**Movie Recommendation System**

**1. Business Objective**

All entertainment websites or online stores have millions/billions of items. It becomes challenging for the customer to select the right one. At this place, recommender systems come into the picture and help the user to find the right item by minimizing the options.

Recommendation Systems in the world of machine learning have become very popular and are a huge advantage to tech giants like Netflix, Amazon and many more to target their content to a specific audience. These recommendation engines are so strong in their predictions that they can dynamically alter the state of what the user sees on their page based on the user’s interaction with the app.

The business objective for us is:

1. To create a Collaborative Filtering based Movie Recommendation System.
2. Predict the rating that a user would give to a movie that he has not yet rated.
3. Minimize the difference between predicted and actual rating (RMSE and MAPE).

**2. Data Collection**

The dataset has been obtained from Grouplens.

Link : <https://grouplens.org/datasets/movielens/20m/>

This dataset (ml-20m) describes 5-star rating and free-text tagging activity from MovieLens, a movie recommendation service. It contains 20000263 ratings and 465564 tag applications across 27278 movies. These data were created by 138493 users between January 09, 1995 and March 31, 2015. This dataset was generated on October 17, 2016.

Users were selected at random for inclusion. All selected users had rated at least 20 movies. No demographic information is included. Each user is represented by an id, and no other information is provided.

The data are contained in the files genome-scores.csv, genome-tags.csv, links.csv, movies.csv, ratings.csv and tags.csv.

For our objective, we would be using "ratings.csv" and "movies.csv" data files.

In [ ]:

*# Connecting to Google drive*

from google.colab import drive

drive.mount('/content/drive')

Mounted at /content/drive

In [ ]:

*# Importing the necessary libraries*

import numpy as np

import pandas as pd

import matplotlib

import matplotlib.pyplot as plt

import seaborn as sns

import warnings

warnings.filterwarnings("ignore")

In [ ]:

*# Setting up some parameters for the workbook*

pd.set\_option('display.max\_rows', 500)

pd.options.display.max\_columns = None

%matplotlib inline

matplotlib.rcParams["figure.figsize"] = (25,5)

In [ ]:

!pip install fuzzywuzzy

!pip install scikit-surprise

Collecting fuzzywuzzy

Downloading https://files.pythonhosted.org/packages/43/ff/74f23998ad2f93b945c0309f825be92e04e0348e062026998b5eefef4c33/fuzzywuzzy-0.18.0-py2.py3-none-any.whl

Installing collected packages: fuzzywuzzy

Successfully installed fuzzywuzzy-0.18.0

Collecting scikit-surprise

Downloading https://files.pythonhosted.org/packages/97/37/5d334adaf5ddd65da99fc65f6507e0e4599d092ba048f4302fe8775619e8/scikit-surprise-1.1.1.tar.gz (11.8MB)

|████████████████████████████████| 11.8MB 317kB/s

Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.6/dist-packages (from scikit-surprise) (0.17.0)

Requirement already satisfied: numpy>=1.11.2 in /usr/local/lib/python3.6/dist-packages (from scikit-surprise) (1.18.5)

Requirement already satisfied: scipy>=1.0.0 in /usr/local/lib/python3.6/dist-packages (from scikit-surprise) (1.4.1)

Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.6/dist-packages (from scikit-surprise) (1.15.0)

Building wheels for collected packages: scikit-surprise

Building wheel for scikit-surprise (setup.py) ... done

Created wheel for scikit-surprise: filename=scikit\_surprise-1.1.1-cp36-cp36m-linux\_x86\_64.whl size=1670917 sha256=022c3578c27572b83fe8f284d66360ea5edc7335896db717f3c8bd080620f05a

Stored in directory: /root/.cache/pip/wheels/78/9c/3d/41b419c9d2aff5b6e2b4c0fc8d25c538202834058f9ed110d0

Successfully built scikit-surprise

Installing collected packages: scikit-surprise

Successfully installed scikit-surprise-1.1.1

In [ ]:

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 SlopeOne

from surprise import SVD

from surprise import SVDpp

from surprise.model\_selection import GridSearchCV

In [ ]:

from datetime import datetime

import os

import random

import gc

from fuzzywuzzy import fuzz

from fuzzywuzzy import process

**3. Data Preparation/Preprocessing**

We will start with loading and familiarizing with the dataset so that we can prepare the data for Machine Learning (ML) modelling.

In [ ]:

*# Loading the dataset*

file\_path = "/content/drive/MyDrive/Colab Datasets/Movie Recommendation"

movie\_ratings = pd.read\_csv(file\_path + "/ratings.csv")

movies = pd.read\_csv(file\_path + "/movies.csv")

In [ ]:

*# Creating a newId for every movie to reduce the range of existing movieId*

movies["newId"] = range(1, movies["movieId"].nunique()+1)

In [ ]:

*# Converting the the UTC timestamp to Datetime*

movie\_ratings["timestamp"] = movie\_ratings["timestamp"].apply(lambda x: datetime.utcfromtimestamp(x).strftime("%Y-%m-%d"))

*# Merging the movies and ratings data files*

movie\_ratings = movie\_ratings.merge(movies, how="left", on="movieId")

*# Renaming the timestamp to date*

movie\_ratings.rename(columns={"timestamp": "date"}, inplace=True)

*# Updating the movieId with the newId*

movie\_ratings["movieId"] = movie\_ratings["newId"]

movies["movieId"] = movies["newId"]

In [ ]:

*# Dropping the newId from the datasets*

movie\_ratings.drop(["newId"], axis=1, inplace=True)

movies.drop(["newId"], axis=1, inplace=True)

*# Sorting ratings based on date*

movie\_ratings.sort\_values(by = "date", inplace = True)

movie\_ratings.reset\_index(drop=True, inplace=True)

In [ ]:

*# Checking the features and no. of records in the dataset*

print("The number of records are : ", movie\_ratings.shape[0])

print("The number of features are : ", movie\_ratings.shape[1])

print("The list of features is : ", movie\_ratings.columns)

movie\_ratings.head()

The number of records are : 20000263

The number of features are : 6

The list of features is : Index(['userId', 'movieId', 'rating', 'date', 'title', 'genres'], dtype='object')

Out[ ]:

|  | **userId** | **movieId** | **rating** | **date** | **title** | **genres** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 131160 | 1058 | 3.0 | 1995-01-09 | Fish Called Wanda, A (1988) | Comedy|Crime |
| **1** | 131160 | 47 | 5.0 | 1995-01-09 | Seven (a.k.a. Se7en) (1995) | Mystery|Thriller |
| **2** | 28507 | 1154 | 4.0 | 1995-01-09 | Double Life of Veronique, The (Double Vie de V... | Drama|Fantasy|Romance |
| **3** | 131160 | 21 | 3.0 | 1995-01-09 | Get Shorty (1995) | Comedy|Crime|Thriller |
| **4** | 85252 | 7 | 5.0 | 1996-01-29 | Sabrina (1995) | Comedy|Romance |

Observations:

1. There are 20M+ records of the data.
2. There are 6 features: userId, movieId, rating, date, title and genres.

**3.1 Data Cleaning**

We will begin with data cleaning such that we can handle missing values, outliers, rare values and drop the unnecessary features that do not carry useful information.

In [ ]:

*# Checking for duplicates*

print("No. of duplicates records in the dataset : ", movie\_ratings.columns.duplicated().sum())

No. of duplicates records in the dataset : 0

Observations:

1. There are no duplicate records in the dataset.

In [ ]:

*# Checking the columns' titles and datatypes*

movie\_ratings.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 20000263 entries, 0 to 20000262

Data columns (total 6 columns):

# Column Dtype

--- ------ -----

0 userId int64

1 movieId int64

2 rating float64

3 date object

4 title object

5 genres object

dtypes: float64(1), int64(2), object(3)

memory usage: 915.5+ MB

**3.1.1 Handling Missing Values**

Identifying the features that have some missing values and imputing them.

In [ ]:

*# Checking the number of missing values in data*

movie\_ratings.isna().sum()

Out[ ]:

userId 0

movieId 0

rating 0

date 0

title 0

genres 0

dtype: int64

Observations:

1. It looks like that the dataset is well maintained as we do not see any missing values, which is good.

**3.2 Exploratory Data Analysis**

After the data cleaning steps, we can now perform EDA on the dataset to discover patterns and relationships that will help in understanding the data better.

**3.2.1 Univariate Analysis**

Analyzing each feature inidividually to gain insights from the data and discover any outliers.

In [ ]:

*# Checking the feature "userID"*

total\_users = len(np.unique(movie\_ratings["userId"]))

print("The count of unique userID in the dataset is : ", total\_users)

print("The top 5 userID in the dataset are : \n", movie\_ratings["userId"].value\_counts()[:5])

The count of unique userID in the dataset is : 138493

The top 5 userID in the dataset are :

118205 9254

8405 7515

82418 5646

121535 5520

125794 5491

Name: userId, dtype: int64

Observations:

1. "userId" are the Users that were selected at random for inclusion and their ids have been anonymized.
2. There are 138K+ unique users in the dataset.
3. userId 118205 has around 9K records in the dataset.

In [ ]:

*# Checking the feature "movieID"*

total\_movies = len(np.unique(movie\_ratings["movieId"]))

print("The count of unique movieID in the dataset is : ", total\_movies)

print("The top 5 movieID in the dataset are : \n", movie\_ratings["movieId"].value\_counts()[:5])

The count of unique movieID in the dataset is : 26744

The top 5 movieID in the dataset are :

294 67310

353 66172

316 63366

588 63299

477 59715

Name: movieId, dtype: int64

Observations:

1. "movieId" represents the movies with at least one rating or tag in the dataset.
2. There are close to 26K+ unique movies in the dataset.
3. movieId 294, 353, 316 and 588 are few popular movies which has been rated over 60K times.

In [ ]:

*# Helper function to Change the numeric label in terms of Millions*

def changingLabels(number):

return str(number/10\*\*6) + "M"

In [ ]:

*# Checking the feature "rating"*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.countplot("rating", data=movie\_ratings, ax=axes)

axes.set\_yticklabels([changingLabels(num) for num in axes.get\_yticks()])

for p in axes.patches:

axes.annotate('{}'.format(p.get\_height()), (p.get\_x()+0.2, p.get\_height()+100))

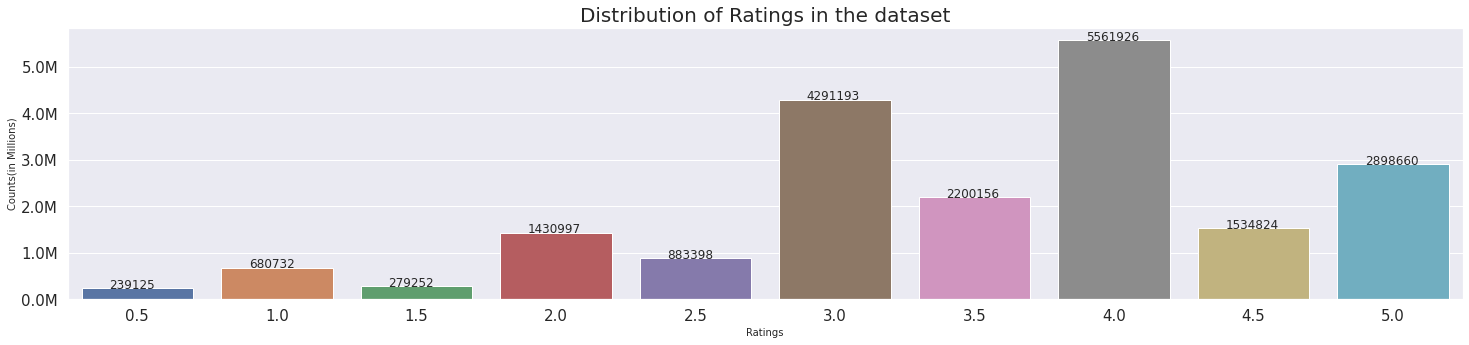
plt.tick\_params(labelsize = 15)

plt.title("Distribution of Ratings in the dataset", fontsize = 20)

plt.xlabel("Ratings", fontsize = 10)

plt.ylabel("Counts(in Millions)", fontsize = 10)

plt.show()



Observations:

1. The ratings given by users to movies lies in between 0.5 to 5.
2. A high proportion of the movies have been rated 3, 3.5 or 4 by the users.
3. The distribution of ratings look a bit left skewed as large proportion of ratings is in between 3 to 5.

In [ ]:

*# Checking the feature "date"*

print("The count of unique date in the dataset is : ", movie\_ratings["date"].nunique())

print("The first rating was given on : ", movie\_ratings["date"].min())

print("The latest rating was given on : ", movie\_ratings["date"].max())

print("The top 5 date in the dataset are : \n", movie\_ratings["date"].value\_counts()[:5])

The count of unique date in the dataset is : 6911

The first rating was given on : 1995-01-09

The latest rating was given on : 2015-03-31

The top 5 date in the dataset are :

2000-11-20 91753

2005-03-22 76568

1999-12-11 65077

2008-10-29 55163

2000-11-21 54131

Name: date, dtype: int64

Observations:

1. There are ~7K unique dates when the ratings were given by a user to a movie.
2. The first rating was given on 1995-01-09 and the latest rating was given on 2015-03-31.
3. Around 91K+ ratings were observed on 2000-11-20.

In [ ]:

*# Checking the feature "title"*

movie\_list = movie\_ratings["title"].unique()

print("The count of unique title in the dataset is : ", movie\_ratings["title"].nunique())

print("The top 5 title in the dataset are : \n", movie\_ratings["title"].value\_counts()[:5])

The count of unique title in the dataset is : 26729

The top 5 title in the dataset are :

Pulp Fiction (1994) 67310

Forrest Gump (1994) 66172

Shawshank Redemption, The (1994) 63366

Silence of the Lambs, The (1991) 63299

Jurassic Park (1993) 59715

Name: title, dtype: int64

Observations:

1. There are 26K+ unique movie titles in the dataset.
2. Pulp Fiction, Forrest Gump, Shawshank Redemption and Silence of the Lambs are the top 4 movies in terms of no. of ratings received which are over 60K+ for each one.

In [ ]:

*# Extract unique Genres along with their count*

unique\_genres = {}

def ExtractGenres(x):

for g in x.split("|"):

if g not in unique\_genres.keys():

unique\_genres[g] = 1

else:

unique\_genres[g] = unique\_genres[g] + 1

movie\_ratings["genres"].apply(ExtractGenres)

print("Genres Extracted from the dataset.")

Genres Extracted from the dataset.

In [ ]:

*# Visualizing the feature "Genres"*

genres\_df = pd.DataFrame(list(unique\_genres.items()))

genres\_df.columns = ["Genre", "Count"]

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 8), sharey=True)

sns.barplot(y="Count", x="Genre", data=genres\_df, ax=axes)

axes.set\_yticklabels([changingLabels(num) for num in axes.get\_yticks()])

for p in axes.patches:

axes.annotate('{}'.format(int(p.get\_height())), (p.get\_x(), p.get\_height()+100))

plt.tick\_params(labelsize = 15)

plt.title("Distribution of Genres in the dataset", fontsize = 20)

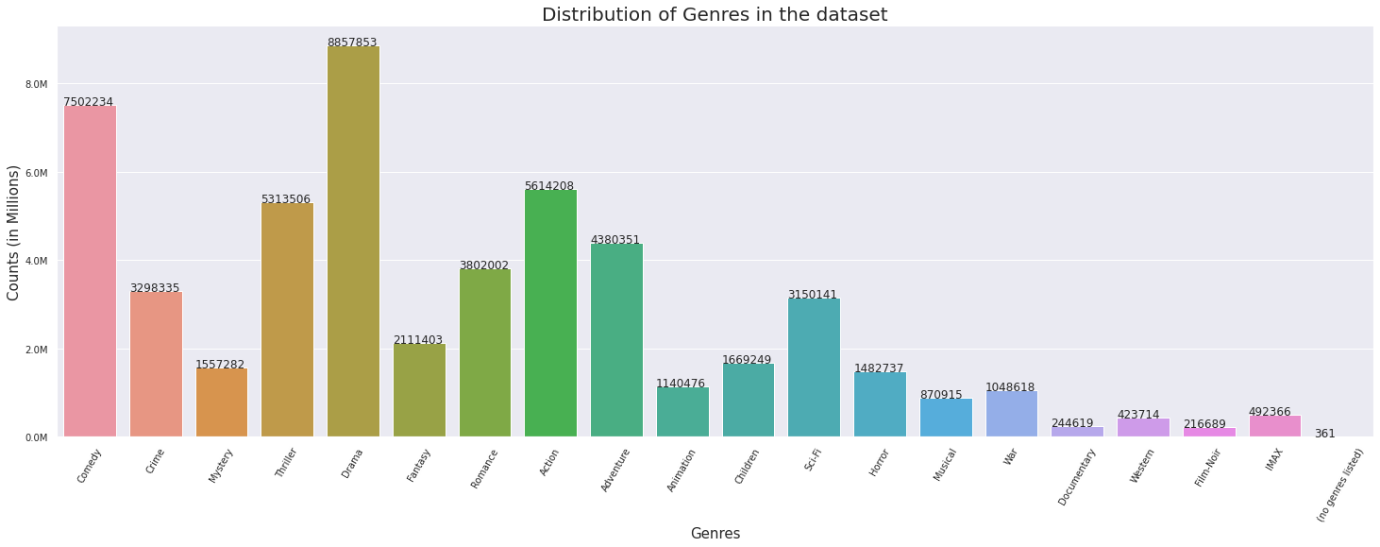
plt.xlabel("Genres", fontsize = 15)

plt.xticks(rotation=60, fontsize=10)

plt.yticks(fontsize=10)

plt.ylabel("Counts (in Millions)", fontsize = 15)

plt.show()



Observations:

1. There are 19 different genres of movies while there are few whose genre has not been mentioned.
2. Drama, Comedy, Action and Thriller are top 4 genres of movies present in the dataset.

In [ ]:

movie\_ratings.head()

Out[ ]:

|  | **userId** | **movieId** | **rating** | **date** | **title** | **genres** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 131160 | 1058 | 3.0 | 1995-01-09 | Fish Called Wanda, A (1988) | Comedy|Crime |
| **1** | 131160 | 47 | 5.0 | 1995-01-09 | Seven (a.k.a. Se7en) (1995) | Mystery|Thriller |
| **2** | 28507 | 1154 | 4.0 | 1995-01-09 | Double Life of Veronique, The (Double Vie de V... | Drama|Fantasy|Romance |
| **3** | 131160 | 21 | 3.0 | 1995-01-09 | Get Shorty (1995) | Comedy|Crime|Thriller |
| **4** | 85252 | 7 | 5.0 | 1996-01-29 | Sabrina (1995) | Comedy|Romance |

**3.2.2 Train & test Splitting**

Splitting the data into train and test sets before proceeding towards further EDA and Feature Engineering.

In [ ]:

*# Creating the train test set*

file\_path = "/content/drive/MyDrive/Colab Datasets/Movie Recommendation"

if not os.path.isfile(file\_path + "/TrainData.pkl"):

print("Creating Train Data and saving it..")

movie\_ratings.iloc[:int(movie\_ratings.shape[0] \* 0.80)].to\_pickle(file\_path + "/TrainData.pkl")

Train\_Data = pd.read\_pickle(file\_path + "/TrainData.pkl")

Train\_Data.reset\_index(drop = True, inplace = True)

else:

print("Loading Train Data..")

Train\_Data = pd.read\_pickle(file\_path + "/TrainData.pkl")

Train\_Data.reset\_index(drop = True, inplace = True)

if not os.path.isfile(file\_path + "/TestData.pkl"):

print("Creating Test Data and saving it..")

movie\_ratings.iloc[int(movie\_ratings.shape[0] \* 0.80):].to\_pickle(file\_path + "/TestData.pkl")

Test\_Data = pd.read\_pickle(file\_path + "/TestData.pkl")

Test\_Data.reset\_index(drop = True, inplace = True)

else:

print("Loading Test Data..")

Test\_Data = pd.read\_pickle(file\_path + "/TestData.pkl")

Test\_Data.reset\_index(drop = True, inplace = True)

Loading Train Data..

Loading Test Data..

In [ ]:

Train\_Data.head()

Out[ ]:

|  | **userId** | **movieId** | **rating** | **date** | **title** | **genres** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 131160 | 1058 | 3.0 | 1995-01-09 | Fish Called Wanda, A (1988) | Comedy|Crime |
| **1** | 131160 | 47 | 5.0 | 1995-01-09 | Seven (a.k.a. Se7en) (1995) | Mystery|Thriller |
| **2** | 28507 | 1154 | 4.0 | 1995-01-09 | Double Life of Veronique, The (Double Vie de V... | Drama|Fantasy|Romance |
| **3** | 131160 | 21 | 3.0 | 1995-01-09 | Get Shorty (1995) | Comedy|Crime|Thriller |
| **4** | 85252 | 7 | 5.0 | 1996-01-29 | Sabrina (1995) | Comedy|Romance |

In [ ]:

*# Creating list of unique movies from Train Set*

movie\_list\_in\_training = Train\_Data.drop\_duplicates(subset=["title"], keep="first")[["movieId", "title", "genres"]]

movie\_list\_in\_training = movie\_list\_in\_training.reset\_index(drop=True)

movie\_list\_in\_training.head()

Out[ ]:

|  | **movieId** | **title** | **genres** |
| --- | --- | --- | --- |
| **0** | 1058 | Fish Called Wanda, A (1988) | Comedy|Crime |
| **1** | 47 | Seven (a.k.a. Se7en) (1995) | Mystery|Thriller |
| **2** | 1154 | Double Life of Veronique, The (Double Vie de V... | Drama|Fantasy|Romance |
| **3** | 21 | Get Shorty (1995) | Comedy|Crime|Thriller |
| **4** | 7 | Sabrina (1995) | Comedy|Romance |

In [ ]:

*# Checking the basic statistics for the training data*

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["userId"]))))

print("Number of unique movies in train data : ", str(len(np.unique(Train\_Data["movieId"]))))

Total Train Data..

Total number of movie ratings in train data : 16000210

Number of unique users in train data : 112466

Number of unique movies in train data : 12387

**3.2.3 Bi-variate Analysis**

Analyzing multiple features together to discover relations, correlations and patterns.

**1. Analyzing the Distribution of Ratings**

In [ ]:

*# Checking basic statistics for "rating"*

print("The basic statistics for the feature is : \n", Train\_Data["rating"].describe())

The basic statistics for the feature is :

count 1.600021e+07

mean 3.512613e+00

std 1.059931e+00

min 5.000000e-01

25% 3.000000e+00

50% 3.500000e+00

75% 4.000000e+00

max 5.000000e+00

Name: rating, dtype: float64

In [ ]:

*# Visualizing the "rating" for the train set*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.countplot("rating", data=Train\_Data, ax=axes)

axes.set\_yticklabels([changingLabels(num) for num in axes.get\_yticks()])

for p in axes.patches:

axes.annotate('{}'.format(p.get\_height()), (p.get\_x()+0.2, p.get\_height()+100))

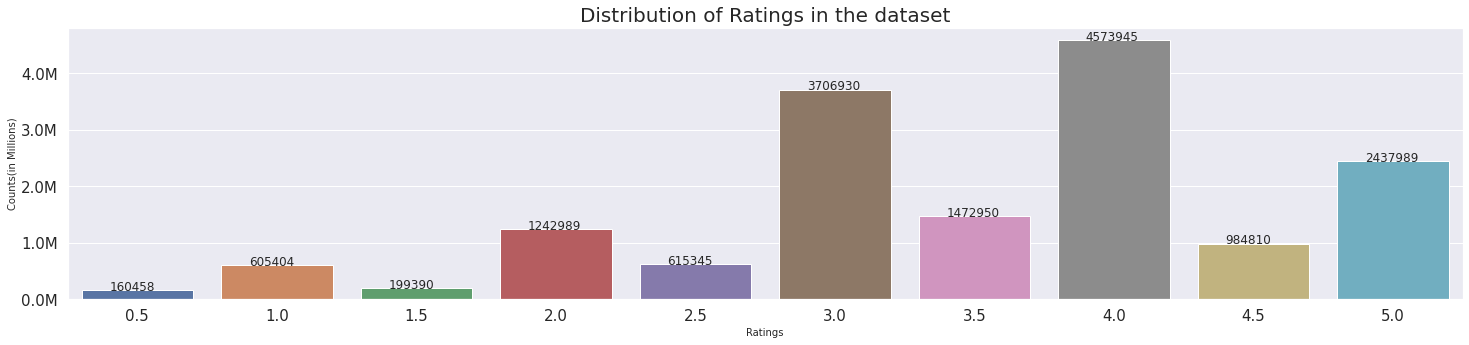
plt.tick\_params(labelsize = 15)

plt.title("Distribution of Ratings in the dataset", fontsize = 20)

plt.xlabel("Ratings", fontsize = 10)

plt.ylabel("Counts(in Millions)", fontsize = 10)

plt.show()



Observations:

1. The distribution of ratings seems to be similar as before.
2. The mean and median value are very close to around 3.5.

**2. Analyzing the number of ratings with date.**

In [ ]:

*# Extracting the day of week from the date when rating was provided*

Train\_Data["date"] = pd.to\_datetime(Train\_Data["date"], errors='coerce')

Train\_Data["DayOfWeek"] = Train\_Data["date"].dt.strftime('%A')

Train\_Data["Weekday"] = Train\_Data["date"].apply(lambda x : 1 if x.dayofweek > 5 else 0)

In [ ]:

*# Converting the number into 'Ks.*

def ChangingLabelsInK(number):

return str(int(number/10\*\*3)) + "K"

In [ ]:

*# Visualizing the count of total ratings made per month*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

axes = Train\_Data.resample("m", on = "date")["rating"].count().plot()

axes.set\_yticklabels([ChangingLabelsInK(num) for num in axes.get\_yticks()])

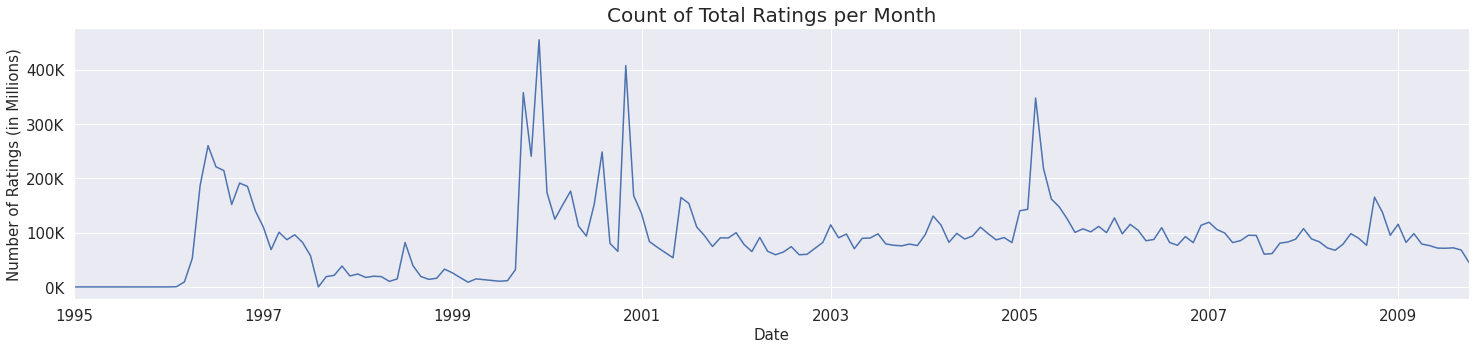
axes.set\_title("Count of Total Ratings per Month", fontsize = 20)

axes.set\_xlabel("Date", fontsize = 15)

axes.set\_ylabel("Number of Ratings (in Millions)", fontsize = 15)

plt.tick\_params(labelsize = 15)

plt.show()



Observations:

1. The no. of ratings per month was very high in few of the months between 1996 to 1998.
2. Similarly, post the 2000s, there are few month that have few months of very high no. of ratings.
3. The count remains steady after 2001 till 2010, with a spike at few month of 2006.

In [ ]:

*# Visualizing the count of ratings by weekday*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.barplot(x="Weekday", y="rating" , data=Train\_Data.groupby(by=["Weekday"], as\_index=False)["rating"].count(), ax=axes)

for p in axes.patches:

axes.annotate('{}'.format(int(p.get\_height())), (p.get\_x(), p.get\_height()+100))

axes.set\_yticklabels([changingLabels(num) for num in axes.get\_yticks()])

plt.tick\_params(labelsize = 15)

plt.title("Distribution of number of ratings by Weekday", fontsize = 20)

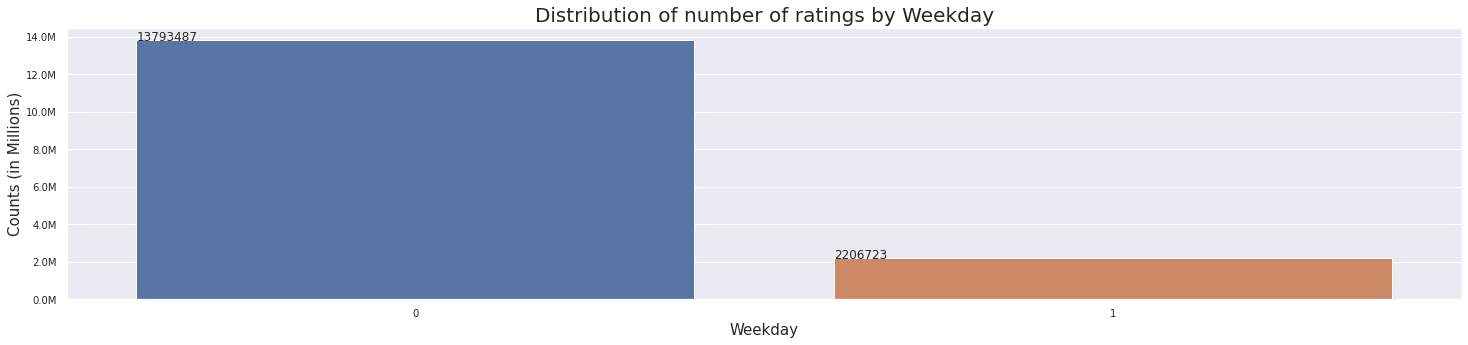
plt.xlabel("Weekday", fontsize = 15)

plt.xticks(fontsize=10)

plt.yticks(fontsize=10)

plt.ylabel("Counts (in Millions)", fontsize = 15)

plt.show()



In [ ]:

*# Visualizing the count of ratings by individual days of the week*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.barplot(x="DayOfWeek", y="rating" , data=Train\_Data.groupby(by=["DayOfWeek"], as\_index=False)["rating"].count(), ax=axes)

for p in axes.patches:

axes.annotate('{}'.format(int(p.get\_height())), (p.get\_x(), p.get\_height()+100))

axes.set\_yticklabels([changingLabels(num) for num in axes.get\_yticks()])

plt.tick\_params(labelsize = 15)

plt.title("Distribution of number of ratings by individual days", fontsize = 20)

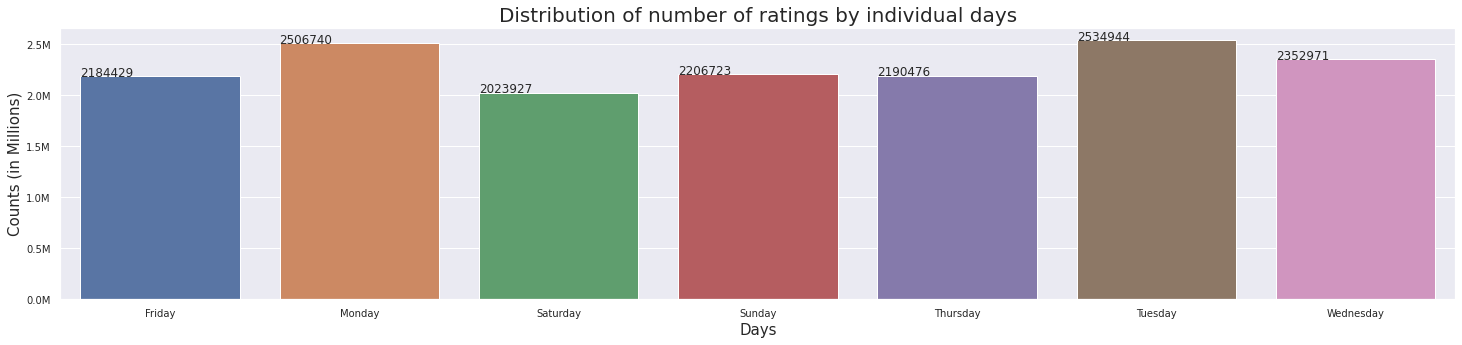
plt.xlabel("Days", fontsize = 15)

plt.xticks(fontsize=10)

plt.yticks(fontsize=10)

plt.ylabel("Counts (in Millions)", fontsize = 15)

plt.show()



Observations:

1. The no. of ratings does not vary too much the days of the week.
2. "Monday" and "Tuesday" clearly has more no. of ratings than any other days.
3. The number of ratings in weekend is clearly extremly less than weekdays.

**3. Analyzing the average ratings by date.**

In [ ]:

*# Visualizing the average ratings by weekday*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.boxplot(x="Weekday", y="rating" , data=Train\_Data, ax=axes)

plt.tick\_params(labelsize = 15)

plt.title("Boxplot of Average Ratings by Weekday", fontsize = 20)

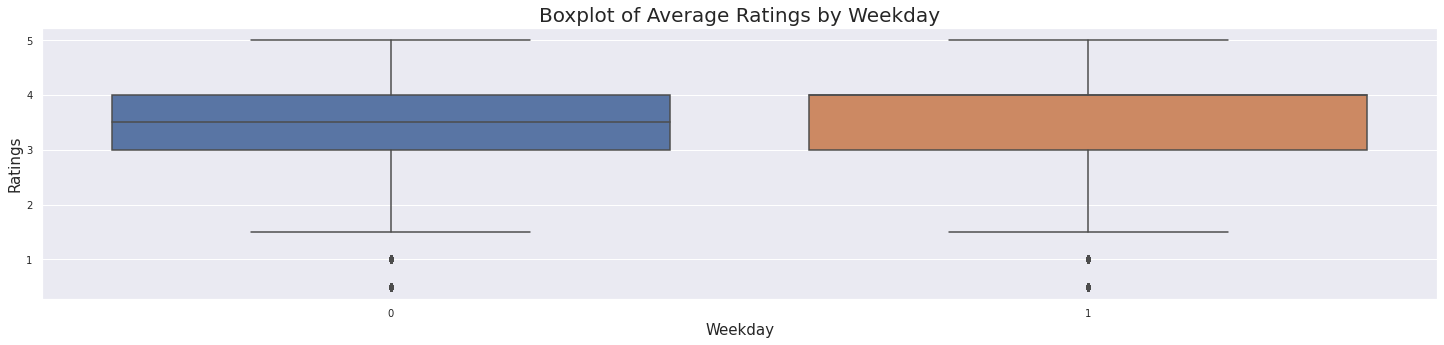
plt.xlabel("Weekday", fontsize = 15)

plt.xticks(fontsize=10)

plt.ylabel("Ratings", fontsize = 15)

plt.yticks(fontsize=10)

plt.show()



In [ ]:

*# Visualizing the average ratings by individual Days of the Week*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.boxplot(x="DayOfWeek", y="rating", data=Train\_Data, ax=axes)

plt.tick\_params(labelsize = 15)

plt.title("Boxplot of Average Ratings by Days of Week", fontsize = 20)

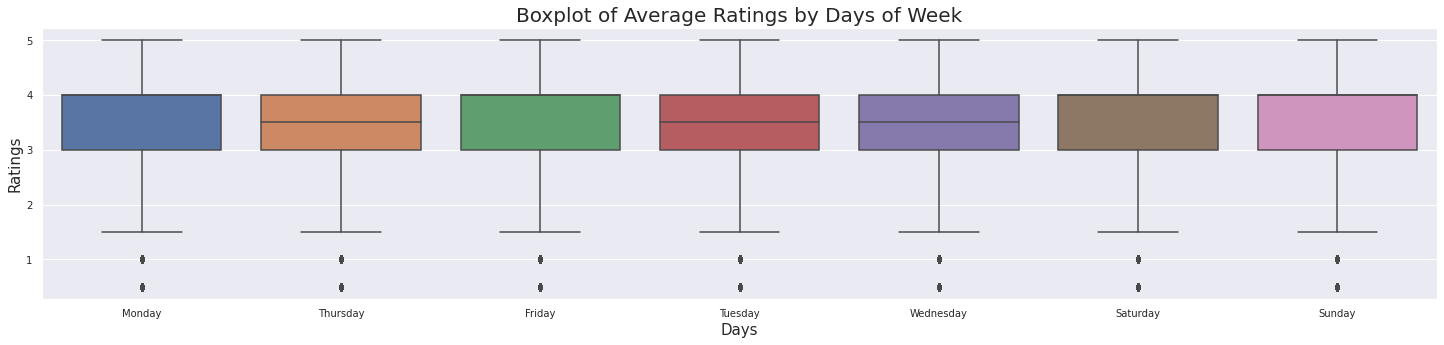
plt.xlabel("Days", fontsize = 15)

plt.xticks(fontsize=10)

plt.ylabel("Ratings", fontsize = 15)

plt.yticks(fontsize=10)

plt.show()



Observations:

1. The average ratings given by the user does not seem to differ by weekday and weekends.
2. Even when we plot the average ratings by individual days, they seem to be similar for all the individual days.

**4. Analyzing the Ratings given by Users.**

In [ ]:

*# Calculating the number of ratings given by individual users*

no\_of\_rated\_movies\_per\_user = Train\_Data.groupby(by=["userId"], as\_index=False)["rating"].count().sort\_values(by="rating", ascending=False)

no\_of\_rated\_movies\_per\_user.reset\_index(drop=True, inplace=True)

In [ ]:

*# Visualizing the count of ratings by individual users*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.barplot(x="userId", y="rating" , data=no\_of\_rated\_movies\_per\_user[:15], ax=axes)

for p in axes.patches:

axes.annotate('{}'.format(int(p.get\_height())), (p.get\_x(), p.get\_height()+100))

axes.set\_yticklabels([ChangingLabelsInK(num) for num in axes.get\_yticks()])

plt.tick\_params(labelsize = 15)

plt.title("Number of ratings for Top 15 Users", fontsize = 20)

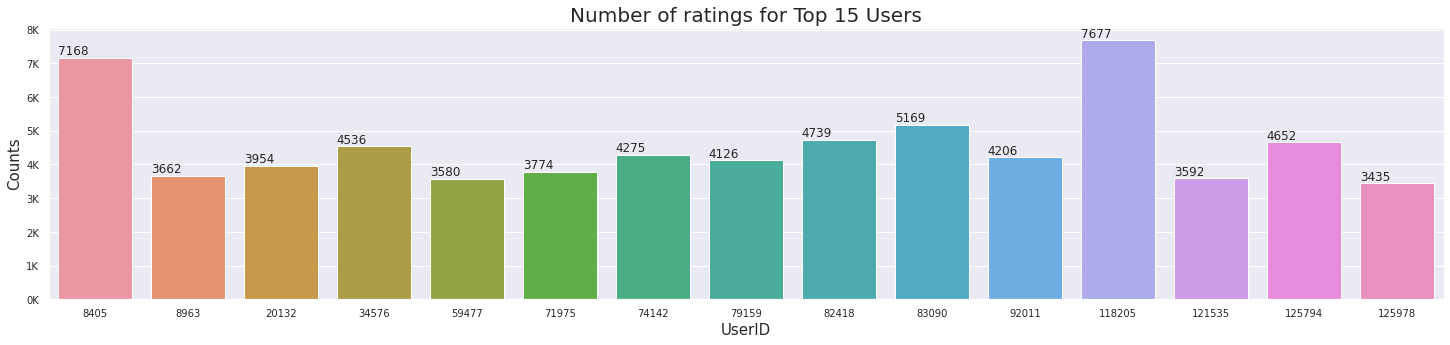
plt.xlabel("UserID", fontsize = 15)

plt.xticks(fontsize=10)

plt.yticks(fontsize=10)

plt.ylabel("Counts", fontsize = 15)

plt.show()



In [ ]:

*# Visualizing the count of ratings by individual users*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 2, figsize=(25, 8))

sns.kdeplot(no\_of\_rated\_movies\_per\_user["rating"], shade = True, ax = axes[0])

axes[0].set\_title("PDF", fontsize = 18)

axes[0].set\_xlabel("Number of Ratings by users", fontsize = 18)

axes[0].tick\_params(labelsize = 15)

sns.kdeplot(no\_of\_rated\_movies\_per\_user["rating"], shade = True, cumulative = True, ax = axes[1])

axes[1].set\_title("CDF", fontsize = 18)

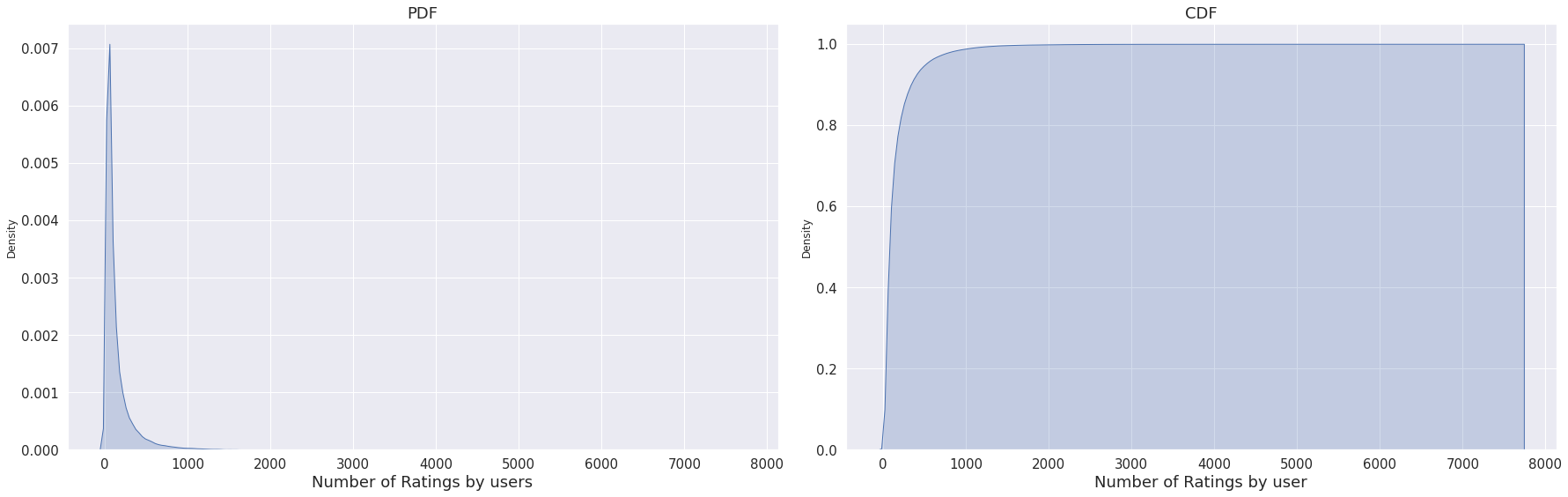
axes[1].set\_xlabel("Number of Ratings by user", fontsize = 18)

axes[1].tick\_params(labelsize = 15)

fig.subplots\_adjust(wspace=2)

plt.tight\_layout()

plt.show()



In [ ]:

*# Checking the basic statistics for the number of ratings per user*

print("Information about no. of ratings by users : \n", no\_of\_rated\_movies\_per\_user["rating"].describe())

Information about no. of ratings by users :

count 112466.000000

mean 142.267085

std 214.808118

min 1.000000

25% 35.000000

50% 69.000000

75% 155.000000

max 7677.000000

Name: rating, dtype: float64

In [ ]:

quantiles = no\_of\_rated\_movies\_per\_user["rating"].quantile(np.arange(0.9, 1.01,0.01))

qvalue = np.arange(0.9, 1.01,0.01)

for ctr in qvalue:

print("The {}th quantile value is : {}".format(int(ctr\*100), quantiles[ctr]))

The 90th quantile value is : 333.0

The 91th quantile value is : 358.0

The 92th quantile value is : 386.0

The 93th quantile value is : 418.0

The 94th quantile value is : 459.0

The 95th quantile value is : 510.0

The 96th quantile value is : 573.0

The 97th quantile value is : 664.0

The 98th quantile value is : 804.7000000000116

The 99th quantile value is : 1060.3500000000058

The 100th quantile value is : 7677.0

In [ ]:

*# Plotting the quantile values*

quantiles = no\_of\_rated\_movies\_per\_user["rating"].quantile(np.arange(0, 1.01,0.01))

fig = plt.figure(figsize = (25, 5))

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")

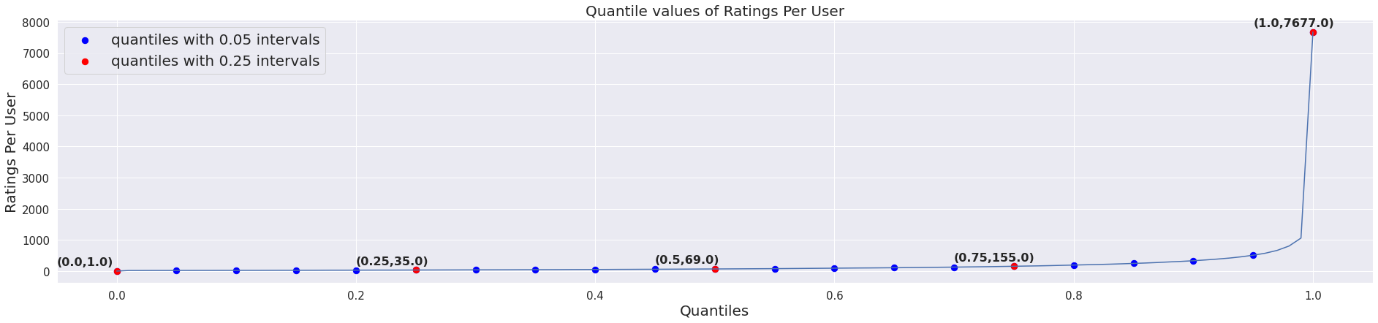
plt.scatter(x = quantiles.index[::25], y = quantiles.values[::25], c = "red", s = 70, label="quantiles with 0.25 intervals")

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)



Observations:

1. The top 10 users tend to have rated more than 4K times, which seems bit extreme behaviour.
2. The userId 118205 has rated over 7K+ times, which seems surprising.
3. From the KDE plot, it is clearly evident that the number of ratings is highly right skewed, and most of the user's ratings is between 0-1000.
4. Similarly, above CDF graph shows that almost 99% of users give very few ratings.
5. The mean no. of ratings a user gives is 142 while the median is 69.
6. The no. of movies start to increase drastically from 90th percentile.

In [ ]:

*# Calculating average ratings given by individual users*

avg\_ratings\_per\_user = Train\_Data.groupby(by = ["userId"], as\_index=False)["rating"].mean()

avg\_ratings\_per\_user = avg\_ratings\_per\_user.reset\_index(drop=True)

avg\_ratings\_per\_user = avg\_ratings\_per\_user.merge(no\_of\_rated\_movies\_per\_user[["userId", "rating"]], how="left", on="userId")

avg\_ratings\_per\_user.rename(columns={"rating\_x":"avg\_rating", "rating\_y": "num\_of\_rating"}, inplace=True)

avg\_ratings\_per\_user = avg\_ratings\_per\_user.sort\_values("num\_of\_rating", ascending=False)

In [ ]:

*# Visualizing the average ratings by individual Days of the Week*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.barplot(x="userId", y="avg\_rating", data=avg\_ratings\_per\_user[:15], ax=axes)

for p in axes.patches:

axes.annotate('{}'.format(round(p.get\_height(), 2)), (p.get\_x()+0.3, p.get\_height()))

plt.tick\_params(labelsize = 15)

plt.title("Average Ratings by top 15 Users", fontsize = 20)

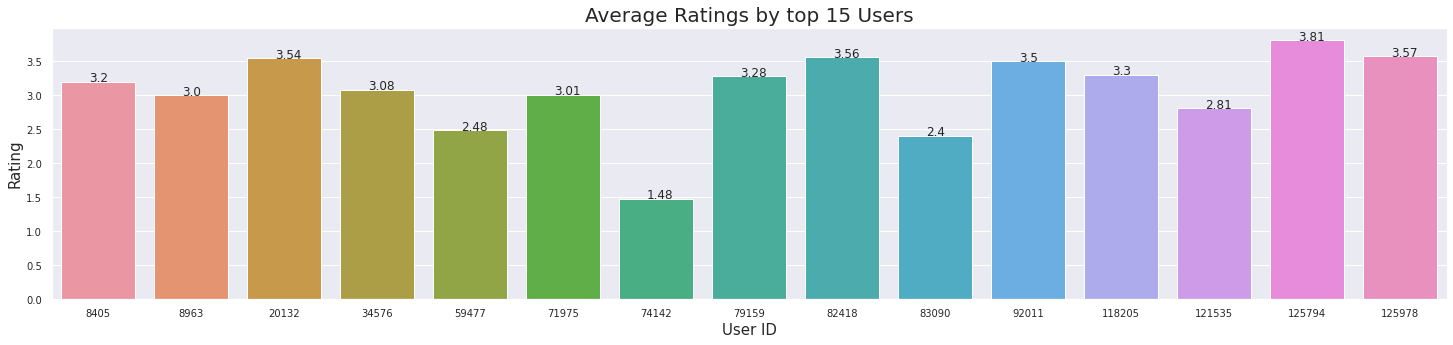
plt.xlabel("User ID", fontsize = 15)

plt.xticks(fontsize=10)

plt.ylabel("Rating", fontsize = 15)

plt.yticks(fontsize=10)

plt.show()



In [ ]:

*# Plotting the PDF and CDF for Avg. rating by Users*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 2, figsize=(25, 5))

fig.suptitle("Avg Ratings per User", fontsize=25)

sns.distplot(avg\_ratings\_per\_user["avg\_rating"], hist = False, ax = axes[0], label = "PDF")

axes[0].set\_title("PDF", fontsize = 18)

axes[0].set\_xlabel("Average Ratings by users", fontsize = 18)

axes[0].tick\_params(labelsize = 15)

sns.kdeplot(avg\_ratings\_per\_user["avg\_rating"], cumulative = True, ax = axes[1], shade=True, label = "CDF")

axes[1].set\_title("CDF", fontsize = 18)

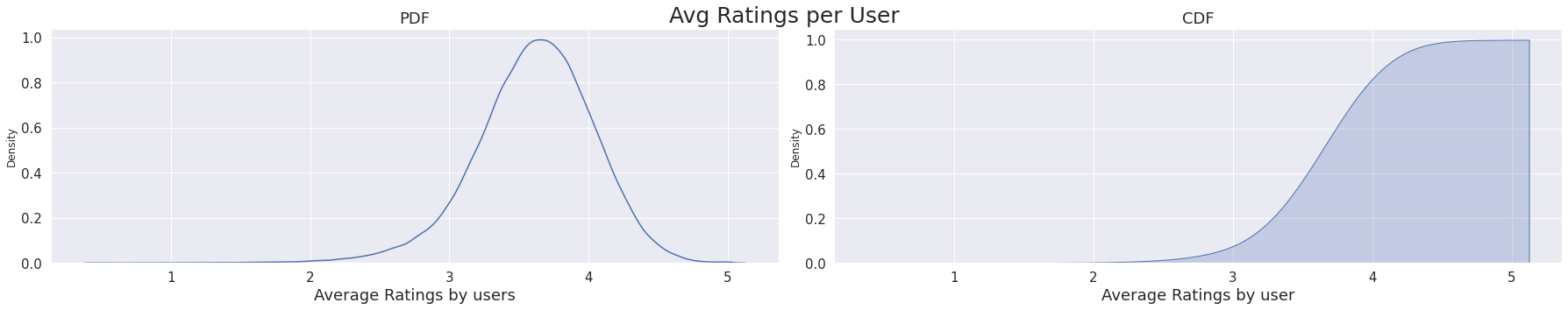
axes[1].set\_xlabel("Average Ratings by user", fontsize = 18)

axes[1].tick\_params(labelsize = 15)

fig.subplots\_adjust(wspace=2)

plt.tight\_layout()

plt.show()



Observations:

1. User ID 125794 has the highest avg. rating of 3.81.
2. The pdf of average ratings given by a user seems to be a bit left skewed, with most of the values centered around 3.5 to 4.
3. THe cdf also shows that avg. ratings is most frequent in between 3 to 5.

**5. Analyzing the Ratings given to the Movies.**

In [ ]:

*# Calculating count of ratings received for movies*

no\_of\_ratings\_per\_movie = Train\_Data.groupby(by = ["movieId", "title"], as\_index=False)["rating"].count().sort\_values(by=["rating"], ascending = False)

no\_of\_ratings\_per\_movie = no\_of\_ratings\_per\_movie.reset\_index(drop=True)

In [ ]:

*# Visualizing the number of ratings for the movies*

sns.set(style="darkgrid")

fig = plt.figure(figsize = (25, 5))

axes = fig.add\_axes([0.1, 0.1, 1, 1])

plt.title("Number of Ratings Per Movie", fontsize = 20)

plt.xlabel("Movie", fontsize = 15)

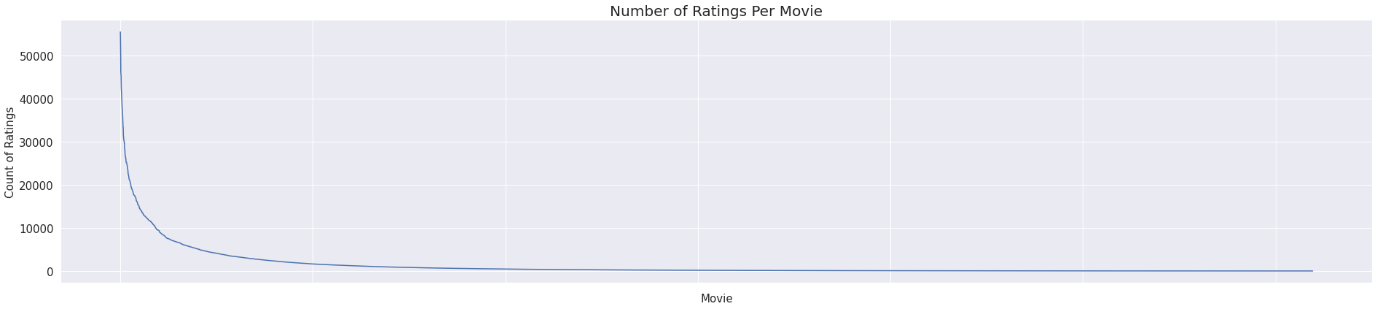
plt.ylabel("Count of Ratings", fontsize = 15)

plt.plot(no\_of\_ratings\_per\_movie["rating"].values)

plt.tick\_params(labelsize = 15)

axes.set\_xticklabels([])

plt.show()



Observations:

1. It is quite clear that there are some movies which are very popular and were rated by many users as comapared to other movies which has caused the plot to be skewed.

In [ ]:

*# Visualizing top 5 movies heavily rated movies.*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.barplot(x="title", y="rating", data=no\_of\_ratings\_per\_movie[:15], ax=axes)

for p in axes.patches:

axes.annotate('{}'.format(int(p.get\_height())), (p.get\_x(), p.get\_height()+100))

axes.set\_yticklabels([ChangingLabelsInK(num) for num in axes.get\_yticks()])

plt.tick\_params(labelsize = 15)

plt.title("Number of ratings for Top 15 Movies", fontsize = 20)

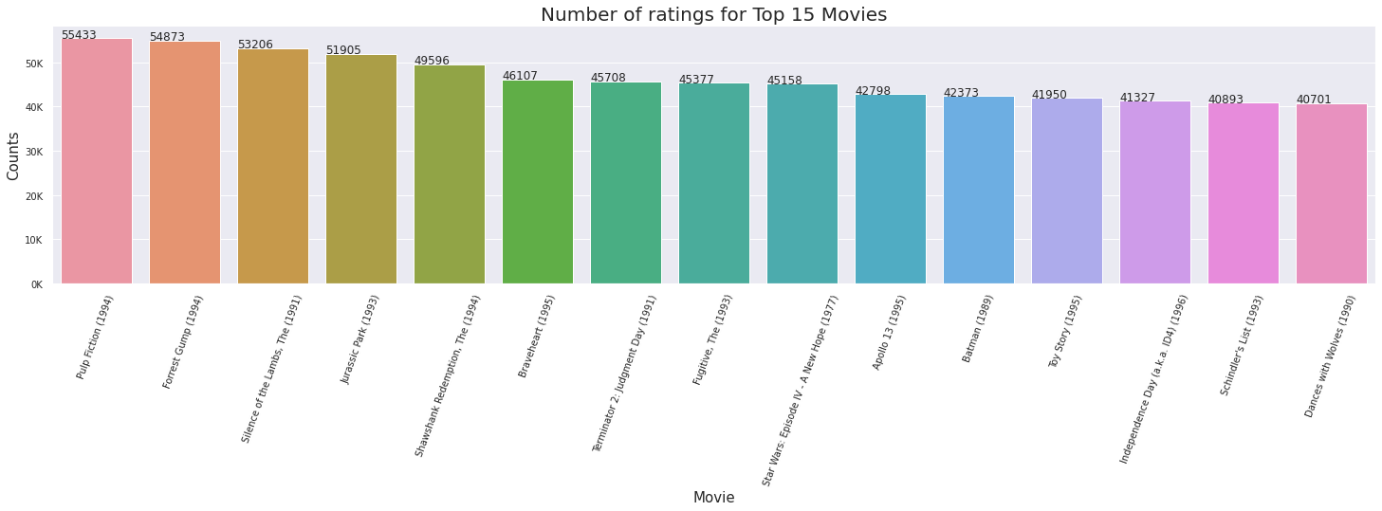
plt.xlabel("Movie", fontsize = 15)

plt.xticks(rotation=70, fontsize=10)

plt.ylabel("Counts", fontsize = 15)

plt.yticks(fontsize=10)

plt.show()



In [ ]:

*# Calculating average ratings for movies*

avg\_ratings\_per\_movie = Train\_Data.groupby(by = ["movieId", "title"], as\_index=False)["rating"].mean()

avg\_ratings\_per\_movie = avg\_ratings\_per\_movie.reset\_index(drop=True)

avg\_ratings\_per\_movie = avg\_ratings\_per\_movie.merge(no\_of\_ratings\_per\_movie[["movieId", "rating"]], how="left", on="movieId")

avg\_ratings\_per\_movie.rename(columns={"rating\_x":"avg\_rating", "rating\_y": "num\_of\_rating"}, inplace=True)

avg\_ratings\_per\_movie = avg\_ratings\_per\_movie.sort\_values("num\_of\_rating", ascending=False)

In [ ]:

*# Visualizing the average ratings by individual Days of the Week*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.barplot(x="title", y="avg\_rating", data=avg\_ratings\_per\_movie[:15], ax=axes)

for p in axes.patches:

axes.annotate('{}'.format(round(p.get\_height(), 2)), (p.get\_x()+0.3, p.get\_height()))

plt.tick\_params(labelsize = 15)

plt.title("Average Ratings For top 15 movies", fontsize = 20)

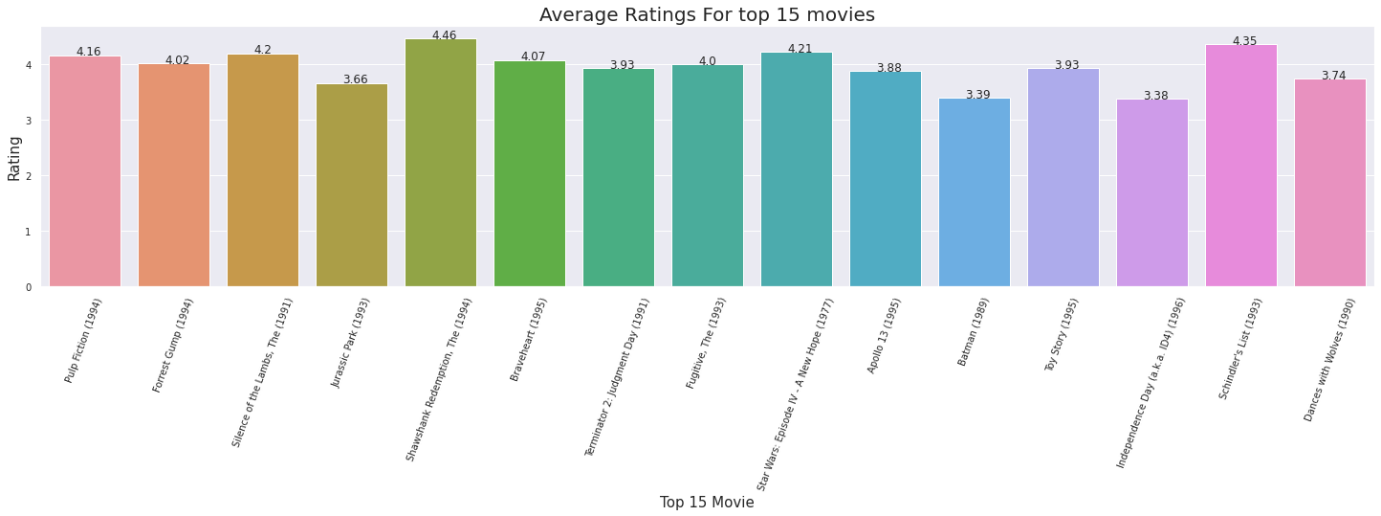
plt.xlabel("Top 15 Movie", fontsize = 15)

plt.xticks(rotation=70, fontsize=10)

plt.ylabel("Rating", fontsize = 15)

plt.yticks(fontsize=10)

plt.show()



Observations:

1. The cult movies form 1990s have been rated the most.
2. Pulp Fiction, Forrest Gump, Shawshank Redemption and Silience of the Lambs have been rated over 50K times.
3. Shawshank Redemption has the highest average rating of 4.56 based on 50K+ ratings.

In [ ]:

*# Plotting the PDF and CDF for Avg. rating by Movies*

sns.set(style="darkgrid")

fig, axes = plt.subplots(1, 2, figsize=(25, 5))

fig.suptitle("Avg Ratings per Movie", fontsize=25)

sns.distplot(avg\_ratings\_per\_movie["avg\_rating"], hist = False, ax = axes[0], label = "PDF")

axes[0].set\_title("PDF", fontsize = 18)

axes[0].set\_xlabel("Average Ratings by Movie", fontsize = 18)

axes[0].tick\_params(labelsize = 15)

sns.kdeplot(avg\_ratings\_per\_movie["avg\_rating"], cumulative = True, ax = axes[1], shade=True, label = "CDF")

axes[1].set\_title("CDF", fontsize = 18)

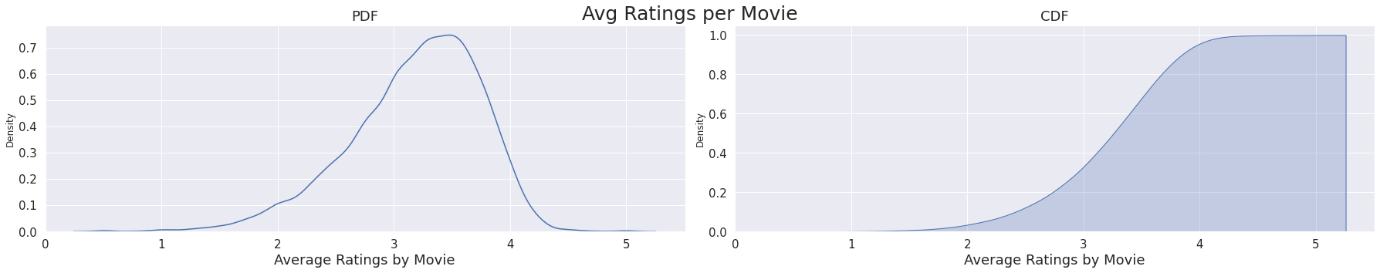
axes[1].set\_xlabel("Average Ratings by Movie", fontsize = 18)

axes[1].tick\_params(labelsize = 15)

fig.subplots\_adjust(wspace=2)

plt.tight\_layout()

plt.show()



Observations:

1. The distribution of average rating for movie is fairly normal one.
2. The cdf shows that the avg. rating is more frequent after 3.

In [ ]:

Train\_Data.head()

Out[ ]:

|  | **userId** | **movieId** | **rating** | **date** | **title** | **genres** | **DayOfWeek** | **Weekday** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 131160 | 1058 | 3.0 | 1995-01-09 | Fish Called Wanda, A (1988) | Comedy|Crime | Monday | 0 |
| **1** | 131160 | 47 | 5.0 | 1995-01-09 | Seven (a.k.a. Se7en) (1995) | Mystery|Thriller | Monday | 0 |
| **2** | 28507 | 1154 | 4.0 | 1995-01-09 | Double Life of Veronique, The (Double Vie de V... | Drama|Fantasy|Romance | Monday | 0 |
| **3** | 131160 | 21 | 3.0 | 1995-01-09 | Get Shorty (1995) | Comedy|Crime|Thriller | Monday | 0 |
| **4** | 85252 | 7 | 5.0 | 1996-01-29 | Sabrina (1995) | Comedy|Romance | Monday | 0 |

**3.3. Feature Engineering**

Now that we have completed the data exploration part, we can start the Feature Engineering in order to prepare the data for the ML algorithms.

**3.3.1 Creating Matrices**

We will be creating matrices like: User-Item matrix, User-User and Item-Item similarity matrix.

**1. Creating USER-ITEM sparse matrix.**

In [ ]:

*# Path for loading/saving files*

file\_path = "/content/drive/MyDrive/Colab Datasets/Movie Recommendation"

In [ ]:

*# Creating/loading user-movie sparse matrix for train data*

startTime = datetime.now()

print("Creating USER\_ITEM sparse matrix for train Data..")

if os.path.isfile(file\_path + "/TrainUISparseData.npz"):

print("Sparse Data is already present in your disk, no need to create further. Loading Sparse Matrix")

TrainUISparseData = sparse.load\_npz(file\_path + "/TrainUISparseData.npz")

print("Shape of Train Sparse matrix = "+str(TrainUISparseData.shape))

else:

print("We are creating sparse data..")

TrainUISparseData = sparse.csr\_matrix((Train\_Data.rating, (Train\_Data.userId, Train\_Data.movieId)))

print("Creation done. Shape of sparse matrix : ", str(TrainUISparseData.shape))

print("Saving it into disk for furthur usage.")

sparse.save\_npz(file\_path + "/TrainUISparseData.npz", TrainUISparseData)

print("Done\n")

print("Time taken : ", 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 = (138494, 14334)

Time taken : 0:00:01.014576

In [ ]:

rows,cols = TrainUISparseData.shape

presentElements = TrainUISparseData.count\_nonzero()

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

Sparsity Of Train matrix : 99.19401432357586%

In [ ]:

*# Creating/loading user-movie sparse matrix for test data*

startTime = datetime.now()

print("Creating USER\_ITEM sparse matrix for test Data..")

if os.path.isfile(file\_path + "/TestUISparseData.npz"):

print("Sparse Data is already present in your disk, no need to create further. Loading Sparse Matrix")

TestUISparseData = sparse.load\_npz(file\_path + "/TestUISparseData.npz")

print("Shape of Test Sparse Matrix : ", str(TestUISparseData.shape))

else:

print("We are creating sparse data..")

TestUISparseData = sparse.csr\_matrix((Test\_Data.rating, (Test\_Data.userId, Test\_Data.movieId)))

print("Creation done. Shape of sparse matrix : ", str(TestUISparseData.shape))

print("Saving it into disk for furthur usage.")

sparse.save\_npz(file\_path + "/TestUISparseData.npz", TestUISparseData)

print("Done\n")

print("Time Taken : ", 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 : (138494, 27279)

Time Taken : 0:00:00.271382

In [ ]:

rows,cols = TestUISparseData.shape

presentElements = TestUISparseData.count\_nonzero()

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

Sparsity Of Test matrix : 99.89412185078294%

Observations:

1. Shape of sparse matrix depends on highest value of userId and highest value of movieId.
2. In the test set, there are few users from train set which is not present in the test set.
3. For the movieId, there are less movies in the train set as compared to test set. The reason stems from the fact that we split the data based on time, and newer movies would have fallen into test set.
4. The sparsity of train set is 99.19% while the sparsity of test set is 99.89%.

In [ ]:

*# Function to Calculate Average rating for users or movies from User-movie sparse matrix*

def getAverageRatings(sparseMatrix, if\_user):

*#axis = 1 means rows and axis = 0 means columns*

ax = 1 if if\_user else 0

sumOfRatings = sparseMatrix.sum(axis = ax).A1

noOfRatings = (sparseMatrix!=0).sum(axis = ax).A1

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

In [ ]:

AvgRatingUser = getAverageRatings(TrainUISparseData, True)

AvgRatingMovie = getAverageRatings(TrainUISparseData, False)

train\_users = len(AvgRatingUser)

uncommonUsers = total\_users - train\_users

print("Total no. of Users : ", total\_users)

print("No. of Users in Train data : ", train\_users)

print("No. of Users not present in Train data : {}({}%)".format(uncommonUsers, np.round((uncommonUsers/total\_users)\*100), 2))

Total no. of Users : 138493

No. of Users in Train data : 112466

No. of Users not present in Train data : 26027(19.0%)

In [ ]:

train\_movies = len(AvgRatingMovie)

uncommonMovies = total\_movies - train\_movies

print("Total no. of Movies : ", total\_movies)

print("No. of Movies in Train data : ", train\_movies)

print("No. of Movies not present in Train data = {}({}%)".format(uncommonMovies, np.round((uncommonMovies/total\_movies)\*100), 2))

Total no. of Movies : 26744

No. of Movies in Train data : 12387

No. of Movies not present in Train data = 14357(54.0%)

Observations:

1. Recommendation System suffers from Cold Start problems, which needs to be tackled wisely in order to design a effective system.
2. There are 26027, ie 19% of the users are not present in the training data.
3. There are 12387, ie 54% of the movies which are not present in the training data.

**2. Creating Similarity Matrix**

Computation of user-user or item-item similarity matrix is impossible if computational power is limited given we have a user vector of size 112K. There will be a matrix of size 14K x 14K.

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.

For the workaround, we will 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 ).

The datastructure to be used 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>

**2.1. Computing Item-Item Similarity Matrix**

In [ ]:

*# Computing user-user similarity matrix for the train data*

*# We have 138K sized sparse vectors using which a 14K x 14K movie similarity matrix would be calculated*

start = datetime.now()

if not os.path.isfile(file\_path + "/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("Dimension of Matrix : ", m\_m\_similarity.shape)

print("Storing the Movie Similarity matrix on disk for further usage")

sparse.save\_npz(file\_path + "/m\_m\_similarity.npz", m\_m\_similarity)

else:

print("File exists in the disk. Loading the file...")

m\_m\_similarity = sparse.load\_npz(file\_path + "/m\_m\_similarity.npz")

print("Dimension of Matrix : ", m\_m\_similarity.shape)

print("The time taken to compute movie-movie similarity matrix is : ", datetime.now() - start)

File exists in the disk. Loading the file...

Dimension of Matrix : (14334, 14334)

The time taken to compute movie-movie similarity matrix is : 0:00:21.091799

In [ ]:

*# Creating a function to take Movie Name and generate the top matched name and generate its N similar movies based on M-M Similary*

def GetSimilarMoviesUsingMovieMovieSimilarity(movie\_name, num\_of\_similar\_movies):

matches = process.extract(movie\_name, movie\_list\_in\_training["title"], scorer=fuzz.partial\_ratio)

if len(matches) == 0:

return "No Match Found"

movie\_id = movie\_list\_in\_training.iloc[matches[0][2]]["movieId"]

similar\_movie\_id\_list = np.argsort(-m\_m\_similarity[movie\_id].toarray().ravel())[0:num\_of\_similar\_movies+1]

sm\_df = movie\_list\_in\_training[movie\_list\_in\_training["movieId"].isin(similar\_movie\_id\_list)]

sm\_df["order"] = sm\_df.apply(lambda x: list(similar\_movie\_id\_list).index(x["movieId"]), axis=1)

return sm\_df.sort\_values("order")

In [ ]:

*# Picking random movie and checking it's top 10 most similar movies*

GetSimilarMoviesUsingMovieMovieSimilarity("Superman", 10)

Out[ ]:

|  | **movieId** | **title** | **genres** | **order** |
| --- | --- | --- | --- | --- |
| **2526** | 2557 | Superman II (1980) | Action|Sci-Fi | 0 |
| **2530** | 2558 | Superman III (1983) | Action|Adventure|Sci-Fi | 1 |
| **2527** | 2556 | Superman (1978) | Action|Adventure|Sci-Fi | 2 |
| **2531** | 2559 | Superman IV: The Quest for Peace (1987) | Action|Adventure|Sci-Fi | 3 |
| **2877** | 2900 | RoboCop (1987) | Action|Crime|Drama|Sci-Fi|Thriller | 4 |
| **1340** | 1345 | Star Trek III: The Search for Spock (1984) | Action|Adventure|Sci-Fi | 5 |
| **2012** | 2022 | Tron (1982) | Action|Adventure|Sci-Fi | 6 |
| **1339** | 1344 | Star Trek II: The Wrath of Khan (1982) | Action|Adventure|Sci-Fi|Thriller | 7 |
| **1341** | 1346 | Star Trek IV: The Voyage Home (1986) | Adventure|Comedy|Sci-Fi | 8 |
| **2807** | 2831 | Total Recall (1990) | Action|Adventure|Sci-Fi|Thriller | 9 |
| **1345** | 1341 | Star Trek: The Motion Picture (1979) | Adventure|Sci-Fi | 10 |

**2.2. Computing User-User Similarity Matrix.**

In [ ]:

*# Getting highest uder id*

row\_index, col\_index = TrainUISparseData.nonzero()

unique\_user\_id = np.unique(row\_index)

print("Max User id is :", np.max(unique\_user\_id))

Max User id is : 138493

In [ ]:

*# Here, we are calculating user-user similarity matrix only for first 100 users in our sparse matrix. And we are calculating*

*# Top 100 most similar users with them.*

def getUser\_UserSimilarity(sparseMatrix, top = 100):

startTimestamp20 = datetime.now()

row\_index, col\_index = sparseMatrix.nonzero()

rows = np.unique(row\_index)

similarMatrix = np.zeros(13849300).reshape(138493,100) *# 138493\*100 = 13849300. 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")

sns.set(style="darkgrid")

fig = plt.figure(figsize = (25, 5))

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

simMatrix = getUser\_UserSimilarity(TrainUISparseData, 100)

Time elapsed for 20 users = 0:00:11.146697sec

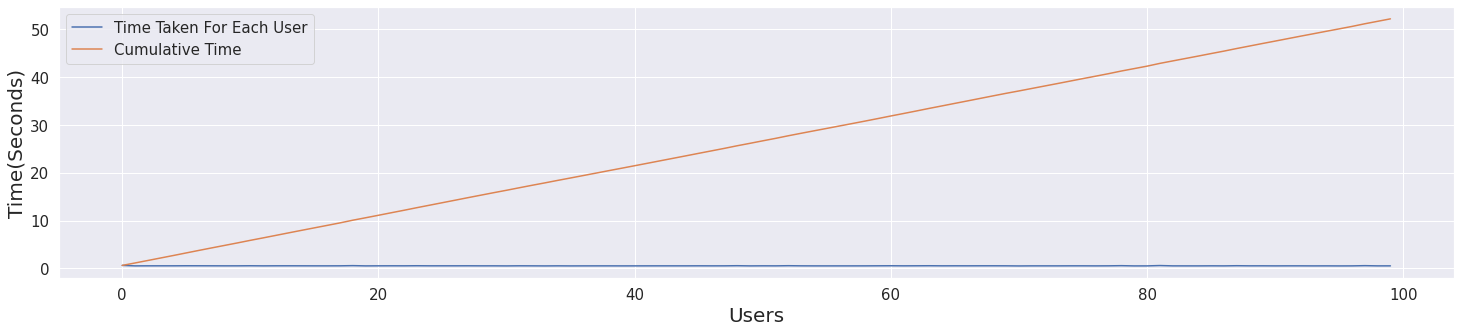
Time elapsed for 40 users = 0:00:21.517260sec

Time elapsed for 60 users = 0:00:31.898194sec

Time elapsed for 80 users = 0:00:42.343119sec

Time elapsed for 100 users = 0:00:52.745646sec

Average Time taken to compute similarity matrix for 1 user = 0.5216764903068543seconds



In [ ]:

*# Calculating user-user similarity only for particular users in our sparse matrix and return user\_ids*

def Calculate\_User\_User\_Similarity(sparseMatrix, user\_id, num\_of\_similar\_users=10):

if user\_id in unique\_user\_id:

*# Calculating the cosine similarity for user\_id with all the "userId"*

sim = cosine\_similarity(sparseMatrix.getrow(user\_id), sparseMatrix).ravel()

*# Sorting the indexs(user\_id) based on the similarity score for all the user ids*

top\_similar\_user\_ids = sim.argsort()[::-1]

*# Sorted the similarity values*

top\_similarity\_values = sim[top\_similar\_user\_ids]

return top\_similar\_user\_ids[1: num\_of\_similar\_users+1]

In [ ]:

*# Getting top 5 users similar to userId: 1*

similar\_users\_1 = Calculate\_User\_User\_Similarity(TrainUISparseData, 1, 5)

similar\_users\_1

Out[ ]:

array([81275, 62235, 2595, 75328, 34101])

**3.3.2 Feature Extraction**

Now we can start extracting meaningful features in order to prepare the data for ML algorithms.

In [ ]:

*# Path for saving/loading files*

file\_path = "/content/drive/MyDrive/Colab Datasets/Movie Recommendation"

In [ ]:

*# Since the given dataset might not completely fit into computaton capacity that we have, we will sample the data and work it*

*# Function for Sampling random movies and users to reduce the size of rating matrix*

def get\_sample\_sparse\_matrix(sparseMatrix, n\_users, n\_movies, matrix\_name):

np.random.seed(15) *#this will give same random number everytime, without replacement*

startTime = datetime.now()

users, movies, ratings = sparse.find(sparseMatrix)

uniq\_users = np.unique(users)

uniq\_movies = np.unique(movies)

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(file\_path + "/" + matrix\_name, sparse\_sample)

print("Shape of Sparse Sampled Matrix = " + str(sparse\_sample.shape))

print("Time taken : ", datetime.now() - startTime)

return sparse\_sample

In [ ]:

*# Creating Sample Sparse Matrix for Train Data*

if not os.path.isfile(file\_path + "/TrainUISparseData\_Sample.npz"):

print("Sample sparse matrix is not present in the disk. We are creating it...")

train\_sample\_sparse = get\_sample\_sparse\_matrix(TrainUISparseData, 5000, 1000, "TrainUISparseData\_Sample.npz")

else:

print("File is already present in the disk. Loading the file...")

train\_sample\_sparse = sparse.load\_npz(file\_path + "/TrainUISparseData\_Sample.npz")

print("Shape of Train Sample Sparse Matrix = " + str(train\_sample\_sparse.shape))

File is already present in the disk. Loading the file...

Shape of Train Sample Sparse Matrix = (138488, 14312)

In [ ]:

*# Creating Sample Sparse Matrix for Test Data*

if not os.path.isfile(file\_path + "/TestUISparseData\_Sample.npz"):

print("Sample sparse matrix is not present in the disk. We are creating it...")

test\_sample\_sparse = get\_sample\_sparse\_matrix(TestUISparseData, 2000, 200, "TestUISparseData\_Sample.npz")

else:

print("File is already present in the disk. Loading the file...")

test\_sample\_sparse = sparse.load\_npz(file\_path + "/TestUISparseData\_Sample.npz")

print("Shape of Test Sample Sparse Matrix = " + str(test\_sample\_sparse.shape))

File is already present in the disk. Loading the file...

Shape of Test Sample Sparse Matrix = (138456, 27198)

In [ ]:

*# Checking the shape of Training and test data*

print("Shape of Train Sparse Matrix : ", train\_sample\_sparse.shape)

print("Shape of Test Sparse Matrix : ", test\_sample\_sparse.shape)

Shape of Train Sparse Matrix : (138488, 14312)

Shape of Test Sparse Matrix : (138456, 27198)

In [ ]:

*# Calculating few GlobalAverageRating, AvgMovieRating, AvgUserRating and TotalNoOfRatings*

globalAvgRating = np.round((train\_sample\_sparse.sum()/train\_sample\_sparse.count\_nonzero()), 2)

globalAvgMovies = getAverageRatings(train\_sample\_sparse, False)

globalAvgUsers = getAverageRatings(train\_sample\_sparse, True)

print("Global average of all movies ratings in Train Set is : ", globalAvgRating)

print("No. of ratings in the train matrix is : ", train\_sample\_sparse.count\_nonzero())

Global average of all movies ratings in Train Set is : 3.5

No. of ratings in the train matrix is : 54216

In [ ]:

*# Function to extract features and create row using the sparse matrix*

def CreateFeaturesForTrainData(SampledSparseData, TrainSampledSparseData):

startTime = datetime.now()

*# Extracting userId list, movieId list and Ratings*

sample\_users, sample\_movies, sample\_ratings = sparse.find(SampledSparseData)

print("No. of rows in the returned dataset : ", len(sample\_ratings))

count = 0

data = []

for user, movie, rating in zip(sample\_users, sample\_movies, sample\_ratings):

row = list()

*#----------------------------------Appending "user Id" average, "movie Id" average & global average rating-----------#*

row.append(user)

row.append(movie)

row.append(globalAvgRating)

*#----------------------------------Appending "user" average, "movie" average & rating of "user""movie"-----------#*

try:

row.append(globalAvgUsers[user])

except (KeyError):

global\_average\_rating = globalAvgRating

row.append(global\_average\_rating)

except:

raise

try:

row.append(globalAvgMovies[movie])

except (KeyError):

global\_average\_rating = globalAvgRating

row.append(global\_average\_rating)

except:

raise

*#----------------------------------Ratings given to "movie" by top 5 similar users with "user"--------------------#*

try:

similar\_users = cosine\_similarity(TrainSampledSparseData[user], TrainSampledSparseData).ravel()

similar\_users\_indices = np.argsort(-similar\_users)[1:]

similar\_users\_ratings = TrainSampledSparseData[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" 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\_rating = [globalAvgRating]\*5

row.extend(global\_average\_rating)

except:

raise

*#----------------------------------Ratings given by "user" to top 5 similar movies with "movie"------------------#*

try:

similar\_movies = cosine\_similarity(TrainSampledSparseData[:,movie].T, TrainSampledSparseData.T).ravel()

similar\_movies\_indices = np.argsort(-similar\_movies)[1:]

similar\_movies\_ratings = TrainSampledSparseData[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" 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\_rating = [globalAvgRating] \* 5

row.extend(global\_average\_rating)

except:

raise

*#----------------------------------Appending rating of "user""movie"-----------#*

row.append(rating)

count += 1

data.append(row)

if count % 5000 == 0:

print("Done for {}. Time elapsed: {}".format(count, (datetime.now() - startTime)))

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

print("Completed..")

return data

In [ ]:

*# Using sampled train data, creating Features for each row and saving it into the list*

data\_rows = CreateFeaturesForTrainData(train\_sample\_sparse, train\_sample\_sparse)

Preparing Train csv file for 54216 rows

Done for 5000. Time elapsed: 0:09:52.844203

Done for 10000. Time elapsed: 0:19:59.049138

Done for 15000. Time elapsed: 0:30:01.427127

Done for 20000. Time elapsed: 0:39:49.924850

Done for 25000. Time elapsed: 0:49:35.816057

Done for 30000. Time elapsed: 0:59:04.743915

Done for 35000. Time elapsed: 1:08:25.503198

Done for 40000. Time elapsed: 1:17:37.023778

Done for 45000. Time elapsed: 1:26:37.648877

Done for 50000. Time elapsed: 1:35:35.413135

Total Time for 54216 rows = 1:43:01.632676

In [ ]:

*# Using sampled train data, creating Features for each row and saving it into the list*

test\_data\_rows = CreateFeaturesForTrainData(test\_sample\_sparse, train\_sample\_sparse)

No. of rows in the returned dataset : 2084

Total Time for 2084 rows = 0:00:26.759629

Completed..

In [ ]:

*# Creating the pandas dataframe from the data rows extracted from the sparse matrix for train and test set*

names = ["User\_ID", "Movie\_ID", "Global\_Average", "User\_Average", "Movie\_Average", "SUR1", "SUR2", "SUR3", "SUR4", "SUR5", "SMR1", "SMR2", "SMR3", "SMR4", "SMR5", "Rating"]

train\_regression\_data = pd.DataFrame(data\_rows, columns=names)

test\_regression\_data = pd.DataFrame(test\_data\_rows, columns=names)

In [ ]:

*# Saving the df to drive for future use*

train\_regression\_data.to\_csv(file\_path + "/Training\_Data\_For\_Regression.csv")

test\_regression\_data.to\_csv(file\_path + "/Testing\_Data\_For\_Regression.csv")

In [ ]:

*# Loading the train and test csv files*

*# Path for saving/loading files*

file\_path = "/content/drive/MyDrive/Colab Datasets/Movie Recommendation"

print("File is already present in the disk. Loading the file...")

train\_regression\_data = pd.read\_csv(file\_path + "/Training\_Data\_For\_Regression.csv")

train\_regression\_data = train\_regression\_data.drop(["Unnamed: 0"], axis=1)

test\_regression\_data = pd.read\_csv(file\_path + "/Testing\_Data\_For\_Regression.csv")

test\_regression\_data = test\_regression\_data.drop(["Unnamed: 0"], axis=1)

print("Done..")

File is already present in the disk. Loading the file...

Done..

In [ ]:

*# Checking the shape and first few records for train data*

print("The shape of the dataframe is : ", train\_regression\_data.shape)

print("Number of missing Values : ", train\_regression\_data.isnull().sum().sum())

train\_regression\_data.head()

The shape of the dataframe is : (54216, 16)

Number of missing Values : 0

Out[ ]:

|  | **User\_ID** | **Movie\_ID** | **Global\_Average** | **User\_Average** | **Movie\_Average** | **SUR1** | **SUR2** | **SUR3** | **SUR4** | **SUR5** | **SMR1** | **SMR2** | **SMR3** | **SMR4** | **SMR5** | **Rating** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 332 | 25 | 3.5 | 3.142857 | 3.70987 | 4.0 | 5.0 | 3.0 | 3.0 | 5.0 | 5.0 | 3.0 | 2.0 | 3.0 | 2.0 | 5.0 |
| **1** | 463 | 25 | 3.5 | 3.875000 | 3.70987 | 5.0 | 5.0 | 4.0 | 4.0 | 5.0 | 4.0 | 4.0 | 4.0 | 3.0 | 4.0 | 5.0 |
| **2** | 484 | 25 | 3.5 | 5.000000 | 3.70987 | 5.0 | 5.0 | 3.0 | 3.0 | 5.0 | 3.0 | 4.0 | 5.0 | 5.0 | 4.0 | 5.0 |
| **3** | 592 | 25 | 3.5 | 5.000000 | 3.70987 | 5.0 | 4.0 | 5.0 | 3.0 | 4.0 | 5.0 | 4.0 | 3.0 | 3.0 | 3.0 | 5.0 |
| **4** | 640 | 25 | 3.5 | 3.166667 | 3.70987 | 3.0 | 3.0 | 5.0 | 4.0 | 5.0 | 3.0 | 5.0 | 3.0 | 3.0 | 3.0 | 2.0 |

In [ ]:

*# Checking the shape and first few records for test data*

print("The shape of the dataframe is : ", test\_regression\_data.shape)

print("Number of missing Values : ", test\_regression\_data.isnull().sum().sum())

test\_regression\_data.head()

The shape of the dataframe is : (2084, 16)

Number of missing Values : 0

Out[ ]:

|  | **User\_ID** | **Movie\_ID** | **Global\_Average** | **User\_Average** | **Movie\_Average** | **SUR1** | **SUR2** | **SUR3** | **SUR4** | **SUR5** | **SMR1** | **SMR2** | **SMR3** | **SMR4** | **SMR5** | **Rating** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 24024 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 |
| **1** | 89299 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 |
| **2** | 89495 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 |
| **3** | 105227 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 4.0 |
| **4** | 500 | 605 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 |

Observations:

The description of the features are stated below:

1. User\_ID: ID of a this User
2. Movie\_ID: ID of a this Movie
3. Global\_Average: Global Average Rating
4. User\_Average: Average Rating of this User
5. Movie\_Average: Average Rating of this Movie
6. Ratings given to this Movie by top 5 similar users with this User: (SUR1, SUR2, SUR3, SUR4, SUR5)
7. Ratings given by this User to top 5 similar movies with this Movie: (SMR1, SMR2, SMR3, SMR4, SMR5)
8. Rating: Rating given by this User to this Movie

**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, KNN, BaseLineOnly, 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>

Transforming Test Data:

For test data we just have to define a tuple (user, item, rating). Check out this link: <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 [ ]:

train\_regression\_data[['User\_ID', 'Movie\_ID', 'Rating']].head(5)

Out[ ]:

|  | **User\_ID** | **Movie\_ID** | **Rating** |
| --- | --- | --- | --- |
| **0** | 332 | 25 | 5.0 |
| **1** | 463 | 25 | 5.0 |
| **2** | 484 | 25 | 5.0 |
| **3** | 592 | 25 | 5.0 |
| **4** | 640 | 25 | 2.0 |

In [ ]:

*# Using Surprise library Data Structures to store train data*

reader = Reader(rating\_scale=(1, 5))

data = Dataset.load\_from\_df(train\_regression\_data[["User\_ID", "Movie\_ID", "Rating"]], reader)

trainset = data.build\_full\_trainset()

In [ ]:

*# Creating tuple for test set*

testset = list(zip(test\_regression\_data["User\_ID"].values, test\_regression\_data["Movie\_ID"].values, test\_regression\_data["Rating"].values))

**4. Model Buliding**

We will try to build a regression model to predict the rating given by an user to a movie based on the generated fetures.

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.

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.

where At is the actual value and Ft is the forecast value.

In [ ]:

*# Utilities to save the modelling results*

error\_cols = ["Model", "Train RMSE", "Train MAPE", "Test RMSE", "Test MAPE"]

error\_table = pd.DataFrame(columns = error\_cols)

model\_train\_evaluation = dict()

model\_test\_evaluation = dict()

In [ ]:

*# Function to save modelling results in a table*

def make\_table(model\_name, rmse\_train, mape\_train, rmse\_test, mape\_test):

global error\_table

error\_table = error\_table.append(pd.DataFrame([[model\_name, rmse\_train, mape\_train, rmse\_test, mape\_test]], columns = error\_cols))

error\_table.reset\_index(drop = True, inplace = True)

In [ ]:

*# Function to calulate RMSE and MAPE values*

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 [ ]:

*# Apply Xgboost Regressor on the Train and Test Data*

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))

train\_result = {"RMSE": rmse\_train, "MAPE": mape\_train, "Prediction": y\_pred\_train}

print("-" \* 50)

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))

test\_result = {"RMSE": rmse\_test, "MAPE": mape\_test, "Prediction": y\_pred\_test}

print("-"\*50)

print("Time Taken : ", datetime.now() - startTime)

plot\_importance(xgb, clf)

make\_table(model\_name, rmse\_train, mape\_train, rmse\_test, mape\_test)

return train\_result, test\_result

*# Function to plot feature importance for a model*

def plot\_importance(model, clf):

sns.set(style="darkgrid")

fig = plt.figure(figsize = (25, 5))

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()

In [ ]:

*# 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*

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

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 [ ]:

my\_seed = 15

random.seed(my\_seed)

np.random.seed(my\_seed)

*# Running Surprise model algorithms*

def run\_surprise(algo, trainset, testset, model\_name):

startTime = datetime.now()

train = dict()

test = dict()

algo.fit(trainset)

*#-----------------Evaluating Train Data------------------#*

print("-"\*50)

print("TRAIN DATA")

train\_pred = algo.test(trainset.build\_testset())

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))

train = {"RMSE": train\_rmse, "MAPE": train\_mape, "Prediction": train\_predicted}

*#-----------------Evaluating Test Data------------------#*

print("-"\*50)

print("TEST DATA")

test\_pred = algo.test(testset)

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))

test = {"RMSE": test\_rmse, "MAPE": test\_mape, "Prediction": test\_predicted}

print("-"\*50)

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

make\_table(model\_name, train\_rmse, train\_mape, test\_rmse, test\_mape)

return train, test

**4.1 Train/test Splitting**

We can split the data for train/test and segregate the independent and dependent features.

In [ ]:

*# Creating the train-test X and y variables for the ML algos*

x\_train = train\_regression\_data.drop(["User\_ID", "Movie\_ID", "Rating"], axis = 1)

x\_test = test\_regression\_data.drop(["User\_ID", "Movie\_ID", "Rating"], axis = 1)

y\_train = train\_regression\_data["Rating"]

y\_test = test\_regression\_data["Rating"]

**4.2 Model Fitting**

Fitting various models and checking its accuracy.

In [ ]:

*# Training the Xgboost Regression Model on with the 13 features*

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

[12:54:17] WARNING: /workspace/src/objective/regression\_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.

--------------------------------------------------

TRAIN DATA

RMSE : 0.807227677568619

MAPE : 26.29768313505046

--------------------------------------------------

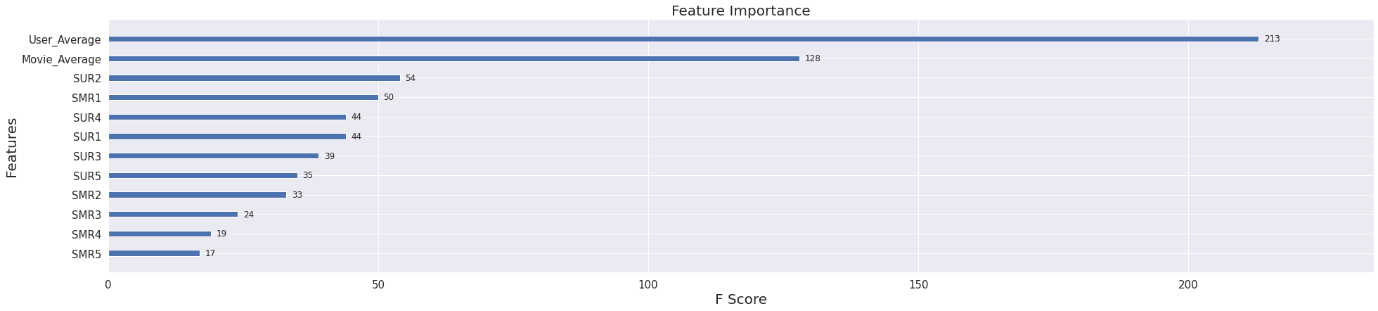
TEST DATA

RMSE : 1.0048489822504847

MAPE : 36.48806593597417

--------------------------------------------------

Time Taken : 0:00:02.881521



Observations:

1. "User\_Average" by far seems to be the most important feature for rating prediction.
2. "Movie\_Average" is the second most important feature to predict the ratings.
3. The top 5 Similar User ratings and top 5 Similar Movie Ratings doesn't seems to be the effective features.

In [ ]:

*# Applying BaselineOnly from the surprise library to predict the ratings*

bsl\_options = {"method":"sgd", "learning\_rate":0.01, "n\_epochs":25}

algo = BaselineOnly(bsl\_options=bsl\_options)

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 = 0.8273711353440799

MAPE = 27.61018921790378

--------------------------------------------------

TEST DATA

RMSE = 0.9995812383564545

MAPE = 36.46762303737812

--------------------------------------------------

Time Taken = 0:00:00.929904

In [ ]:

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

train\_regression\_data["BaselineOnly"] = model\_train\_evaluation["BaselineOnly"]["Prediction"]

test\_regression\_data["BaselineOnly"] = model\_test\_evaluation["BaselineOnly"]["Prediction"]

In [ ]:

train\_regression\_data.head()

Out[ ]:

|  | **User\_ID** | **Movie\_ID** | **Global\_Average** | **User\_Average** | **Movie\_Average** | **SUR1** | **SUR2** | **SUR3** | **SUR4** | **SUR5** | **SMR1** | **SMR2** | **SMR3** | **SMR4** | **SMR5** | **Rating** | **BaselineOnly** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 332 | 25 | 3.5 | 3.142857 | 3.70987 | 4.0 | 5.0 | 3.0 | 3.0 | 5.0 | 5.0 | 3.0 | 2.0 | 3.0 | 2.0 | 5.0 | 3.545173 |
| **1** | 463 | 25 | 3.5 | 3.875000 | 3.70987 | 5.0 | 5.0 | 4.0 | 4.0 | 5.0 | 4.0 | 4.0 | 4.0 | 3.0 | 4.0 | 5.0 | 2.667714 |
| **2** | 484 | 25 | 3.5 | 5.000000 | 3.70987 | 5.0 | 5.0 | 3.0 | 3.0 | 5.0 | 3.0 | 4.0 | 5.0 | 5.0 | 4.0 | 5.0 | 2.806045 |
| **3** | 592 | 25 | 3.5 | 5.000000 | 3.70987 | 5.0 | 4.0 | 5.0 | 3.0 | 4.0 | 5.0 | 4.0 | 3.0 | 3.0 | 3.0 | 5.0 | 4.296783 |
| **4** | 640 | 25 | 3.5 | 3.166667 | 3.70987 | 3.0 | 3.0 | 5.0 | 4.0 | 5.0 | 3.0 | 5.0 | 3.0 | 3.0 | 3.0 | 2.0 | 3.520443 |

In [ ]:

test\_regression\_data.head()

Out[ ]:

|  | **User\_ID** | **Movie\_ID** | **Global\_Average** | **User\_Average** | **Movie\_Average** | **SUR1** | **SUR2** | **SUR3** | **SUR4** | **SUR5** | **SMR1** | **SMR2** | **SMR3** | **SMR4** | **SMR5** | **Rating** | **BaselineOnly** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 24024 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 |
| **1** | 89299 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 |
| **2** | 89495 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 |
| **3** | 105227 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 4.0 | 3.496708 |
| **4** | 500 | 605 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 |

In [ ]:

*# Fitting the Xgboost again with new BaselineOnly feature*

x\_train = train\_regression\_data.drop(["User\_ID", "Movie\_ID", "Rating"], axis = 1)

x\_test = test\_regression\_data.drop(["User\_ID", "Movie\_ID", "Rating"], axis = 1)

y\_train = train\_regression\_data["Rating"]

y\_test = test\_regression\_data["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

[12:58:36] WARNING: /workspace/src/objective/regression\_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.

--------------------------------------------------

TRAIN DATA

RMSE : 0.807345922880538

MAPE : 26.302975814620705

--------------------------------------------------

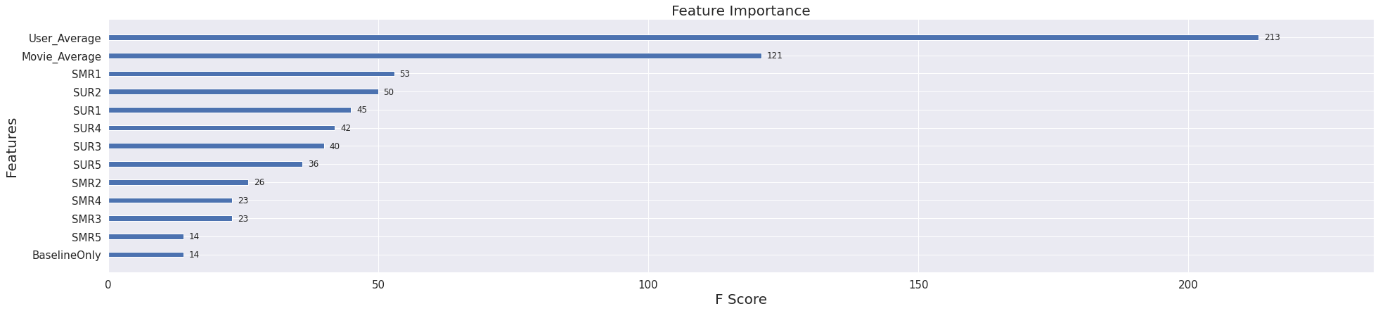
TEST DATA

RMSE : 1.0036028861472048

MAPE : 36.48211064537693

--------------------------------------------------

Time Taken : 0:00:03.011782



Observations:

1. The "BaselineOnly" feature is also not an important feature.

In [ ]:

*# Finding the suitable parameter for Surprise KNN-Baseline with User-User Similarity*

param\_grid = {'sim\_options':{'name': ["pearson\_baseline"], "user\_based": [True], "min\_support": [2], "shrinkage": [60, 80, 80, 140]}, 'k': [5, 20, 40, 80]}

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...

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...

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...

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...

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...

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...

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...

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.

0.9182634631541132

{'sim\_options': {'name': 'pearson\_baseline', 'user\_based': True, 'min\_support': 2, 'shrinkage': 60}, 'k': 80}

In [ ]:

*# Applying the KNN-Baseline with the searched parameters*

sim\_options = {'name':'pearson\_baseline', 'user\_based':True, 'min\_support':2, 'shrinkage':gs.best\_params['rmse']['sim\_options']['shrinkage']}

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 = 0.3378426389103735

MAPE = 10.313457187234661

--------------------------------------------------

TEST DATA

RMSE = 0.999480476125186

MAPE = 36.460647517968034

--------------------------------------------------

Time Taken = 0:00:32.438173

In [ ]:

*# Similarly finding best parameters for Surprise KNN-Baseline with Item-Item Similarity*

param\_grid = {'sim\_options':{'name': ["pearson\_baseline"], "user\_based": [False], "min\_support": [2], "shrinkage": [60, 80, 80, 140]}, 'k': [5, 20, 40, 80]}

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...

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...

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...

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...

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...

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...

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...

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.

0.9631564690350932

{'sim\_options': {'name': 'pearson\_baseline', 'user\_based': False, 'min\_support': 2, 'shrinkage': 60}, 'k': 40}

In [ ]:

*# Applying KNN-Baseline with best parameters searched*

sim\_options = {'name':'pearson\_baseline', 'user\_based':False, 'min\_support':2, 'shrinkage':gs.best\_params['rmse']['sim\_options']['shrinkage']}

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 = 0.27415961082608303

MAPE = 7.982776544544613

--------------------------------------------------

TEST DATA

RMSE = 0.999480476125186

MAPE = 36.460647517968034

--------------------------------------------------

Time Taken = 0:00:04.240161

In [ ]:

*# Addding the KNNBaseline features to the train and test dataset*

train\_regression\_data["KNNBaseline\_User"] = model\_train\_evaluation["KNNBaseline\_User"]["Prediction"]

train\_regression\_data["KNNBaseline\_Item"] = model\_train\_evaluation["KNNBaseline\_Item"]["Prediction"]

test\_regression\_data["KNNBaseline\_User"] = model\_test\_evaluation["KNNBaseline\_User"]["Prediction"]

test\_regression\_data["KNNBaseline\_Item"] = model\_test\_evaluation["KNNBaseline\_Item"]["Prediction"]

In [ ]:

train\_regression\_data.head()

Out[ ]:

|  | **User\_ID** | **Movie\_ID** | **Global\_Average** | **User\_Average** | **Movie\_Average** | **SUR1** | **SUR2** | **SUR3** | **SUR4** | **SUR5** | **SMR1** | **SMR2** | **SMR3** | **SMR4** | **SMR5** | **Rating** | **BaselineOnly** | **KNNBaseline\_User** | **KNNBaseline\_Item** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 332 | 25 | 3.5 | 3.142857 | 3.70987 | 4.0 | 5.0 | 3.0 | 3.0 | 5.0 | 5.0 | 3.0 | 2.0 | 3.0 | 2.0 | 5.0 | 3.545173 | 4.747819 | 4.907127 |
| **1** | 463 | 25 | 3.5 | 3.875000 | 3.70987 | 5.0 | 5.0 | 4.0 | 4.0 | 5.0 | 4.0 | 4.0 | 4.0 | 3.0 | 4.0 | 5.0 | 2.667714 | 1.805819 | 2.015138 |
| **2** | 484 | 25 | 3.5 | 5.000000 | 3.70987 | 5.0 | 5.0 | 3.0 | 3.0 | 5.0 | 3.0 | 4.0 | 5.0 | 5.0 | 4.0 | 5.0 | 2.806045 | 1.969524 | 2.046117 |
| **3** | 592 | 25 | 3.5 | 5.000000 | 3.70987 | 5.0 | 4.0 | 5.0 | 3.0 | 4.0 | 5.0 | 4.0 | 3.0 | 3.0 | 3.0 | 5.0 | 4.296783 | 4.875671 | 4.905156 |
| **4** | 640 | 25 | 3.5 | 3.166667 | 3.70987 | 3.0 | 3.0 | 5.0 | 4.0 | 5.0 | 3.0 | 5.0 | 3.0 | 3.0 | 3.0 | 2.0 | 3.520443 | 3.203659 | 3.070491 |

In [ ]:

test\_regression\_data.head()

Out[ ]:

|  | **User\_ID** | **Movie\_ID** | **Global\_Average** | **User\_Average** | **Movie\_Average** | **SUR1** | **SUR2** | **SUR3** | **SUR4** | **SUR5** | **SMR1** | **SMR2** | **SMR3** | **SMR4** | **SMR5** | **Rating** | **BaselineOnly** | **KNNBaseline\_User** | **KNNBaseline\_Item** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 24024 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 | 3.496708 | 3.496708 |
| **1** | 89299 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 | 3.496708 | 3.496708 |
| **2** | 89495 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 | 3.496708 | 3.496708 |
| **3** | 105227 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 4.0 | 3.496708 | 3.496708 | 3.496708 |
| **4** | 500 | 605 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 | 3.496708 | 3.496708 |

In [ ]:

*# Applying Xgboost with the KNN-Baseline newly added features*

x\_train = train\_regression\_data.drop(["User\_ID", "Movie\_ID", "Rating"], axis = 1)

x\_test = test\_regression\_data.drop(["User\_ID", "Movie\_ID", "Rating"], axis = 1)

y\_train = train\_regression\_data["Rating"]

y\_test = test\_regression\_data["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

[13:14:52] WARNING: /workspace/src/objective/regression\_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.

--------------------------------------------------

TRAIN DATA

RMSE : 0.8073658605682237

MAPE : 26.30030493852801

--------------------------------------------------

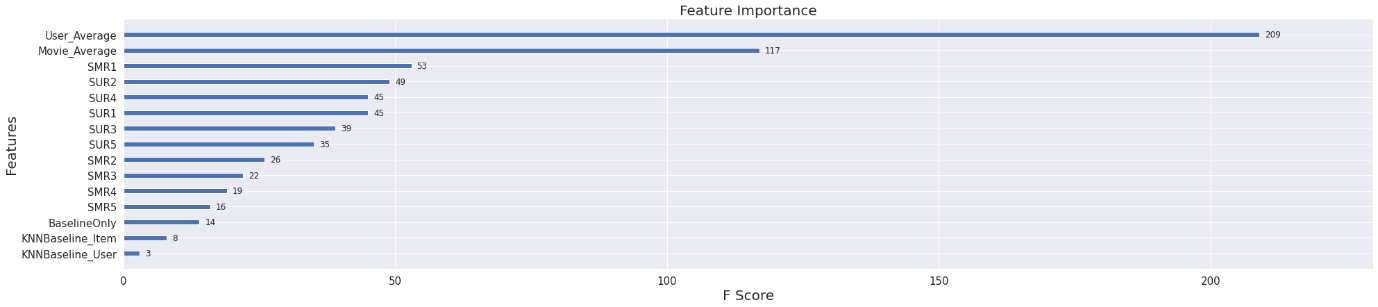
TEST DATA

RMSE : 1.0041836766076881

MAPE : 36.485630009435745

--------------------------------------------------

Time Taken : 0:00:03.862090



Observations:

1. The KNN Baseline features are also not an effective predictor.

In [ ]:

*# Appling the SlopeOne algorithm from the Surprise library*

so = SlopeOne()

train\_result, test\_result = run\_surprise(so, trainset, testset, "SlopeOne")

model\_train\_evaluation["SlopeOne"] = train\_result

model\_test\_evaluation["SlopeOne"] = test\_result

--------------------------------------------------

TRAIN DATA

RMSE = 0.7605719565534242

MAPE = 24.411323730221813

--------------------------------------------------

TEST DATA

RMSE = 0.9991458413459062

MAPE = 36.447592340762306

--------------------------------------------------

Time Taken = 0:00:02.210390

In [ ]:

*# Adding the SlopOne predictions to the train and test datasets*

train\_regression\_data["SlopeOne"] = model\_train\_evaluation["SlopeOne"]["Prediction"]

train\_regression\_data["SlopeOne"] = model\_train\_evaluation["SlopeOne"]["Prediction"]

test\_regression\_data["SlopeOne"] = model\_test\_evaluation["SlopeOne"]["Prediction"]

test\_regression\_data["SlopeOne"] = model\_test\_evaluation["SlopeOne"]["Prediction"]

In [ ]:

*# Matrix Factorization using SVD from Surprise Library*

*# here, n\_factors is the equivalent to dimension 'd' when matrix 'A'*

*# 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 and m\*d.*

param\_grid = {'n\_factors': [5,7,10,15,20,25,35,50,70,90]}

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'])

0.902368311587538

{'n\_factors': 5}

In [ ]:

*# Applying SVD with best parameters*

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 = 0.8311047819961574

MAPE = 28.103553732700792

--------------------------------------------------

TEST DATA

RMSE = 0.9994877877949663

MAPE = 36.46033691358595

--------------------------------------------------

Time Taken = 0:00:01.551123

In [ ]:

*# Matrix Factorization SVDpp with implicit feedback*

*# Hyper-parameter optimization for SVDpp*

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'])

0.896413643654871

{'n\_factors': 10, 'lr\_all': 0.006}

In [ ]:

*#Applying SVDpp with best parameters¶*

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 = 0.7631256379833844

MAPE = 25.417482535813974

--------------------------------------------------

TEST DATA

RMSE = 0.999497987467072

MAPE = 36.45769475543719

--------------------------------------------------

Time Taken = 0:00:24.066648

In [ ]:

*# XGBoost 13 Features + Surprise BaselineOnly + Surprise KNN Baseline + SVD + SVDpp*

train\_regression\_data["SVD"] = model\_train\_evaluation["SVD"]["Prediction"]

train\_regression\_data["SVDpp"] = model\_train\_evaluation["SVDpp"]["Prediction"]

test\_regression\_data["SVD"] = model\_test\_evaluation["SVD"]["Prediction"]

test\_regression\_data["SVDpp"] = model\_test\_evaluation["SVDpp"]["Prediction"]

In [ ]:

train\_regression\_data.head()

Out[ ]:

|  | **User\_ID** | **Movie\_ID** | **Global\_Average** | **User\_Average** | **Movie\_Average** | **SUR1** | **SUR2** | **SUR3** | **SUR4** | **SUR5** | **SMR1** | **SMR2** | **SMR3** | **SMR4** | **SMR5** | **Rating** | **BaselineOnly** | **KNNBaseline\_User** | **KNNBaseline\_Item** | **SlopeOne** | **SVD** | **SVDpp** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 332 | 25 | 3.5 | 3.142857 | 3.70987 | 4.0 | 5.0 | 3.0 | 3.0 | 5.0 | 5.0 | 3.0 | 2.0 | 3.0 | 2.0 | 5.0 | 3.545173 | 4.747819 | 4.907127 | 3.726879 | 3.738652 | 4.001132 |
| **1** | 463 | 25 | 3.5 | 3.875000 | 3.70987 | 5.0 | 5.0 | 4.0 | 4.0 | 5.0 | 4.0 | 4.0 | 4.0 | 3.0 | 4.0 | 5.0 | 2.667714 | 1.805819 | 2.015138 | 2.297585 | 2.664938 | 2.203447 |
| **2** | 484 | 25 | 3.5 | 5.000000 | 3.70987 | 5.0 | 5.0 | 3.0 | 3.0 | 5.0 | 3.0 | 4.0 | 5.0 | 5.0 | 4.0 | 5.0 | 2.806045 | 1.969524 | 2.046117 | 2.607133 | 2.857118 | 2.652471 |
| **3** | 592 | 25 | 3.5 | 5.000000 | 3.70987 | 5.0 | 4.0 | 5.0 | 3.0 | 4.0 | 5.0 | 4.0 | 3.0 | 3.0 | 3.0 | 5.0 | 4.296783 | 4.875671 | 4.905156 | 4.409421 | 4.368232 | 4.495686 |
| **4** | 640 | 25 | 3.5 | 3.166667 | 3.70987 | 3.0 | 3.0 | 5.0 | 4.0 | 5.0 | 3.0 | 5.0 | 3.0 | 3.0 | 3.0 | 2.0 | 3.520443 | 3.203659 | 3.070491 | 3.555524 | 3.554536 | 3.546419 |

In [ ]:

test\_regression\_data.head()

Out[ ]:

|  | **User\_ID** | **Movie\_ID** | **Global\_Average** | **User\_Average** | **Movie\_Average** | **SUR1** | **SUR2** | **SUR3** | **SUR4** | **SUR5** | **SMR1** | **SMR2** | **SMR3** | **SMR4** | **SMR5** | **Rating** | **BaselineOnly** | **KNNBaseline\_User** | **KNNBaseline\_Item** | **SlopeOne** | **SVD** | **SVDpp** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 24024 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 | 3.496708 | 3.496708 | 3.496708 | 3.496708 | 3.496708 |
| **1** | 89299 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 | 3.496708 | 3.496708 | 3.496708 | 3.496708 | 3.496708 |
| **2** | 89495 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 | 3.496708 | 3.496708 | 3.496708 | 3.496708 | 3.496708 |
| **3** | 105227 | 78 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 4.0 | 3.496708 | 3.496708 | 3.496708 | 3.496708 | 3.496708 | 3.496708 |
| **4** | 500 | 605 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.0 | 3.496708 | 3.496708 | 3.496708 | 3.496708 | 3.496708 | 3.496708 |

In [ ]:

*# Applying Xgboost on the feature set*

x\_train = train\_regression\_data.drop(["User\_ID", "Movie\_ID", "Rating"], axis = 1)

x\_test = test\_regression\_data.drop(["User\_ID", "Movie\_ID", "Rating"], axis = 1)

y\_train = train\_regression\_data["Rating"]

y\_test = test\_regression\_data["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

[14:22:55] WARNING: /workspace/src/objective/regression\_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.

--------------------------------------------------

TRAIN DATA

RMSE : 0.8071468705811731

MAPE : 26.299162680035952

--------------------------------------------------

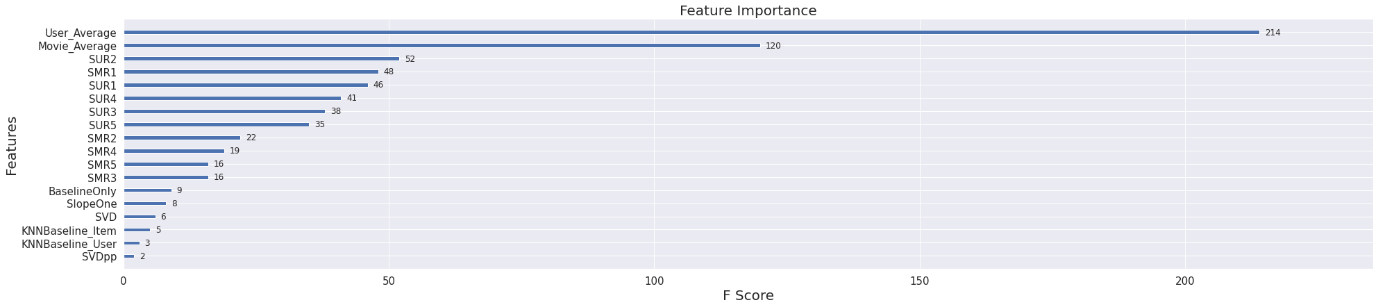
TEST DATA

RMSE : 1.0052311591273282

MAPE : 36.48948664355278

--------------------------------------------------

Time Taken : 0:00:04.508467



Observations:

1. SVD did better than KNNBaseline features but SVDpp turned out to be the most ineffective predictor.

In [ ]:

*# Applying Xgboost with Surprise's BaselineOnly + KNN Baseline + SVD + SVDpp + SlopeOne*

x\_train = train\_regression\_data[["BaselineOnly", "KNNBaseline\_User", "KNNBaseline\_Item", "SVD", "SVDpp", "SlopeOne"]]

x\_test = test\_regression\_data[["BaselineOnly", "KNNBaseline\_User", "KNNBaseline\_Item", "SVD", "SVDpp", "SlopeOne"]]

y\_train = train\_regression\_data["Rating"]

y\_test = test\_regression\_data["Rating"]

train\_result, test\_result = train\_test\_xgboost(x\_train, x\_test, y\_train, y\_test, "XGB\_KNN\_MF\_SO")

model\_train\_evaluation["XGB\_KNN\_MF\_SO"] = train\_result

model\_test\_evaluation["XGB\_KNN\_MF\_SO"] = test\_result

[14:26:39] WARNING: /workspace/src/objective/regression\_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.

--------------------------------------------------

TRAIN DATA

RMSE : 1.0664774841738622

MAPE : 38.51506030087847

--------------------------------------------------

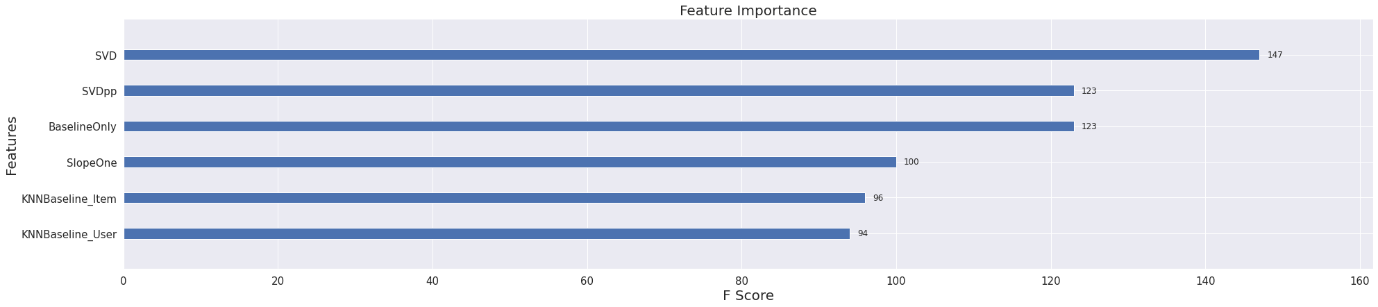
TEST DATA

RMSE : 0.9981137245466845

MAPE : 36.54328232632636

--------------------------------------------------

Time Taken : 0:00:03.196987



Observations:

1. SVD seems to be the best estimator of the rating predictions.
2. SVDpp and Baseline Only also seems to be a important feature.
3. SlopeOne, KNNBaseline features are also decent estimators.

In [ ]:

*# Visualizing the errors of all the models we tested out*

error\_table2 = error\_table.drop(["Train MAPE", "Test MAPE"], axis = 1)

error\_table2.plot(x = "Model", kind = "bar", figsize = (25, 8), grid = True, fontsize = 15)

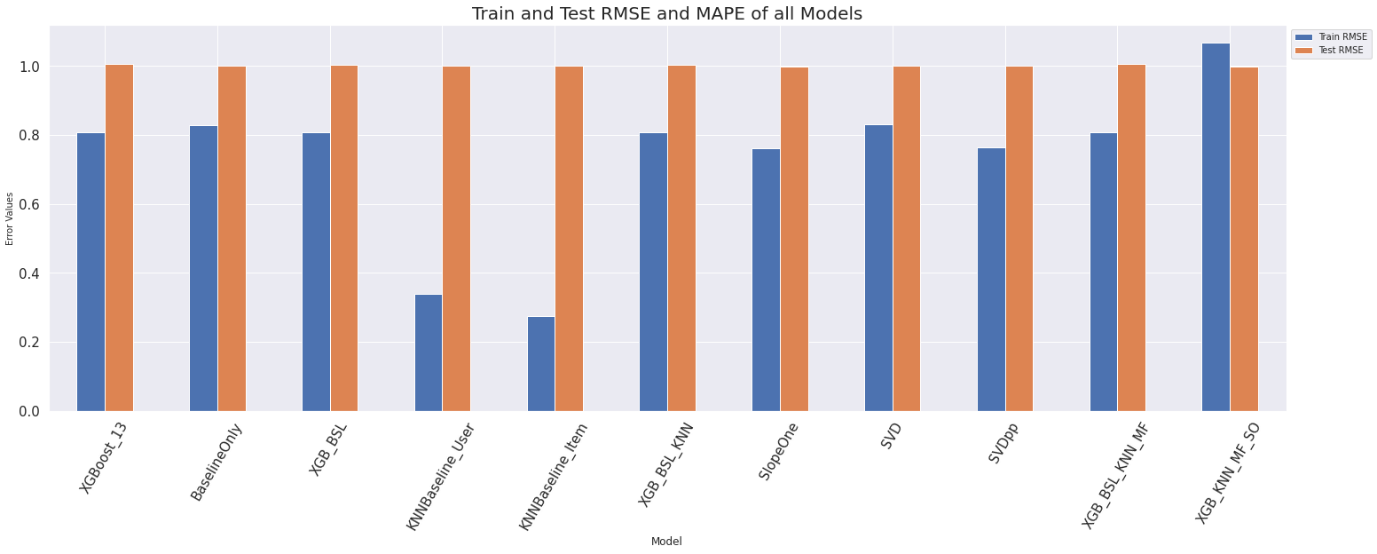
plt.title("Train and Test RMSE and MAPE of all Models", fontsize = 20)

plt.ylabel("Error Values", fontsize = 10)

plt.xticks(rotation=60)

plt.legend(bbox\_to\_anchor=(1, 1), fontsize = 10)

plt.show()



In [ ]:

*# Tabular Values of Errors*

error\_table.drop(["Train MAPE", "Test MAPE"], axis = 1)

Out[ ]:

|  | **Model** | **Train RMSE** | **Test RMSE** |
| --- | --- | --- | --- |
| **0** | XGBoost\_13 | 0.807228 | 1.004849 |
| **1** | BaselineOnly | 0.827371 | 0.999581 |
| **2** | XGB\_BSL | 0.807346 | 1.003603 |
| **3** | KNNBaseline\_User | 0.337843 | 0.999480 |
| **4** | KNNBaseline\_Item | 0.274160 | 0.999480 |
| **5** | XGB\_BSL\_KNN | 0.807366 | 1.004184 |
| **6** | SlopeOne | 0.760572 | 0.999146 |
| **7** | SVD | 0.831105 | 0.999488 |
| **8** | SVDpp | 0.763126 | 0.999498 |
| **9** | XGB\_BSL\_KNN\_MF | 0.807147 | 1.005231 |
| **10** | XGB\_KNN\_MF\_SO | 1.066477 | 0.998114 |

Observations:

1. All the algorithms seems to do great with the differences remaining very close to each other.
2. We can see that by using various rating predicting algorithms together and stacking them up, then using final algorithms seems to result in lowest Testing RMSE. Eg: Surprise's BaselineOnly + KNN Baseline + SVD + SVDpp + SlopeOne together with Xgboost.
3. SlopeOne seems to have lowest Testing RMSE out of all other algorithms.
4. SVDpp and SVD are algorithms showing lower Testing RMSE among rest of the predictors except SlopeOne.

**4.3 Generating Recommendation for Users**

We are using SVDpp to generate atmost 10 recommendated movies for various users.

In [ ]:

*# Testing the recommendations made by SVDpp Algorithm*

from collections import defaultdict

def Get\_top\_n(predictions, n=10):

*# First map the predictions to each user.*

top\_n = defaultdict(list)

for uid, mid, true\_r, est, \_ in predictions:

top\_n[uid].append((mid, est))

*# Then sort the predictions for each user and retrieve the k highest ones.*

for uid, user\_ratings in top\_n.items():

user\_ratings.sort(key=lambda x: x[1], reverse=True)

top\_n[uid] = user\_ratings[:n]

return top\_n

In [ ]:

*# Creating instance of svd\_pp*

svd\_pp = SVDpp(n\_factors = 10, lr\_all = 0.006, verbose=True)

svd\_pp.fit(trainset)

predictions = svd\_pp.test(testset)

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

In [ ]:

*# Saving the training predictions*

train\_pred = svd\_pp.test(trainset.build\_anti\_testset())

top\_n = Get\_top\_n(train\_pred, n=10)

In [ ]:

*# Print the recommended items for each user*

def Generate\_Recommendated\_Movies(u\_id, n=10):

recommend = pd.DataFrame(top\_n[u\_id], columns=["Movie\_Id", "Predicted\_Rating"])

recommend = recommend.merge(movies, how="inner", left\_on="Movie\_Id", right\_on="movieId")

recommend = recommend[["Movie\_Id", "title", "genres", "Predicted\_Rating"]]

return recommend[:n]

In [ ]:

*# Saving the sampled user id list to help generate movies*

sampled\_user\_id = list(top\_n.keys())

In [ ]:

*# Generating recommendation using the user\_Id*

test\_id = random.choice(sampled\_user\_id)

print("The user Id is : ", test\_id)

Generate\_Recommendated\_Movies(test\_id)

The user Id is : 55237

Out[ ]:

|  | **Movie\_Id** | **title** | **genres** | **Predicted\_Rating** |
| --- | --- | --- | --- | --- |
| **0** | 893 | Apartment, The (1960) | Comedy|Drama|Romance | 4.337311 |
| **1** | 2850 | Lady Eve, The (1941) | Comedy|Romance | 4.293874 |
| **2** | 213 | Before the Rain (Pred dozhdot) (1994) | Drama|War | 4.249252 |
| **3** | 2120 | Shadow of a Doubt (1943) | Crime|Drama|Thriller | 4.235916 |
| **4** | 3380 | Hustler, The (1961) | Drama | 4.224963 |
| **5** | 3711 | Anatomy of a Murder (1959) | Drama|Mystery | 4.219006 |
| **6** | 4312 | Man Who Shot Liberty Valance, The (1962) | Crime|Drama|Western | 4.190298 |
| **7** | 6902 | Night of the Hunter, The (1955) | Drama|Film-Noir|Thriller | 4.176537 |
| **8** | 4913 | Witness for the Prosecution (1957) | Drama|Mystery|Thriller | 4.132051 |
| **9** | 7045 | Fog of War: Eleven Lessons from the Life of Ro... | Documentary|War | 4.121651 |

In [ ]:

*# Generating recommendation using the user\_Id*

test\_id = random.choice(sampled\_user\_id)

print("The user Id is : ", test\_id)

Generate\_Recommendated\_Movies(test\_id)

The user Id is : 13514

Out[ ]:

|  | **Movie\_Id** | **title** | **genres** | **Predicted\_Rating** |
| --- | --- | --- | --- | --- |
| **0** | 316 | Shawshank Redemption, The (1994) | Crime|Drama | 4.303549 |
| **1** | 213 | Before the Rain (Pred dozhdot) (1994) | Drama|War | 4.280302 |
| **2** | 2850 | Lady Eve, The (1941) | Comedy|Romance | 4.224732 |
| **3** | 5918 | City of God (Cidade de Deus) (2002) | Action|Adventure|Crime|Drama|Thriller | 4.209128 |
| **4** | 3380 | Hustler, The (1961) | Drama | 4.108861 |
| **5** | 1733 | Fireworks (Hana-bi) (1997) | Crime|Drama | 4.100823 |
| **6** | 7045 | Fog of War: Eleven Lessons from the Life of Ro... | Documentary|War | 4.092413 |
| **7** | 1184 | Once Upon a Time in the West (C'era una volta ... | Action|Drama|Western | 4.071094 |
| **8** | 2120 | Shadow of a Doubt (1943) | Crime|Drama|Thriller | 4.062668 |
| **9** | 9696 | Downfall (Untergang, Der) (2004) | Drama|War | 4.059000 |

In [ ]:

*# Generating recommendation using the user\_Id*

test\_id = random.choice(sampled\_user\_id)

print("The user Id is : ", test\_id)

Generate\_Recommendated\_Movies(test\_id)

The user Id is : 38547

Out[ ]:

|  | **Movie\_Id** | **title** | **genres** | **Predicted\_Rating** |
| --- | --- | --- | --- | --- |
| **0** | 316 | Shawshank Redemption, The (1994) | Crime|Drama | 4.260094 |
| **1** | 5918 | City of God (Cidade de Deus) (2002) | Action|Adventure|Crime|Drama|Thriller | 4.195031 |
| **2** | 2850 | Lady Eve, The (1941) | Comedy|Romance | 4.178648 |
| **3** | 213 | Before the Rain (Pred dozhdot) (1994) | Drama|War | 4.153221 |
| **4** | 1733 | Fireworks (Hana-bi) (1997) | Crime|Drama | 4.097498 |
| **5** | 3380 | Hustler, The (1961) | Drama | 4.086681 |
| **6** | 893 | Apartment, The (1960) | Comedy|Drama|Romance | 4.063860 |
| **7** | 1184 | Once Upon a Time in the West (C'era una volta ... | Action|Drama|Western | 4.058724 |
| **8** | 3179 | Hard-Boiled (Lat sau san taam) (1992) | Action|Crime|Drama|Thriller | 4.035215 |
| **9** | 937 | It's a Wonderful Life (1946) | Drama|Fantasy|Romance | 4.034673 |

In [ ]:

*# Generating recommendation using the user\_Id*

test\_id = random.choice(sampled\_user\_id)

print("The user Id is : ", test\_id)

Generate\_Recommendated\_Movies(test\_id)

The user Id is : 47133

Out[ ]:

|  | **Movie\_Id** | **title** | **genres** | **Predicted\_Rating** |
| --- | --- | --- | --- | --- |
| **0** | 213 | Before the Rain (Pred dozhdot) (1994) | Drama|War | 4.525050 |
| **1** | 2850 | Lady Eve, The (1941) | Comedy|Romance | 4.476013 |
| **2** | 5918 | City of God (Cidade de Deus) (2002) | Action|Adventure|Crime|Drama|Thriller | 4.378274 |
| **3** | 3380 | Hustler, The (1961) | Drama | 4.363563 |
| **4** | 4913 | Witness for the Prosecution (1957) | Drama|Mystery|Thriller | 4.356607 |
| **5** | 1733 | Fireworks (Hana-bi) (1997) | Crime|Drama | 4.333344 |
| **6** | 3179 | Hard-Boiled (Lat sau san taam) (1992) | Action|Crime|Drama|Thriller | 4.326098 |
| **7** | 531 | Shadowlands (1993) | Drama|Romance | 4.292230 |
| **8** | 7045 | Fog of War: Eleven Lessons from the Life of Ro... | Documentary|War | 4.286349 |
| **9** | 3005 | Kagemusha (1980) | Drama|War | 4.277592 |

**5. Conclusion**

In this project, we learned the importance of Recommendation Systems, the types of recommender systems being implemented, and how to use matrix factorization to enhance a system.

We then built a movie recommendation system that considers user-user similarity, movie-movie similarity, global averages and matrix factorization. These concepts can be applied to any other user-item interactions systems.

We tried generating recommendations based on similarity matrix and Collaborative Filtering techniques.

We tried to predict the ratings for movies that the user might give based on its past rating behaviours and measure the accuracy using RMSE and MAPE error metrics.

Surely, there is huge scope of improvement and tring out different techniques and ML/DL algorithms.

In [ ]: