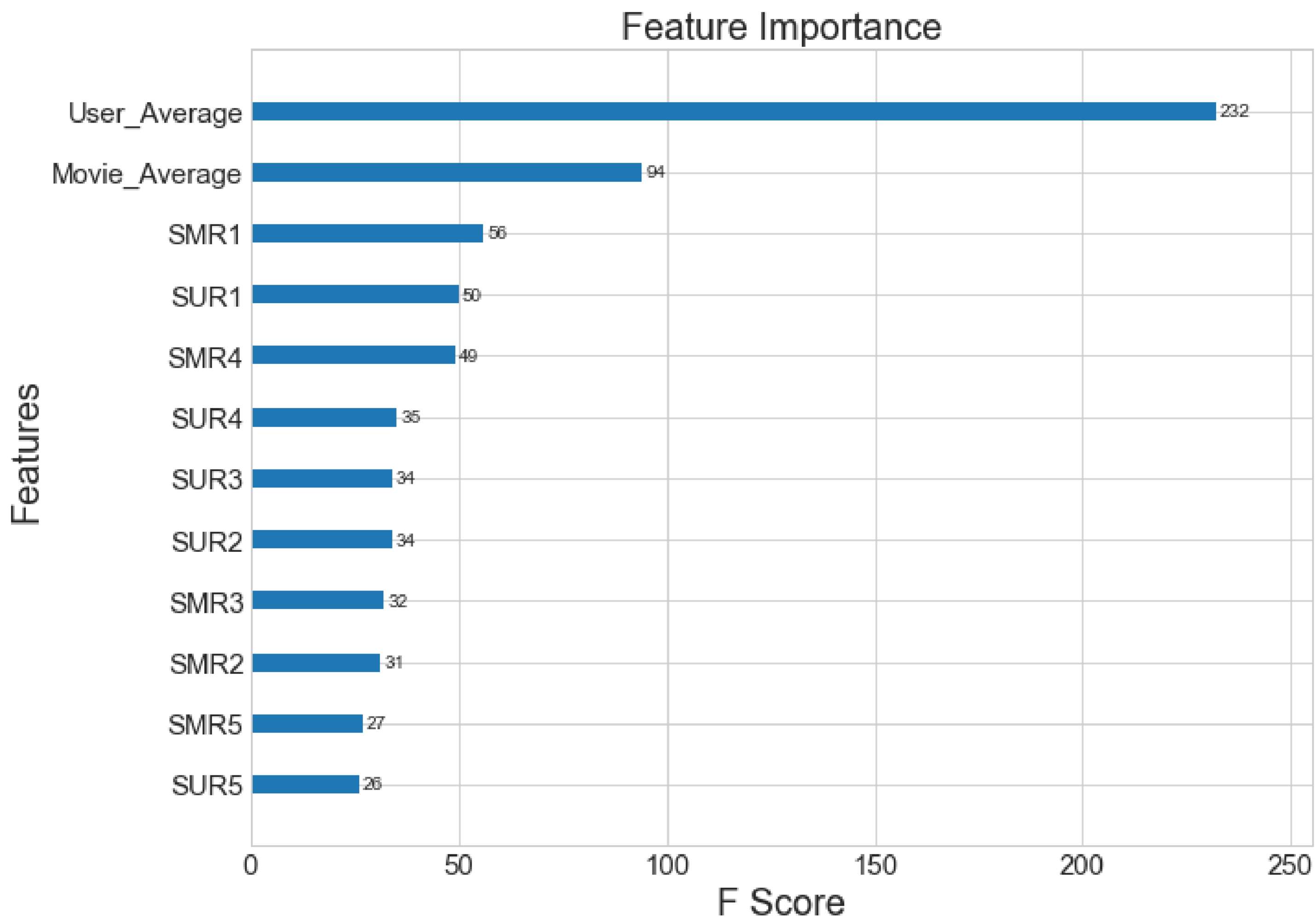
In [19]:
```
my_seed = 15
random.seed(my_seed)
np.random.seed(my_seed)

def run_surprise(algo, trainset, testset, model_name):
    startTime = datetime.now()

    train = dict()
    test = dict()

    algo.fit(trainset)
    #You can check out above function at "https://surprise.readthedocs.io/en/stable/getting_started.html" in
    #"Train-test split and the fit() method" section

#-----------------Evaluating Train Data------------------#
    print("-"*50)
    print("TRAIN DATA")
    train_pred = algo.test(trainset.build_testset())
    #You can check out "algo.test()" function at "https://surprise.readthedocs.io/en/stable/getting_started.html" in
    #"Train-test split and the fit() method" section
    #You can check out "trainset.build_testset()" function at "https://surprise.readthedocs.io/en/stable/FAQ.html#can-i-u
    #"How to get accuracy measures on the training set" section
    train_actual, train_predicted = get_ratings(train_pred)
    train_rmse, train_mape = get_error(train_pred)
    print("RMSE = {}".format(train_rmse))
    print("MAPE = {}".format(train_mape))
    print("-"*50)
    train = {"RMSE": train_rmse, "MAPE": train_mape, "Prediction": train_predicted}

#-----------------Evaluating Test Data------------------#
    print("TEST DATA")
    test_pred = algo.test(testset)
    #You can check out "algo.test()" function at "https://surprise.readthedocs.io/en/stable/getting_started.html" in
    #"Train-test split and the fit() method" section
    test_actual, test_predicted = get_ratings(test_pred)
    test_rmse, test_mape = get_error(test_pred)
    print("RMSE = {}".format(test_rmse))
    print("MAPE = {}".format(test_mape))
    print("-"*50)
    test = {"RMSE": test_rmse, "MAPE": test_mape, "Prediction": test_predicted}
```

```
    print("Time Taken = "+str(datetime.now() - startTime))

    make_table(model_name, train_rmse, train_mape, test_rmse, test_mape)

    return train, test
```

1. XGBoost 13 Features

In [20]:
```
x_train = Train_Reg.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)

x_test = Test_Reg.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)

y_train = Train_Reg["Rating"]

y_test = Test_Reg["Rating"]

train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test, "XGBoost_13")

model_train_evaluation["XGBoost_13"] = train_result
model_test_evaluation["XGBoost_13"] = test_result
```

```
--------------------------------------------------
TRAIN DATA
RMSE = 0.8101861960249761
MAPE = 24.154755473136902
--------------------------------------------------
TEST DATA
RMSE = 1.068611182233979
MAPE = 33.35963301935049
--------------------------------------------------
Time Taken = 0:00:01.135756
```



2. Surprise BaselineOnly Model

Predicted Rating

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

- μ: Average Global Ratings in training data
- b_u: User-Bias
- b_i: Item-Bias

Optimization Function

$$\sum_{r_u i \in R_{Train}} \left(r_{ui} - (\mu + b_u + b_i)\right)^2 + \lambda \left(b_u^2 + b_i^2\right).\ [minimize\ b_u, b_i]$$

In [22]:
```
bsl_options = {"method":"sgd", "learning_rate":0.01, "n_epochs":25}

algo = BaselineOnly(bsl_options=bsl_options)
#You can check the docs of above used functions at:https://surprise.readthedocs.io/en/stable/prediction_algorithms.html#b
#at section "Baselines estimates configuration".

train_result, test_result = run_surprise(algo, trainset, testset, "BaselineOnly")

model_train_evaluation["BaselineOnly"] = train_result
model_test_evaluation["BaselineOnly"] = test_result
```

```
Estimating biases using sgd...
--------------------------------------------------
TRAIN DATA
RMSE = 0.8811426214928658
MAPE = 27.158727146074078
--------------------------------------------------
TEST DATA
RMSE = 1.0678388468431512
MAPE = 33.39729060309592
--------------------------------------------------
Time Taken = 0:00:00.516484
```

3. XGBoost 13 Features + Surprise BaselineOnly Model

Adding predicted ratings from Surprise BaselineOnly model to our Train and Test Dataframe

In [23]:
```
Train_Reg["BaselineOnly"] = model_train_evaluation["BaselineOnly"]["Prediction"]
```

In [24]:
```
Train_Reg.head()
```

Out[24]:

	User_ID	Movie_ID	Global_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	User_Average	Movie_Average	Ratir
0	180921	4512	3.582804	3.0	2.0	1.0	2.0	1.0	4.0	3.0	4.0	2.0	2.0	2.900000	2.5	
1	210185	4512	3.582804	2.0	1.0	3.0	3.0	4.0	3.0	3.0	3.0	4.0	4.0	3.388889	2.5	
2	218038	4512	3.582804	2.0	3.0	3.0	2.0	4.0	4.0	4.0	4.0	3.0	5.0	4.250000	2.5	
3	221936	4512	3.582804	4.0	2.0	2.0	1.0	2.0	3.0	4.0	4.0	5.0	3.0	3.458333	2.5	
4	370736	4512	3.582804	2.0	4.0	1.0	2.0	2.0	4.0	4.0	4.0	4.0	5.0	4.038462	2.5	

In [25]:
```
print("Number of nan values = "+str(Train_Reg.isnull().sum().sum()))
```

```
Number of nan values = 0
```

In [26]:
```
Test_Reg["BaselineOnly"] = model_test_evaluation["BaselineOnly"]["Prediction"]
Test_Reg.head()
```

Out[26]:

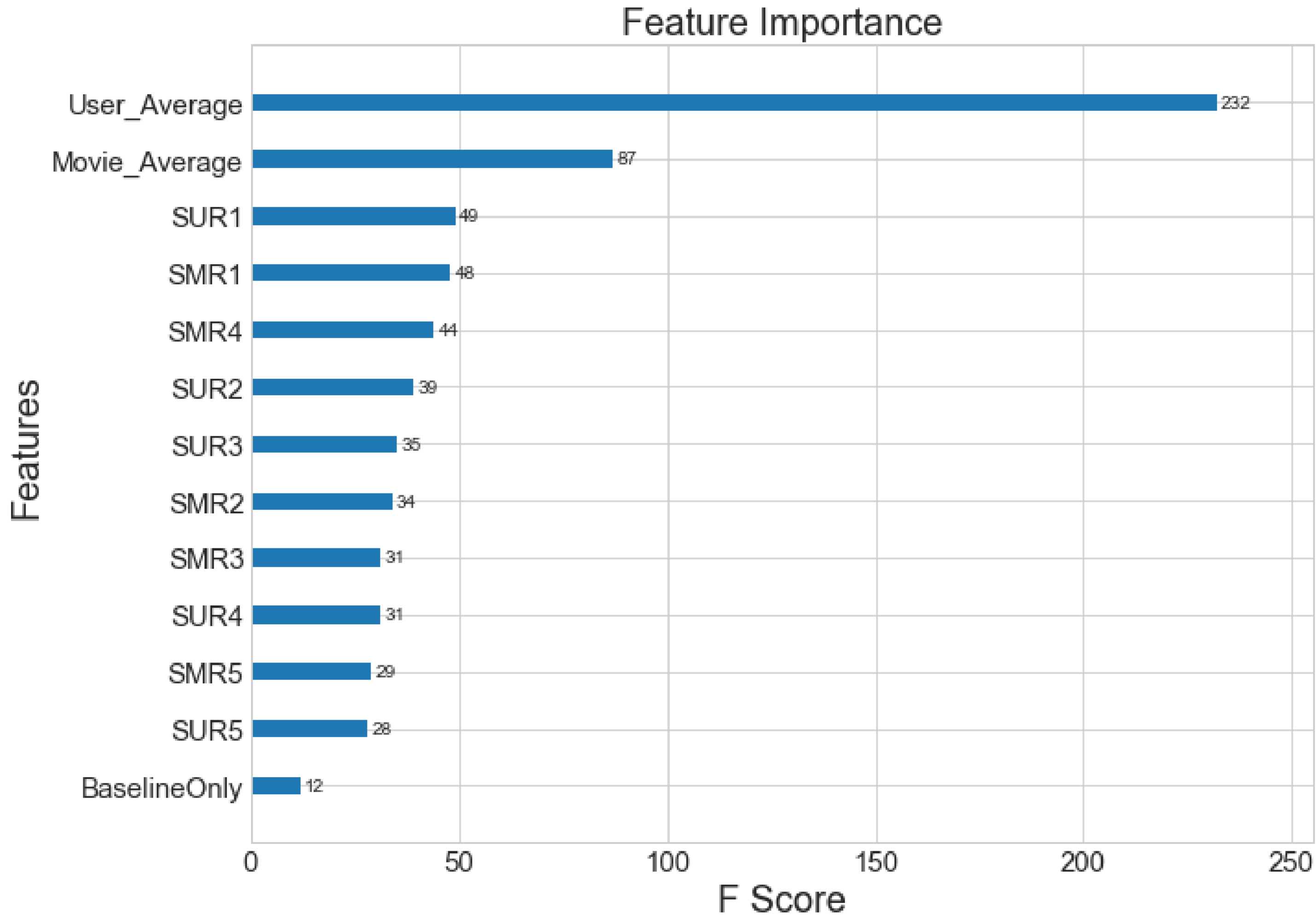
	User_ID	Movie_ID	Global_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	User_A
0	464626	4614	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
1	1815614	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
2	2298717	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
3	2532402	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
4	2027	4798	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5

In [27]:
```
print("Number of nan values = "+str(Test_Reg.isnull().sum().sum()))
```

```
Number of nan values = 0
```

In [28]:
```
x_train = Train_Reg.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)

x_test = Test_Reg.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)

y_train = Train_Reg["Rating"]

y_test = Test_Reg["Rating"]

train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test, "XGB_BSL")

model_train_evaluation["XGB_BSL"] = train_result
model_test_evaluation["XGB_BSL"] = test_result
```

```
--------------------------------------------------
TRAIN DATA
RMSE = 0.8098916039498553
MAPE = 24.152618646621704
--------------------------------------------------
TEST DATA
RMSE = 1.067675996745546
MAPE = 33.42641695858742
--------------------------------------------------
Time Taken = 0:00:01.004710
```



4. Surprise KNN-Baseline with User-User and Item-Item Similarity

Prediction $\hat{r}_{ui}$ in case of user-user similarity

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

- $\boldsymbol{b_{ui}}$ - Baseline prediction_ of (user,movie) rating which is "$b_{ui} = \mu + b_u + b_i$".
- $\boldsymbol{N_i^k(u)}$ - Set of **K similar** users (neighbours) of **user (u)** who rated **movie(i)**
- *sim (u, v)* - Similarity between users **u and v** who also rated movie 'i'. This is exactly same as our hand-crafted features 'SUR'- 'Similar User Rating'. Means here we have taken 'k' such similar users 'v' with user 'u' who also rated movie 'i'. r_{vi} is the rating which user 'v' gives on item 'i'. b_{vi} is the predicted baseline model rating of user 'v' on item 'i'.
  - 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)

Prediction $\hat{r}_{ui}$ in case of item-item similarity

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

- **Notation is same as of user-user similarity**

Documentation you can check at:

KNN BASELINE: https://surprise.readthedocs.io/en/stable/knn_inspired.html (https://surprise.readthedocs.io/en/stable/knn_inspired.html)

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

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

4.1 Surprise KNN-Baseline with User-User.

Cross- Validation

In [56]:
```
param_grid  = {'sim_options':{'name': ["pearson_baseline"], "user_based": [True], "min_support": [2], "shrinkage": [60, 8

gs = GridSearchCV(KNNBaseline, param_grid, measures=['rmse', 'mae'], cv=3)

gs.fit(data)

# best RMSE score
print(gs.best_score['rmse'])

# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
```

```
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
```

Applying KNNBaseline User-User with best parameters

In [61]:
```
sim_options = {'name':'pearson_baseline', 'user_based':True, 'min_support':2, 'shrinkage':gs.best_params['rmse']['sim_opt

bsl_options = {'method': 'sgd'}

algo = KNNBaseline(k = gs.best_params['rmse']['k'], sim_options = sim_options, bsl_options=bsl_options)

train_result, test_result = run_surprise(algo, trainset, testset, "KNNBaseline_User")

model_train_evaluation["KNNBaseline_User"] = train_result
model_test_evaluation["KNNBaseline_User"] = test_result
```

```
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
--------------------------------------------------
TRAIN DATA
RMSE = 0.3044975188091617
MAPE = 8.090955152033715
--------------------------------------------------
TEST DATA
RMSE = 1.067654798722828
MAPE = 33.39814334762251
--------------------------------------------------
Time Taken = 0:00:13.622646
```

4.2 Surprise KNN-Baseline with Item-Item.

Cross- Validation

In [62]:
```
param_grid  = {'sim_options':{'name': ["pearson_baseline"], "user_based": [False], "min_support": [2], "shrinkage": [60,

gs = GridSearchCV(KNNBaseline, param_grid, measures=['rmse', 'mae'], cv=3)

gs.fit(data)

# best RMSE score
print(gs.best_score['rmse'])

# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
```

```
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
```

Applying KNNBaseline Item-Item with best parameters

In [65]:
```
sim_options = {'name':'pearson_baseline', 'user_based':False, 'min_support':2, 'shrinkage':gs.best_params['rmse']['sim_op

bsl_options = {'method': 'sgd'}

algo = KNNBaseline(k = gs.best_params['rmse']['k'], sim_options = sim_options, bsl_options=bsl_options)

train_result, test_result = run_surprise(algo, trainset, testset, "KNNBaseline_Item")

model_train_evaluation["KNNBaseline_Item"] = train_result
model_test_evaluation["KNNBaseline_Item"] = test_result
```

```
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
--------------------------------------------------
TRAIN DATA
RMSE = 0.1818822561823507
MAPE = 4.2501507953116135
--------------------------------------------------
TEST DATA
RMSE = 1.067654798722828
MAPE = 33.39814334762251
--------------------------------------------------
Time Taken = 0:00:00.914647
```

5. XGBoost 13 Features + Surprise BaselineOnly + Surprise KNN Baseline

Adding predicted ratings from Surprise KNN Baseline model to our Train and Test Dataframe

In [68]:
```
Train_Reg["KNNBaseline_User"] = model_train_evaluation["KNNBaseline_User"]["Prediction"]
Train_Reg["KNNBaseline_Item"] = model_train_evaluation["KNNBaseline_Item"]["Prediction"]

Test_Reg["KNNBaseline_User"] = model_test_evaluation["KNNBaseline_User"]["Prediction"]
Test_Reg["KNNBaseline_Item"] = model_test_evaluation["KNNBaseline_Item"]["Prediction"]
```

In [69]:
```
Train_Reg.head()
```

Out[69]:

	User_ID	Movie_ID	Global_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	User_Average	Movie_Average	Ratir
0	180921	4512	3.582804	3.0	2.0	1.0	2.0	1.0	4.0	3.0	4.0	2.0	2.0	2.900000	2.5	
1	210185	4512	3.582804	2.0	1.0	3.0	3.0	4.0	3.0	3.0	3.0	4.0	4.0	3.388889	2.5	
2	218038	4512	3.582804	2.0	3.0	3.0	2.0	4.0	4.0	4.0	4.0	3.0	5.0	4.250000	2.5	
3	221936	4512	3.582804	4.0	2.0	2.0	1.0	2.0	3.0	4.0	4.0	5.0	3.0	3.458333	2.5	
4	370736	4512	3.582804	2.0	4.0	1.0	2.0	2.0	4.0	4.0	4.0	4.0	5.0	4.038462	2.5	

In [72]:
```
print("Number of nan values in Train Data "+str(Train_Reg.isnull().sum().sum()))
```

```
Number of nan values in Train Data 0
```

In [73]:
```
Test_Reg.head()
```

Out[73]:

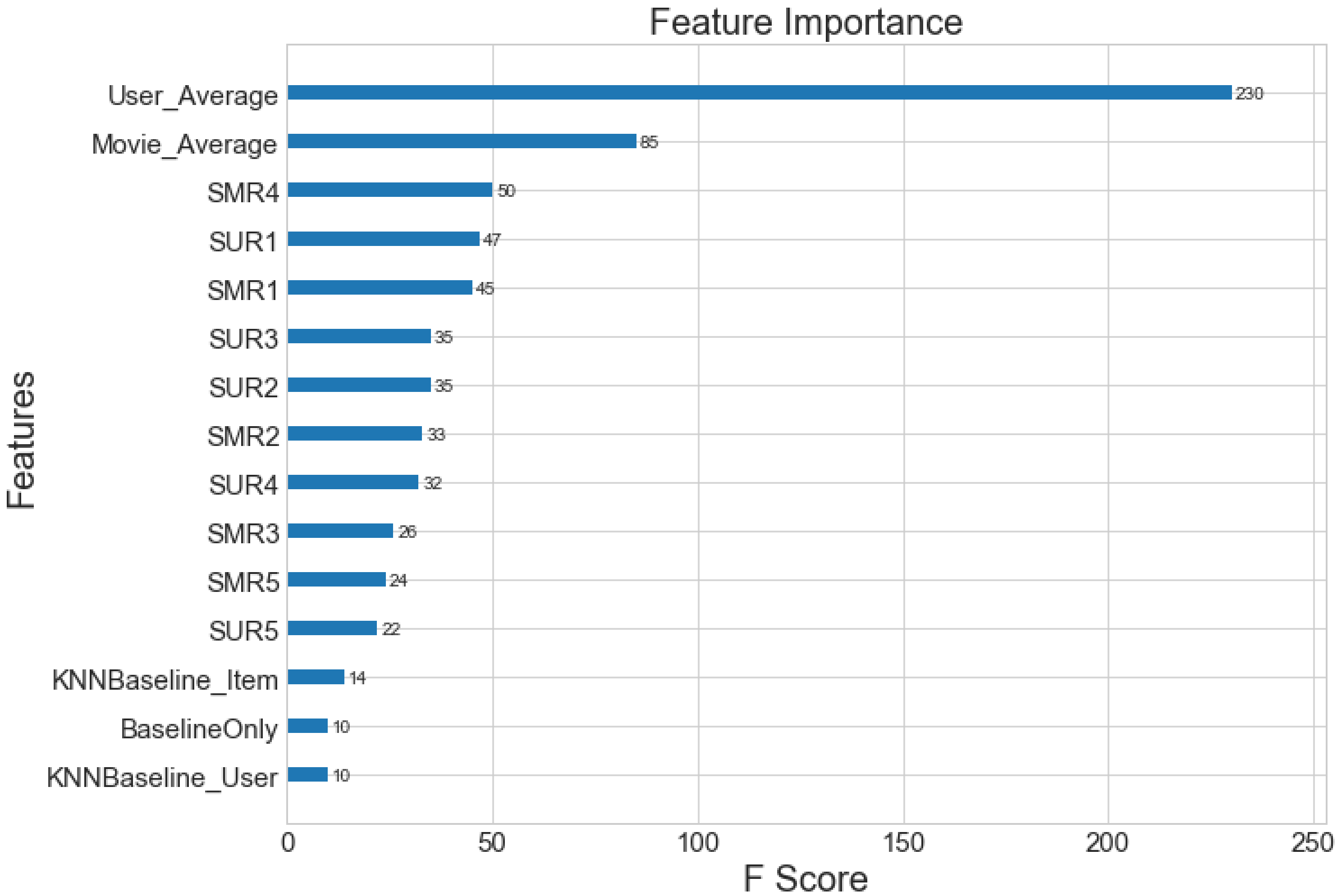
	User_ID	Movie_ID	Global_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	User_Av
0	464626	4614	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
1	1815614	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
2	2298717	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
3	2532402	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5
4	2027	4798	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.5

In [74]:
```
print("Number of nan values in Test Data "+str(Test_Reg.isnull().sum().sum()))
```

```
Number of nan values in Test Data 0
```

In [75]:
```
x_train = Train_Reg.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)

x_test = Test_Reg.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)

y_train = Train_Reg["Rating"]

y_test = Test_Reg["Rating"]

train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test, "XGB_BSL_KNN")

model_train_evaluation["XGB_BSL_KNN"] = train_result
model_test_evaluation["XGB_BSL_KNN"] = test_result
```

```
--------------------------------------------------
TRAIN DATA
RMSE = 0.8105482928866625
MAPE = 24.165594577789307
--------------------------------------------------
TEST DATA
RMSE = 1.06927734319224
MAPE = 33.32028125334464
--------------------------------------------------
Time Taken = 0:00:00.911646
```



6. Matrix Factorization SVD

Prediction $\hat{r}_{ui}$ is set as:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$$

- $\boldsymbol{q}_i$ - Representation of item(movie) in latent factor space

- $\boldsymbol{p}_u$ - Representation of user in new latent factor space

If user u is unknown, then the bias b_u and the factors p_u are assumed to be zero. The same applies for item i with b_i and q_i.

Optimization Problem

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

SVD Documentation: https://surprise.readthedocs.io/en/stable/matrix_factorization.html (https://surprise.readthedocs.io/en/stable/matrix_factorization.html)

Cross- Validation

In [90]:
```
param_grid  = {'n_factors': [5,7,10,15,20,25,35,50,70,90]}   #here, n_factors is the equivalent to dimension 'd' when mat
#is broken into 'b' and 'c'. So, matrix 'A' will be of dimension n*m. So, matrices 'b' and 'c' will be of dimension n*d a

gs = GridSearchCV(SVD, param_grid, measures=['rmse', 'mae'], cv=3)

gs.fit(data)

# best RMSE score
print(gs.best_score['rmse'])

# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
```

```
0.9917179812600221
{'n_factors': 5}
```

Applying SVD with best parameters

In [91]:
```
algo = SVD(n_factors = gs.best_params['rmse']['n_factors'], biased=True, verbose=True)

train_result, test_result = run_surprise(algo, trainset, testset, "SVD")

model_train_evaluation["SVD"] = train_result
model_test_evaluation["SVD"] = test_result
```

```
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
--------------------------------------------------
TRAIN DATA
RMSE = 0.8914245646368677
MAPE = 27.90407603935644
--------------------------------------------------
TEST DATA
RMSE = 1.0676670421292496
MAPE = 33.39888276741577
--------------------------------------------------
Time Taken = 0:00:00.716508
```

7. Matrix Factorization SVDpp with implicit feedback

Prediction $\hat{r}_{ui}$ is set as:

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

- $\pmb{I_u}$ --- the set of all items rated by user u. $|I_u|$ is a length of that set.
- $\pmb{y_j}$ --- Our new set of item factors that capture implicit ratings. Here, an implicit rating describes the fact that a user u rated an item j, regardless of the rating value. y_i is an item vector. For every item j, there is an item vector y_j which is an implicit feedback. Implicit feedback indirectly reflects opinion by observing user behavior including purchase history, browsing history, search patterns, or even mouse movements. Implicit feedback usually denotes the presence or absence of an event. For example, there is a movie 10 where user has just checked the details of the movie and spend some time there, will contribute to implicit rating. Now, since here Netflix has not provided us the details that for how long a user has spend time on the movie, so here we are considering the fact that even if a user has rated some movie then it means that he has spend some time on that movie which contributes to implicit rating.

If user u is unknown, then the bias b_u and the factors p_u are assumed to be zero. The same applies for item i with b_i, q_i and y_i.

Optimization Problem

$$\sum_{r_{ui} \in R_{train}} \left(r_{ui} - \hat{r}_{ui}\right)^2 + \lambda \left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2 + ||y_j||^2\right) . \left[minimize\ b_u, b_i, q_i, p_u, y_j\right]$$

SVDpp Documentation: [https://surprise.readthedocs.io/en/stable/matrix_factorization.html (https://surprise.readthedocs.io/en/stable/matrix_factorization.html)](https://surprise.readthedocs.io/en/stable/matrix_factorization.html)

Cross- Validation

In [109]:
```
param_grid = {'n_factors': [10, 30, 50, 80, 100], 'lr_all': [0.002, 0.006, 0.018, 0.054, 0.10]}

gs = GridSearchCV(SVDpp, param_grid, measures=['rmse', 'mae'], cv=3)

gs.fit(data)

# best RMSE score
print(gs.best_score['rmse'])

# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
```

```
0.9912340650066573
{'n_factors': 10, 'lr_all': 0.006}
```

Applying SVDpp with best parameters

In [111]:
```
algo = SVDpp(n_factors = gs.best_params['rmse']['n_factors'], lr_all = gs.best_params['rmse']["lr_all"], verbose=True)

train_result, test_result = run_surprise(algo, trainset, testset, "SVDpp")

model_train_evaluation["SVDpp"] = train_result
model_test_evaluation["SVDpp"] = test_result
```

```
 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
--------------------------------------------------
TRAIN DATA
RMSE = 0.7891759935507388
MAPE = 24.165955103679742
--------------------------------------------------
TEST DATA
RMSE = 1.0675830366748182
MAPE = 33.396452697149705
--------------------------------------------------
Time Taken = 0:00:07.075453
```

8. XGBoost 13 Features + Surprise BaselineOnly + Surprise KNN Baseline + SVD + SVDpp

In [112]:
```
Train_Reg["SVD"] = model_train_evaluation["SVD"]["Prediction"]
Train_Reg["SVDpp"] = model_train_evaluation["SVDpp"]["Prediction"]

Test_Reg["SVD"] = model_test_evaluation["SVD"]["Prediction"]
Test_Reg["SVDpp"] = model_test_evaluation["SVDpp"]["Prediction"]
```

In [113]:
```
Train_Reg.head()
```

Out[113]:

	User_ID	Movie_ID	Global_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	...	SMR4	SMR5	User_Average	Movie_Average	Rating
0	180921	4512	3.582804	3.0	2.0	1.0	2.0	1.0	4.0	3.0	...	2.0	2.0	2.900000	2.5	1
1	210185	4512	3.582804	2.0	1.0	3.0	3.0	4.0	3.0	3.0	...	4.0	4.0	3.388889	2.5	2
2	218038	4512	3.582804	2.0	3.0	3.0	2.0	4.0	4.0	4.0	...	3.0	5.0	4.250000	2.5	4
3	221936	4512	3.582804	4.0	2.0	2.0	1.0	2.0	3.0	4.0	...	5.0	3.0	3.458333	2.5	3
4	370736	4512	3.582804	2.0	4.0	1.0	2.0	2.0	4.0	4.0	...	4.0	5.0	4.038462	2.5	4

5 rows × 21 columns

In [114]:
```
print("Number of nan values in Train Data "+str(Train_Reg.isnull().sum().sum()))
```

```
Number of nan values in Train Data 0
```

In [115]:
```
Test_Reg.head()
```

Out[115]:

	User_ID	Movie_ID	Global_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	...	SMR4	SMR5	User_Average
0	464626	4614	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	...	3.582804	3.582804	3.582804
1	1815614	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	...	3.582804	3.582804	3.582804
2	2298717	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	...	3.582804	3.582804	3.582804
3	2532402	4627	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	...	3.582804	3.582804	3.582804
4	2027	4798	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	3.582804	...	3.582804	3.582804	3.582804

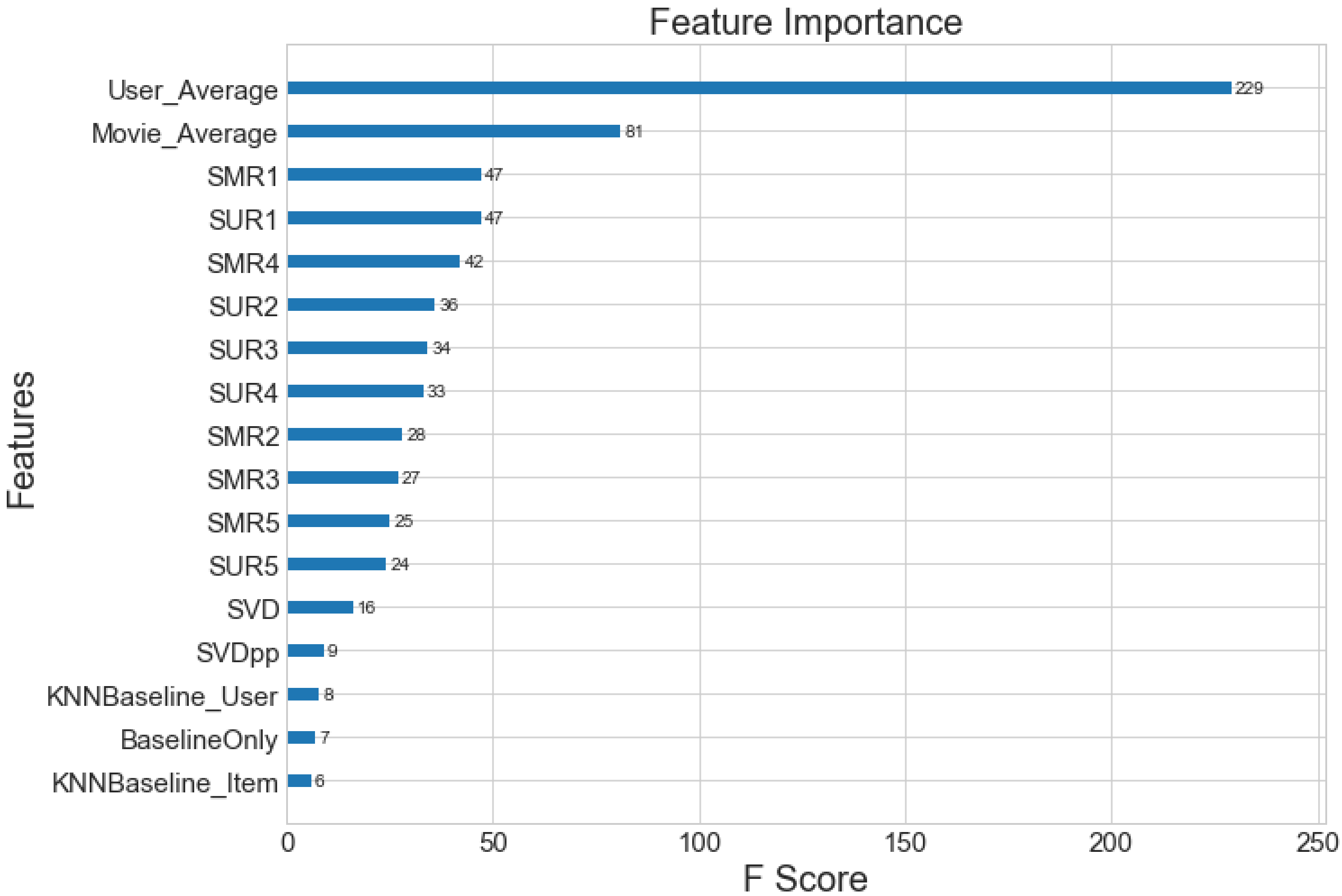
5 rows × 21 columns

In [116]:
```
print("Number of nan values in Test Data "+str(Test_Reg.isnull().sum().sum()))
```

```
Number of nan values in Test Data 0
```

In [117]:
```
x_train = Train_Reg.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)

x_test = Test_Reg.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)

y_train = Train_Reg["Rating"]

y_test = Test_Reg["Rating"]

train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test, "XGB_BSL_KNN_MF")

model_train_evaluation["XGB_BSL_KNN_MF"] = train_result
model_test_evaluation["XGB_BSL_KNN_MF"] = test_result
```

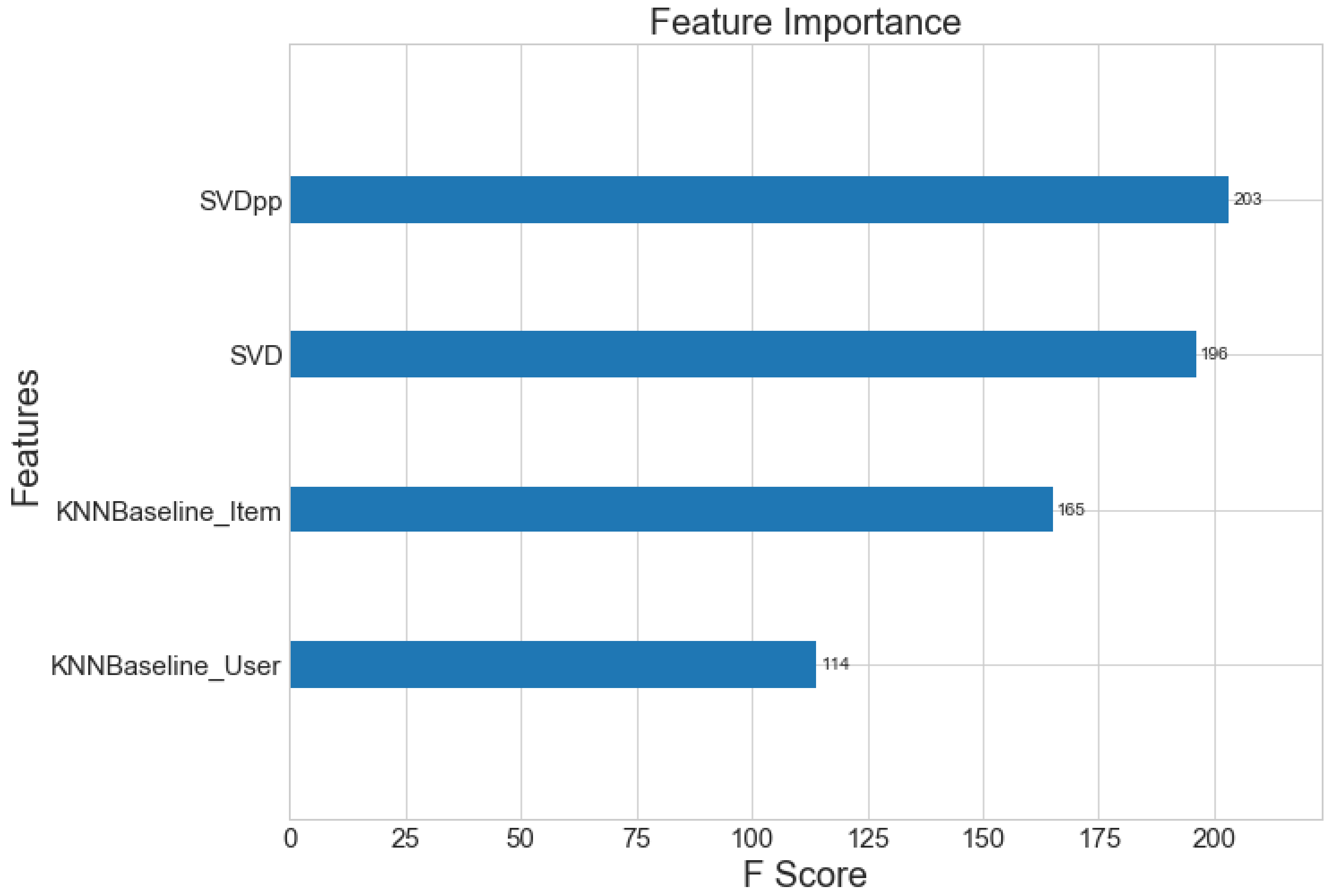
```
--------------------------------------------------
TRAIN DATA
RMSE = 0.8099673298735584
MAPE = 24.15734976530075
--------------------------------------------------
TEST DATA
RMSE = 1.0694262484720989
MAPE = 33.31253186861674
--------------------------------------------------
Time Taken = 0:00:01.035735
```



9. Surprise KNN Baseline + SVD + SVDpp

In [122]:
```
x_train = Train_Reg[["KNNBaseline_User", "KNNBaseline_Item", "SVD", "SVDpp"]]

x_test = Test_Reg[["KNNBaseline_User", "KNNBaseline_Item", "SVD", "SVDpp"]]

y_train = Train_Reg["Rating"]

y_test = Test_Reg["Rating"]

train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test, "XGB_KNN_MF")

model_train_evaluation["XGB_KNN_MF"] = train_result
model_test_evaluation["XGB_KNN_MF"] = test_result
```

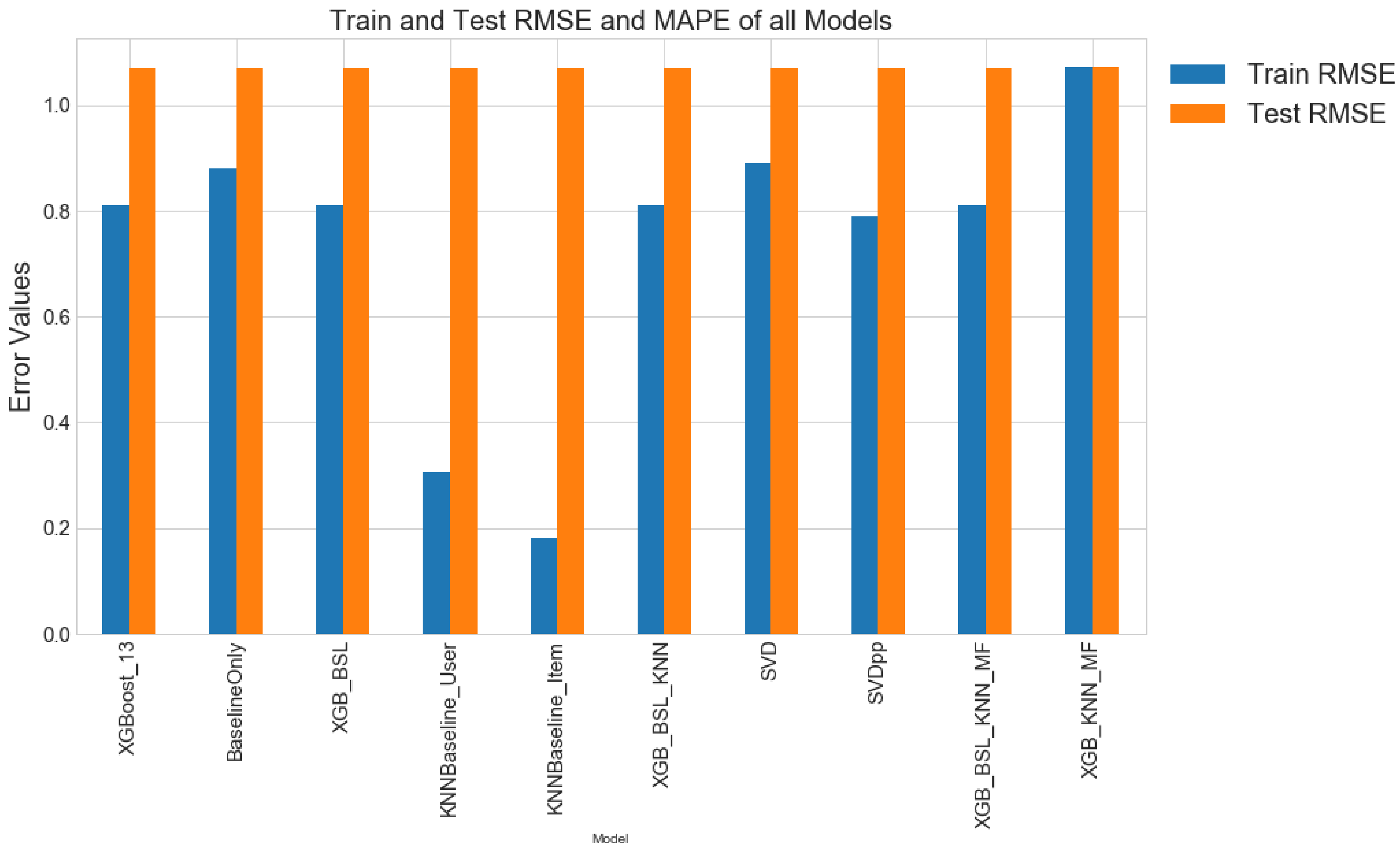
```
--------------------------------------------------
TRAIN DATA
RMSE = 1.072048298658654
MAPE = 35.50347089767456
--------------------------------------------------
TEST DATA
RMSE = 1.0704493877647514
MAPE = 33.36904800491091
--------------------------------------------------
Time Taken = 0:00:00.595421
```



Summary

In [125]:
```
error_table2 = error_table.drop(["Train MAPE", "Test MAPE"], axis = 1)
```

In [140]:
```
error_table2.plot(x = "Model", kind = "bar", figsize = (14, 8), grid = True, fontsize = 15)
plt.title("Train and Test RMSE and MAPE of all Models", fontsize = 20)
plt.ylabel("Error Values", fontsize = 20)
plt.legend(bbox_to_anchor=(1, 1), fontsize = 20)
plt.show()
```



In [151]:
```
error_table.drop(["Train MAPE", "Test MAPE"], axis = 1).style.highlight_min(axis=0)
```

Out[151]:

	Model	Train RMSE	Test RMSE
0	XGBoost_13	0.810186	1.06861
1	BaselineOnly	0.881143	1.06784
2	XGB_BSL	0.809892	1.06768
3	KNNBaseline_User	0.304498	1.06765
4	KNNBaseline_Item	0.181882	1.06765
5	XGB_BSL_KNN	0.810548	1.06928
6	SVD	0.891425	1.06767
7	SVDpp	0.789176	1.06758
8	XGB_BSL_KNN_MF	0.809967	1.06943
9	XGB_KNN_MF	1.07205	1.07045

So, far our best model is SVDpp with Test RMSE of 1.067583