ECE 657-A Project

April 5, 2019

1 ECE 657A - Project

1.0.1 Import libraries

```
In [1]: import os
        import pandas as pd
        import numpy as np
        import seaborn as sns
        from datetime import datetime
        import random
        import matplotlib
        import matplotlib.pyplot as plt
        from scipy import sparse
        from sklearn.metrics.pairwise import cosine_similarity
        from sklearn.metrics import mean_squared_error
        from keras.layers import Input, Embedding, Reshape, Dot, Concatenate, Dense, Dropout
        from keras.models import Model
        from surprise import Reader, Dataset,evaluate
        from surprise import SVD
```

```
Using TensorFlow backend.
```

1.1 Load movie_titles.csv

```
In [2]: movie_titles = pd.read_csv('movie_titles.csv', encoding = 'ISO-8859-1', header = None,
```

```
In [3]: movie_titles.shape
```

```
Out[3]: (17770, 2)
```

```
In [4]: movie_titles.head()
```

```
Out[4]:       Year                           Name
        Id
        1   2003.0                Dinosaur Planet
        2   2004.0     Isle of Man TT 2004 Review
        3   1997.0                      Character
        4   1994.0  Paula Abdul's Get Up & Dance
        5   2004.0       The Rise and Fall of ECW
```

1.2 Data Pre-processing

1.2.1 We have 4 text files consist of ratings data. For each Movie, Movie Id is present in first line and then its corresponding rating data.

```
In [5]: temp_df = pd.read_csv('combined_data_1.txt')
        temp_df.head()
```

```
Out[5]:                     1:
        1488844 3.0  2005-09-06
        822109  5.0  2005-05-13
        885013  4.0  2005-10-19
        30878   4.0  2005-12-26
        823519  3.0  2004-05-03
```

1.2.2 In data cleaning our main task is to merge all text files and create one csv file by removing movie IDs lines from text files and appending it to each appropriate rating data (We have performed all analysis on one text file due to lack of processing power)

```
In [6]: if not os.path.isfile("NetflixRatings.csv"):
            csv_file = open("NetflixRatings.csv", mode = "w")
            data_files = ['combined_data_1.txt']
            for data_file in data_files:
                with open(data_file) as file:
                    for line in file:
                        line = line.strip()
                        if line.endswith(":"):
                            movieID = line.replace(":", "")
                        else:
                            rating = []
                            rating = [k for k in line.split(",")]
                            rating.insert(0, movieID)
                            csv_file.write(",".join(rating))
                            csv_file.write("\n")
            csv_file.close()
```

1.2.3 Creatting dataframe from generated csv file

```
In [7]: if not os.path.isfile("NetflixData.pkl"):
            final_df = pd.read_csv("NetflixRatings.csv", sep=",", names = ["MovieID","CustID",
            final_df["Date"] = pd.to_datetime(final_df["Date"])
            final_df.sort_values(by = "Date", inplace = True)
```

1.2.4 Storing this dataframe into pickle object for later use

```
In [8]: if not os.path.isfile("NetflixData.pkl"):
            final_df.to_pickle("NetflixData.pkl")
        else:
            final_df = pd.read_pickle("NetflixData.pkl")
```

```
In [9]: final_df.shape
```

```
Out[9]: (24053764, 4)
```

1.2.5 Checking for NaNs

```
In [10]: final_df.isnull().sum()
```

```
Out[10]: MovieID    0
         CustID     0
         Ratings    0
         Date       0
         dtype: int64
```

1.2.6 Checking for duplicate entries

```
In [11]: final_df.duplicated(["MovieID","CustID", "Ratings"]).sum()
```

```
Out[11]: 0
```

1.2.7 Finding unique movies

```
In [12]: len(np.unique(final_df["MovieID"]))
```

```
Out[12]: 4499
```

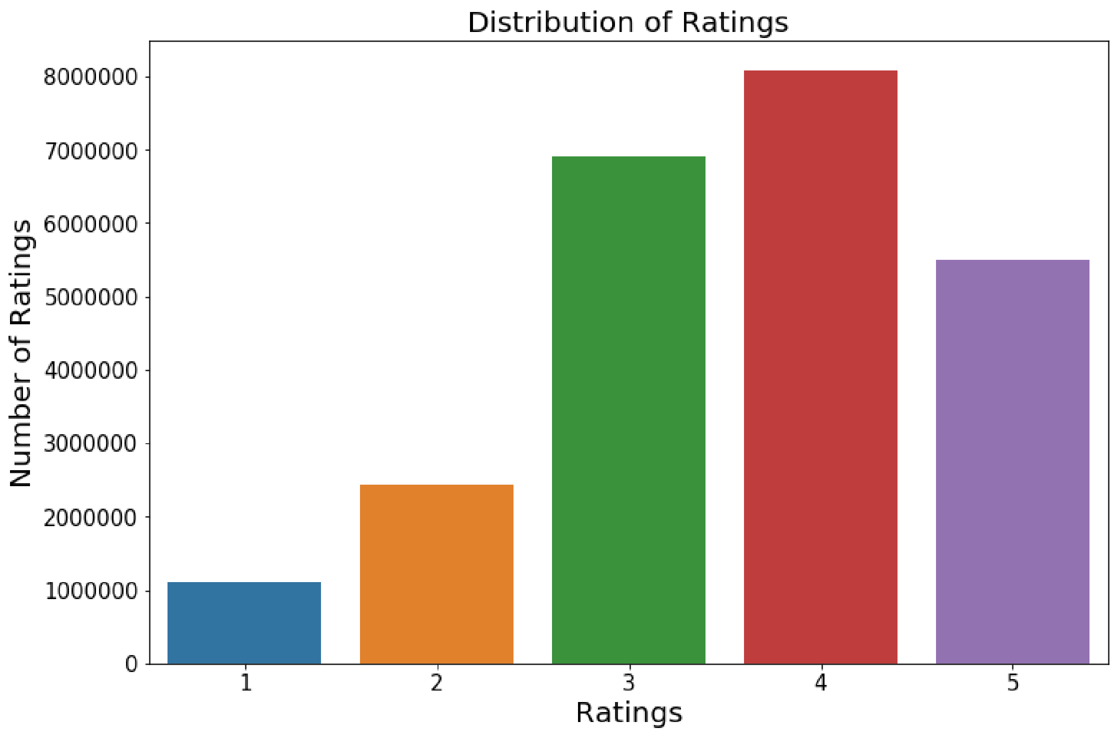
1.2.8 Finding unique users

```
In [13]: len(np.unique(final_df["CustID"]))
```

```
Out[13]: 470758
```

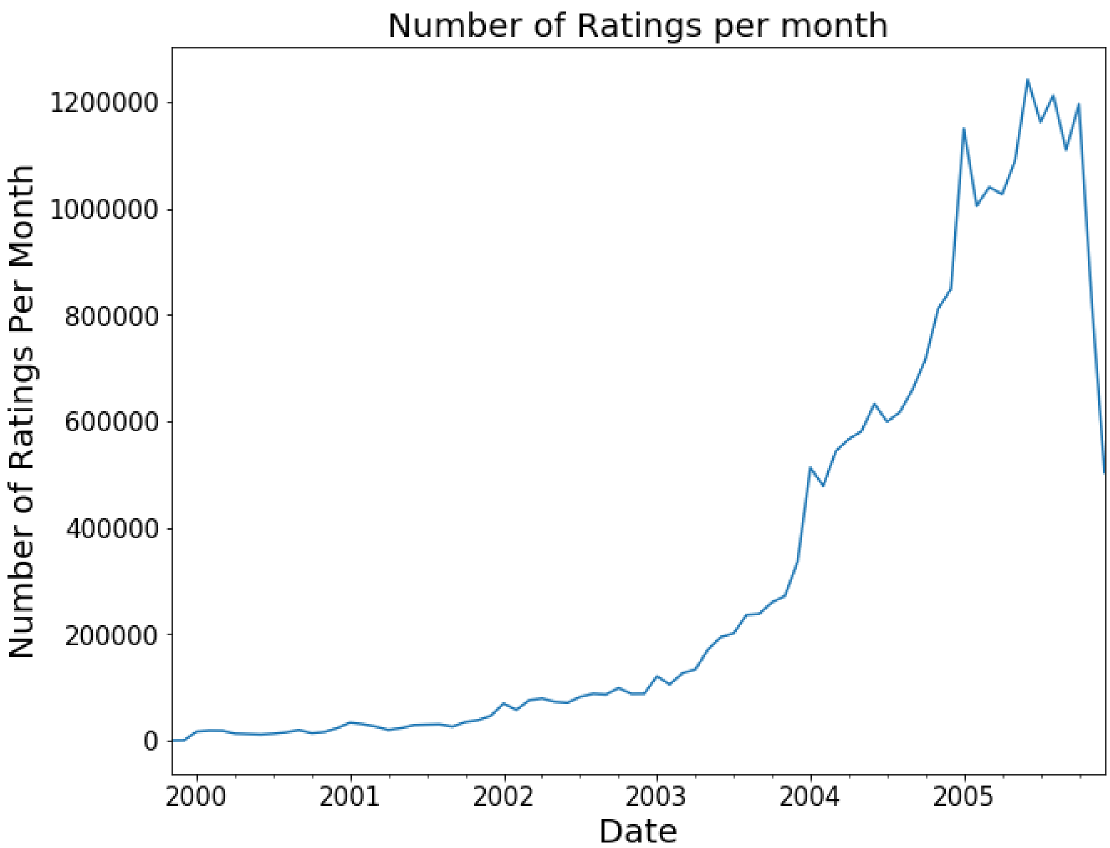
1.2.9 Distribution of ratings

```
In [38]: plt.figure(figsize = (12, 8))
         ax = sns.countplot(x="Ratings", data=final_df)
         plt.tick_params(labelsize = 15)
         plt.title("Distribution of Ratings", fontsize = 20)
         plt.xlabel("Ratings", fontsize = 20)
         plt.ylabel("Number of Ratings", fontsize = 20)
         plt.savefig('01_rating_distribution.png')
         plt.show()
```



1.2.10 Trends of Ratings Count Per Month

In [39]:
```
plt.figure(figsize = (10,8))
ax = final_df.resample("M", on = "Date")["Ratings"].count().plot()
ax.set_title("Number of Ratings per month", fontsize = 20)
ax.set_xlabel("Date", fontsize = 20)
ax.set_ylabel("Number of Ratings Per Month", fontsize = 20)
plt.tick_params(labelsize = 15)
plt.savefig('02_trends_of_ratings_per_month.png')
plt.show()
```



1.2.11 Distributions of number of rating given by user

In [14]:
```
no_of_movies_rated_by_user = final_df.groupby(by = "CustID")["Ratings"].count().sort_
```

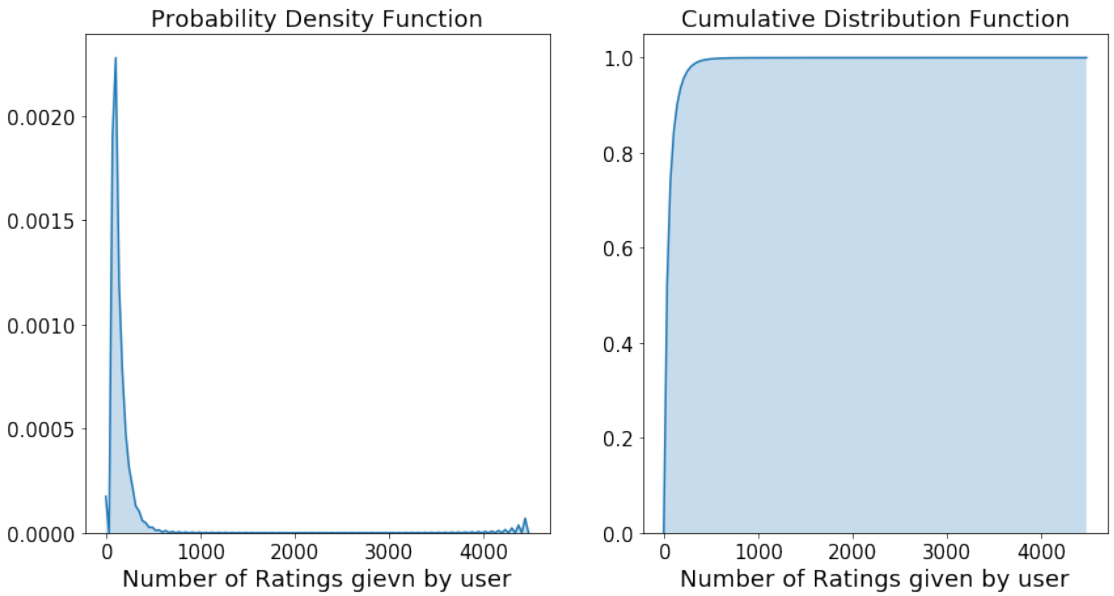
In [15]:
```
no_of_movies_rated_by_user.head()
```

Out[15]:
```
CustID
305344     4467
387418     4422
2439493    4195
1664010    4019
2118461    3769
Name: Ratings, dtype: int64
```

In [40]:
```
fig, axes = plt.subplots(ncols = 2, nrows = 1, figsize=(14,7))
sns.kdeplot(no_of_movies_rated_by_user.values, shade = True, ax = axes[0])
axes[0].set_title("Probability Density Function", fontsize = 18)
axes[0].set_xlabel("Number of Ratings gievn by user", fontsize = 18)
axes[0].tick_params(labelsize = 15)

sns.kdeplot(no_of_movies_rated_by_user.values, shade = True, cumulative = True, ax = a
```

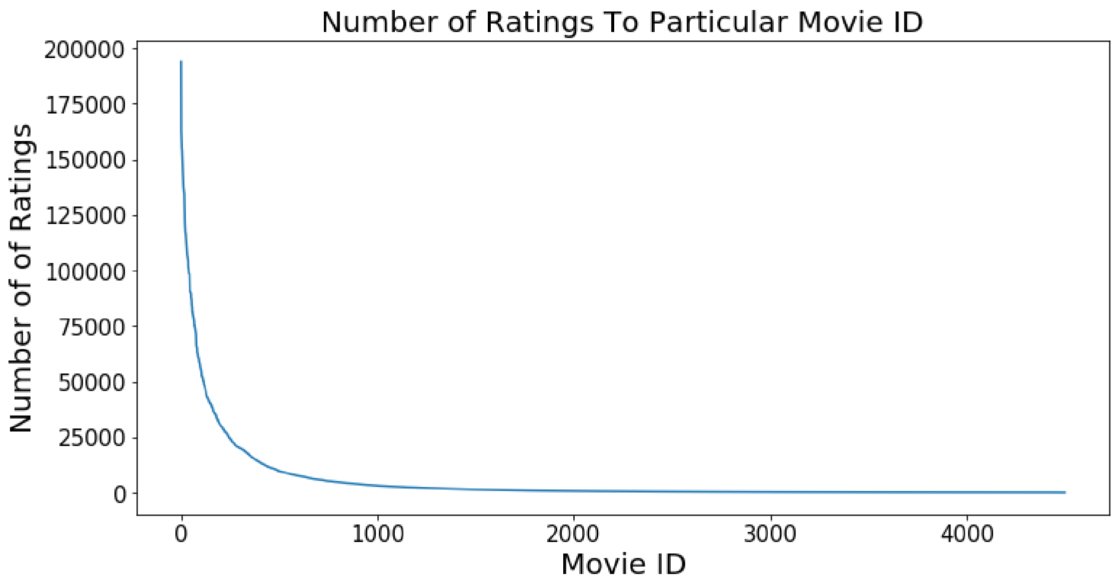
```
axes[1].set_title("Cumulative Distribution Function", fontsize = 18)
axes[1].set_xlabel("Number of Ratings given by user", fontsize = 18)
axes[1].tick_params(labelsize = 15)
plt.savefig('03_distribution-of_ratings_by_user.png')
plt.show()
```



1.2.12 Trend of Ratings Given To Each Movie

```
In [16]: no_of_ratings_to_each_movie = final_df.groupby(by = "MovieID")["Ratings"].count().sor
```

```
In [41]: fig = plt.figure(figsize = (12, 6))
         plt.title("Number of Ratings To Particular Movie ID", fontsize = 20)
         plt.xlabel("Movie ID", fontsize = 20)
         plt.ylabel("Number of of Ratings", fontsize = 20)
         plt.plot(no_of_ratings_to_each_movie.values)
         plt.tick_params(labelsize = 15)
         plt.savefig('04_trend_of_ratings_gievn_to_each_movie.png')
         plt.show()
```



1.2.13 Filter movies with less number of rating by setting threshold of 15000 ratings

In [17]:
```
threshold = 15000
filter_movies = (final_df['MovieID'].value_counts()>threshold)
filter_movies = filter_movies[filter_movies].index.tolist()
filter_movies
```

Out[17]:
```
[1905,
 2152,
 3860,
 4432,
 571,
 3938,
 4306,
 2452,
 1962,
 3962,
 1145,
 3624,
 2372,
 3427,
 2782,
 3106,
 1220,
 2862,
 4123,
 1307,
 30,
```

```
3282,
457,
3151,
1542,
1428,
1798,
1865,
1180,
886,
2913,
3925,
4356,
2122,
3825,
758,
2391,
4472,
607,
313,
191,
1110,
1470,
3917,
3756,
4043,
175,
4345,
483,
985,
2342,
3333,
2112,
2095,
2612,
4266,
2874,
3638,
798,
197,
2660,
2580,
1144,
2200,
1202,
3368,
357,
299,
3610,
```

3371,
3605,
1406,
3320,
1102,
1744,
708,
3290,
3079,
329,
2800,
312,
1754,
4330,
1799,
3254,
3905,
2186,
1561,
2743,
3538,
468,
1073,
1615,
788,
1719,
3433,
1810,
482,
4315,
1975,
4302,
3864,
2470,
2465,
1650,
4227,
3385,
4056,
2617,
2290,
3782,
2699,
2209,
2462,
3713,
4262,
406,

```
2430,
1703,
1843,
2953,
290,
331,
3463,
4216,
4141,
1324,
2128,
2992,
658,
1625,
3418,
241,
1590,
2395,
3138,
3684,
3966,
1267,
1659,
2578,
2009,
1466,
4402,
1571,
3936,
1046,
4256,
3730,
1509,
2922,
334,
3648,
28,
2178,
1255,
705,
4393,
3798,
1642,
143,
2000,
963,
2594,
3650,
```

```
443,
270,
3148,
2499,
361,
2960,
1174,
199,
3579,
3446,
4369,
1305,
1066,
2457,
550,
4159,
2171,
4135,
4488,
3315,
1289,
692,
1832,
3875,
273,
2192,
872,
789,
3824,
3197,
4080,
2734,
4389,
896,
4384,
3198,
2252,
111,
3017,
3541,
3626,
413,
3165,
1877,
4364,
2001,
2360,
2690,
```

```
295,
3256,
3085,
3817,
528,
2554,
2495,
4341,
1918,
494,
1700,
2015,
257,
2443,
3466,
4392,
330,
3893,
2016,
2890,
720,
2675,
3611,
3274,
3161,
1682,
1861,
4149,
442,
851,
1367,
4109,
311,
1314,
1770,
3489,
252,
3098,
746,
223,
1467,
4479,
937,
473,
3355,
3617,
3350,
3522,
```

```
1901,
2851,
1645,
2172,
2153,
2866,
2139,
3015,
148,
1890,
2779,
2809,
269,
2938,
1585,
962,
3239,
4089,
3078,
1833,
3058,
4260,
3728,
3890,
660,
1020,
1138,
1974,
2348,
3113,
2389,
2371,
1902,
353,
3900,
3894,
3715,
516,
3364,
187,
1709,
2783,
1027,
3168,
2803,
3689,
3670,
2400,
```

```
2456,
2780,
831,
2680,
1148,
3128,
862,
3414,
118,
2012,
2942,
1661,
2136,
108,
83,
3267,
1578,
940,
1637,
2577,
3071,
989,
1435,
3222,
1425,
3840,
3153,
3265,
1035,
1983,
2751,
3434,
3680,
3312,
58,
2848,
1595,
2162,
4171,
759,
3544,
2376,
3742,
2254,
1495,
2163,
4284,
3920,
```

```
 3347,
 2755,
 3253,
 1482,
 3326,
 2519,
 4420,
 1803,
 819,
 4269,
 2072,
 3046,
 2329,
 3954,
 3478,
 2905,
 2212,
 501,
 3285,
 4418,
 3972]
```

1.2.14 Filter users who gave less number of ratings by setting threshold of 250 ratings

```
In [18]: threshold = 250
         filter_users = (final_df['CustID'].value_counts()>threshold)
         filter_users = filter_users[filter_users].index.tolist()
         filter_users
```

```
Out[18]: [305344,
          387418,
          2439493,
          1664010,
          2118461,
          1639792,
          1314869,
          1461435,
          1932594,
          2606799,
          2056022,
          1114324,
          752642,
          491531,
          1663888,
          1403217,
          727242,
          1473980,
          716173,
```

```
798296,
1001129,
1852040,
507603,
1792741,
2625420,
303948,
2457095,
2143500,
2147527,
16272,
1299887,
1037245,
1806515,
322009,
2238060,
1061195,
1710658,
57633,
1927580,
2237185,
794999,
682963,
1673185,
1977959,
525356,
2291306,
1784150,
2083367,
1298511,
786312,
530789,
1028463,
2062350,
504620,
1227322,
1300759,
1110156,
636262,
1935793,
1105029,
184705,
789014,
2256485,
2537543,
1612901,
1876520,
1602153,
```

2040859,
1272379,
1902838,
952156,
3321,
818752,
862596,
1707198,
2061495,
319058,
2315012,
1470123,
2460347,
447759,
1819462,
1903324,
1519378,
2176465,
341649,
722591,
595778,
238740,
2102030,
1563935,
647979,
555528,
1984086,
2150746,
2602249,
769702,
402266,
2297136,
147386,
2327743,
71594,
507094,
789969,
1509395,
2485642,
1166912,
1984315,
74441,
1516350,
1033930,
1029163,
1431356,
397212,
1557557,

```
2634967,
1146000,
1205174,
2584676,
1452127,
2248080,
1233297,
235789,
1777406,
394895,
743633,
1559083,
2119007,
2157060,
2220732,
316155,
844028,
411290,
958687,
638020,
684876,
929308,
2255880,
166041,
825353,
1139570,
873713,
1317671,
716874,
1257336,
908205,
2035299,
157667,
2225743,
1268699,
2514777,
518137,
121182,
2284890,
603277,
49890,
2604976,
2260832,
2541385,
1988735,
1370564,
1494683,
1515355,
```

```
199257,
2089599,
178860,
1556831,
230112,
159159,
2519299,
952063,
1532336,
2554745,
2375962,
188613,
2588510,
1374197,
883478,
1986034,
2534061,
1184498,
306569,
1978089,
2450433,
2266083,
864172,
2438449,
2299661,
629389,
1301819,
1107588,
411705,
1757328,
2475820,
1661344,
2320697,
386947,
78404,
655385,
1567202,
608234,
461075,
2538506,
732363,
1001928,
237263,
307427,
684531,
2488357,
901520,
1907667,
```

1771252,
2155681,
1875549,
503235,
1086481,
691808,
642384,
2099446,
1830000,
2018894,
320314,
438340,
1333,
1934002,
1002509,
2626336,
2389367,
1220170,
2207031,
101807,
1038880,
663118,
1567970,
2329026,
1872980,
1117152,
1045879,
2479320,
923631,
1616870,
1190829,
462171,
2268573,
837133,
825819,
1561134,
524142,
1030857,
1949871,
2194079,
1309838,
1116080,
2218735,
1737149,
1092521,
2029979,
620010,
576990,

```
21823,
893014,
835265,
75976,
981753,
2552319,
2404142,
1913414,
574834,
243612,
1179117,
1627955,
1569593,
383451,
1629440,
1816884,
2217731,
2256731,
221926,
76196,
1026933,
1380915,
11043,
2360906,
594196,
998236,
2433610,
2499884,
830370,
2484944,
2588755,
2147714,
2282041,
1582952,
2464081,
429260,
329958,
1995860,
407037,
408457,
2127108,
1357209,
1224299,
414536,
285174,
1704223,
2600558,
1420308,
```

```
1238184,
1072312,
1945809,
2429294,
2171187,
2124351,
1745107,
1933317,
1744889,
1660348,
2070798,
831103,
1999176,
1640334,
1189269,
578191,
356615,
775020,
651230,
2041854,
1282217,
2494753,
589682,
1123175,
1865249,
2430041,
1037705,
1480900,
1495111,
240923,
754161,
1979820,
1650301,
1762695,
257622,
575714,
325073,
200255,
1528604,
2254751,
1818059,
1998055,
302344,
2535052,
504440,
1143750,
670488,
1859573,
```

```
10268,
1284016,
400440,
1156852,
199435,
2398923,
332698,
155342,
327122,
1643146,
1684416,
753107,
364749,
1545709,
860520,
2232563,
826696,
1684232,
1675680,
66109,
922551,
2339135,
877394,
998502,
122197,
27061,
1042938,
601317,
1279360,
2043784,
74320,
57186,
2385435,
1106241,
340479,
1123618,
399971,
151821,
624388,
53694,
2566515,
2354740,
2185614,
2495584,
491513,
1829748,
2299951,
125296,
```

595870,
71743,
2523426,
2291170,
1988192,
490057,
879760,
1452669,
97893,
1334853,
686510,
187896,
2328511,
2645579,
752853,
1310425,
2520506,
1645904,
549534,
467182,
2538567,
1223912,
111033,
349572,
2226510,
428688,
136477,
785992,
1420020,
946380,
1229628,
717542,
1147641,
2019240,
1181550,
706832,
151004,
170248,
929407,
1114342,
165247,
768876,
1548363,
10374,
954742,
1878798,
2203225,
1007577,

```
1558286,
823802,
2166018,
16273,
2511278,
701443,
620741,
2250323,
962019,
2029858,
1662714,
1788346,
272840,
1119113,
2005241,
968184,
1800701,
974174,
2446249,
478176,
485601,
380505,
1287892,
1398256,
1490538,
2101288,
2251582,
10679,
2209351,
2535512,
107776,
1484157,
455024,
2417531,
974823,
2393478,
1919269,
2376301,
953110,
1257454,
1814516,
2442646,
2463759,
1583631,
515436,
2450942,
320856,
277837,
```

1957786,
872408,
110938,
2366462,
12812,
240748,
2630337,
491331,
567705,
1274035,
1430108,
461110,
1073300,
685565,
1194911,
992021,
326356,
17864,
2401585,
2023883,
2174151,
374741,
2226525,
2012897,
2136927,
361886,
1785209,
2290732,
310887,
1887657,
241634,
1255703,
1663694,
1421913,
2550506,
2153721,
492291,
832780,
2509048,
1519692,
452667,
232877,
1575462,
1456579,
165374,
2350588,
290916,
2049734,

```
1877483,
411512,
1385788,
230265,
828349,
1813197,
596995,
371701,
1044050,
2545730,
2102846,
753231,
1741099,
2100574,
1720861,
1559022,
1072585,
612111,
1275094,
2394024,
2349412,
286684,
1769985,
499257,
206893,
764785,
2260788,
1820960,
779811,
651541,
603543,
522180,
835353,
828444,
1399249,
2237988,
49585,
1087944,
1054564,
2519865,
983069,
965104,
1489224,
476695,
2198123,
1777614,
111086,
2536547,
```

```
525071,
1625122,
723448,
1727232,
1615850,
282654,
826184,
922922,
1351421,
2541025,
1823645,
48607,
1109043,
1458854,
1002943,
128389,
1177340,
1741303,
2584980,
2037693,
1480781,
942523,
2398061,
182203,
2244849,
1559165,
1033433,
2406705,
817339,
1428930,
1112365,
758383,
1181558,
1867238,
1134379,
1573228,
200805,
1397843,
551208,
2289197,
279966,
1413367,
2110645,
1174530,
1672832,
759737,
306466,
1174811,
```

```
1578801,
2601377,
1034831,
256676,
2526414,
774602,
1526365,
1563698,
953604,
524295,
1480571,
2640550,
862040,
1565175,
1952860,
1605580,
2387463,
91234,
890573,
2366006,
1184537,
1611056,
923517,
43751,
416560,
334643,
2043149,
431326,
1310250,
702403,
342510,
2110107,
922812,
170551,
760481,
2069442,
2554035,
826270,
2409465,
1313126,
1115451,
1629801,
1935592,
2035595,
2232041,
2375169,
2588416,
2205594,
```

```
1435891,
1782185,
1486829,
2043555,
659526,
1294425,
980292,
1821474,
2365581,
2024111,
1312661,
1662098,
325844,
2023518,
224795,
973313,
1596469,
1130133,
437586,
885386,
1832073,
1383836,
97578,
2536114,
2503887,
1200256,
554137,
46650,
1060057,
1848938,
219888,
200427,
282887,
1296679,
1253619,
70476,
1719610,
1065445,
2495474,
701962,
248904,
2455281,
1173289,
1393159,
940516,
1057992,
309424,
780768,
```

```
867311,
265257,
1251664,
1815129,
2524669,
924960,
981199,
1242044,
1827859,
377937,
655601,
652618,
2640278,
952582,
2637886,
138106,
1510091,
442963,
1191408,
1297347,
1828803,
2607300,
2278766,
2145441,
2144655,
1132940,
2161899,
1348456,
2513621,
275600,
1424487,
1992205,
2540684,
2126215,
700490,
2141105,
1299355,
635056,
242681,
892694,
2103029,
819314,
510834,
1610234,
864647,
2640760,
748141,
1105686,
```

128311,
5980,
403915,
109177,
956229,
1939431,
1456270,
2356870,
55339,
138835,
2636478,
247794,
2152273,
1814136,
1392942,
2457720,
2295594,
2257189,
1450883,
1374216,
1899913,
2551641,
1446775,
1635582,
1520166,
189887,
194528,
1150705,
1337026,
596255,
1226844,
1854658,
349331,
1800459,
2491620,
1459727,
2086265,
670416,
408645,
1327900,
515441,
318409,
1733406,
1599243,
1538355,
2107068,
151153,
1207990,

2599262,
828344,
1133763,
1092677,
1706873,
3998,
152858,
1130380,
1771365,
2388205,
720503,
2059696,
835847,
1812378,
1099833,
567875,
504211,
1847613,
2345098,
1416836,
472734,
2598647,
1201695,
1110164,
1696308,
935732,
1444777,
2033070,
427186,
2000256,
1009622,
2552315,
250509,
1439882,
2472640,
962151,
1675194,
1592339,
2556305,
2064995,
2332765,
1477153,
1488844,
1726438,
2264426,
432259,
511899,
507921,

1951144,
1167767,
407877,
441361,
2549238,
920932,
2518947,
782611,
2531844,
1087416,
1772901,
2205259,
2021377,
348960,
1436780,
1743756,
227799,
1109700,
1365840,
236160,
847313,
1914525,
683254,
1161725,
1232164,
1932377,
1125150,
1197493,
1955660,
2041252,
572481,
637266,
1841531,
1906860,
2312349,
609908,
2077664,
1486956,
401047,
1057021,
827240,
1262753,
1178846,
689488,
1197972,
1103080,
332300,
476512,

```
1783225,
436122,
2372141,
1332725,
1717979,
2228750,
1047830,
452299,
328212,
43530,
2120912,
1533807,
1268483,
1281882,
1763979,
1984978,
2319282,
1329904,
894205,
2030523,
1703879,
1181609,
366233,
1880039,
545026,
290700,
838130,
1084626,
904391,
574499,
2339734,
1875008,
1116110,
388514,
2198070,
2418086,
1758002,
32902,
2133714,
1125075,
296452,
1197233,
317779,
1018379,
154669,
1049142,
1906611,
2465236,
```

```
        1551751,
        2512235,
        2584252,
        1811832,
        778615,
        1210199,
        1276711,
        804027,
        2576843,
        974375,
        360005,
        1737529,
        2481504,
        2333850,
        262411,
        1084999,
        1486501,
        2238629,
        1449358,
        486798,
        2446506,
        ...]
```

1.2.15 Making final dataframe after filtering users and movies with less ratings

```
In [19]: df_filterd = final_df[(final_df['MovieID'].isin(filter_movies)) & (final_df['CustID']
         del filter_movies, filter_users, threshold
         print('Shape User-Ratings before filtering:\t{}'.format(final_df.shape))
         print('Shape User-Ratings after filtering:\t{}'.format(df_filterd.shape))
```

```
Shape User-Ratings before filtering:        (24053764, 4)
Shape User-Ratings after filtering:        (2333077, 4)
```

1.2.16 Dropping data column as it is not going to use in further analysis

```
In [20]: df_filterd = df_filterd.drop('Date', axis=1).sample(frac=1).reset_index(drop=True)
```

1.2.17 Selecting the appropriate split size by testing various split sizes on matrix factorization with gradient descent technique

```
In [37]: split_ratios = [0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50]
         rmse=[]
         mape=[]
         for split in split_ratios:
             n = int(len(df_filterd) * split)
             df_train = df_filterd[:-n]
             df_test = df_filterd[-n:]
             user_id_mapping = {id:i for i, id in enumerate(df_filterd['CustID'].unique())}
```

```python
movie_id_mapping = {id:i for i, id in enumerate(df_filterd['MovieID'].unique())}


# Create correctly mapped train- & testset
train_user_data = df_train['CustID'].map(user_id_mapping)
train_movie_data = df_train['MovieID'].map(movie_id_mapping)

test_user_data = df_test['CustID'].map(user_id_mapping)
test_movie_data = df_test['MovieID'].map(movie_id_mapping)


# Get input variable-sizes
users = len(user_id_mapping)
movies = len(movie_id_mapping)
embedding_size = 10


##### Create model
# Set input layers
user_id_input = Input(shape=[1], name='user')
movie_id_input = Input(shape=[1], name='movie')

# Create embedding layers for users and movies
user_embedding = Embedding(output_dim=embedding_size,
                           input_dim=users,
                           input_length=1,
                           name='user_embedding')(user_id_input)
movie_embedding = Embedding(output_dim=embedding_size,
                            input_dim=movies,
                            input_length=1,
                            name='item_embedding')(movie_id_input)

# Reshape the embedding layers
user_vector = Reshape([embedding_size])(user_embedding)
movie_vector = Reshape([embedding_size])(movie_embedding)

# Compute dot-product of reshaped embedding layers as prediction
y = Dot(1, normalize=False)([user_vector, movie_vector])

# Setup model
model = Model(inputs=[user_id_input, movie_id_input], outputs=y)
model.compile(loss='mse', optimizer='adam')


# Fit model
model.fit([train_user_data, train_movie_data],
          df_train['Ratings'],
          batch_size=256,
```

```
                  epochs=3,
                  validation_split=0.1,
                  shuffle=True)

        # Test model
        y_pred = model.predict([test_user_data, test_movie_data])
        y_true = df_test['Ratings'].values

        #  Compute RMSE
        mf_rmse = np.sqrt(mean_squared_error(y_pred=y_pred, y_true=y_true))
        mf_mape = np.mean(abs((y_true[:1000] - y_pred[:1000])/y_true[:1000]))*100
        print('Testing Result With Keras Matrix-Factorization with GD: {:.4f} RMSE'.forma
        print('Testing Result With Keras Matrix-Factorization with GD: {:.4f} MPAE'.forma
        rmse.append(mf_rmse)
        mape.append(mf_mape)
```

```
WARNING:tensorflow:From /home/harshit/.local/lib/python3.6/site-packages/tensorflow/python/fra
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From /home/harshit/.local/lib/python3.6/site-packages/tensorflow/python/ops/
Instructions for updating:
Use tf.cast instead.
Train on 1784804 samples, validate on 198312 samples
Epoch 1/3
1784804/1784804 [==============================] - 19s 11us/step - loss: 2.9968 - val_loss: 0.8
Epoch 2/3
1784804/1784804 [==============================] - 18s 10us/step - loss: 0.8515 - val_loss: 0.8
Epoch 3/3
1784804/1784804 [==============================] - 18s 10us/step - loss: 0.8448 - val_loss: 0.8
Testing Result With Keras Matrix-Factorization with GD: 0.9142 RMSE
Testing Result With Keras Matrix-Factorization with GD: 37.9351 MPAE
Train on 1679815 samples, validate on 186647 samples
Epoch 1/3
1679815/1679815 [==============================] - 17s 10us/step - loss: 3.0621 - val_loss: 0.8
Epoch 2/3
1679815/1679815 [==============================] - 17s 10us/step - loss: 0.8519 - val_loss: 0.8
Epoch 3/3
1679815/1679815 [==============================] - 17s 10us/step - loss: 0.8468 - val_loss: 0.8
Testing Result With Keras Matrix-Factorization with GD: 0.9163 RMSE
Testing Result With Keras Matrix-Factorization with GD: 35.2665 MPAE
Train on 1574827 samples, validate on 174981 samples
Epoch 1/3
1574827/1574827 [==============================] - 16s 10us/step - loss: 3.2842 - val_loss: 0.8
Epoch 2/3
1574827/1574827 [==============================] - 16s 10us/step - loss: 0.8507 - val_loss: 0.8
Epoch 3/3
1574827/1574827 [==============================] - 16s 10us/step - loss: 0.8428 - val_loss: 0.8
Testing Result With Keras Matrix-Factorization with GD: 0.9131 RMSE
```

```
Testing Result With Keras Matrix-Factorization with GD: 34.3522 MPAE
Train on 1469838 samples, validate on 163316 samples
Epoch 1/3
1469838/1469838 [==============================] - 15s 10us/step - loss: 3.3409 - val_loss: 0.8
Epoch 2/3
1469838/1469838 [==============================] - 15s 10us/step - loss: 0.8518 - val_loss: 0.8
Epoch 3/3
1469838/1469838 [==============================] - 14s 10us/step - loss: 0.8503 - val_loss: 0.8
Testing Result With Keras Matrix-Factorization with GD: 0.9219 RMSE
Testing Result With Keras Matrix-Factorization with GD: 38.5486 MPAE
Train on 1364850 samples, validate on 151651 samples
Epoch 1/3
1364850/1364850 [==============================] - 14s 10us/step - loss: 3.5746 - val_loss: 0.8
Epoch 2/3
1364850/1364850 [==============================] - 14s 10us/step - loss: 0.8513 - val_loss: 0.8
Epoch 3/3
1364850/1364850 [==============================] - 14s 10us/step - loss: 0.8496 - val_loss: 0.8
Testing Result With Keras Matrix-Factorization with GD: 0.9218 RMSE
Testing Result With Keras Matrix-Factorization with GD: 37.2793 MPAE
Train on 1259862 samples, validate on 139985 samples
Epoch 1/3
1259862/1259862 [==============================] - 13s 10us/step - loss: 4.0203 - val_loss: 0.8
Epoch 2/3
1259862/1259862 [==============================] - 12s 10us/step - loss: 0.8507 - val_loss: 0.8
Epoch 3/3
1259862/1259862 [==============================] - 12s 10us/step - loss: 0.8491 - val_loss: 0.8
Testing Result With Keras Matrix-Factorization with GD: 0.9219 RMSE
Testing Result With Keras Matrix-Factorization with GD: 36.4426 MPAE
Train on 1154873 samples, validate on 128320 samples
Epoch 1/3
1154873/1154873 [==============================] - 12s 10us/step - loss: 4.0735 - val_loss: 0.8
Epoch 2/3
1154873/1154873 [==============================] - 12s 10us/step - loss: 0.8505 - val_loss: 0.8
Epoch 3/3
1154873/1154873 [==============================] - 12s 10us/step - loss: 0.8497 - val_loss: 0.8
Testing Result With Keras Matrix-Factorization with GD: 0.9232 RMSE
Testing Result With Keras Matrix-Factorization with GD: 36.2077 MPAE
Train on 1049885 samples, validate on 116654 samples
Epoch 1/3
1049885/1049885 [==============================] - 11s 10us/step - loss: 4.5395 - val_loss: 0.8
Epoch 2/3
1049885/1049885 [==============================] - 11s 10us/step - loss: 0.8506 - val_loss: 0.8
Epoch 3/3
1049885/1049885 [==============================] - 11s 10us/step - loss: 0.8494 - val_loss: 0.8
Testing Result With Keras Matrix-Factorization with GD: 0.9223 RMSE
Testing Result With Keras Matrix-Factorization with GD: 37.4254 MPAE
```

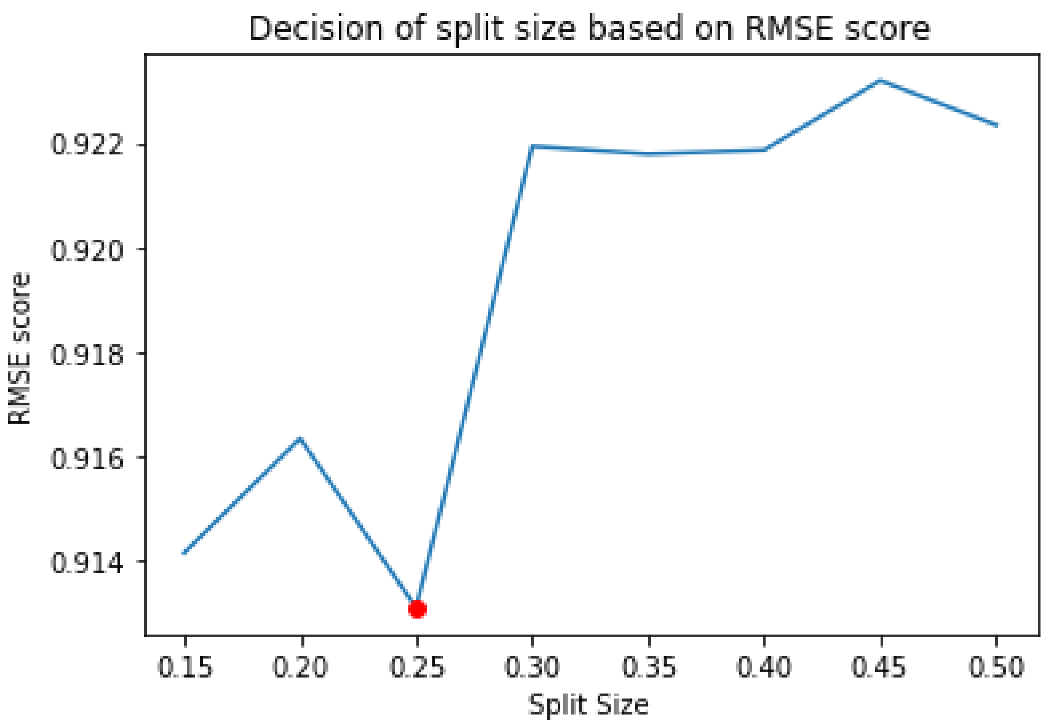
In [42]: rmse

Out[42]:
```
[0.9141546451633403,
 0.9163425707111188,
 0.913105159878866 3,
 0.9219311475353017,
 0.921783036387291,
 0.9218576314126754,
 0.9231886009293375,
 0.9223373205435946]
```

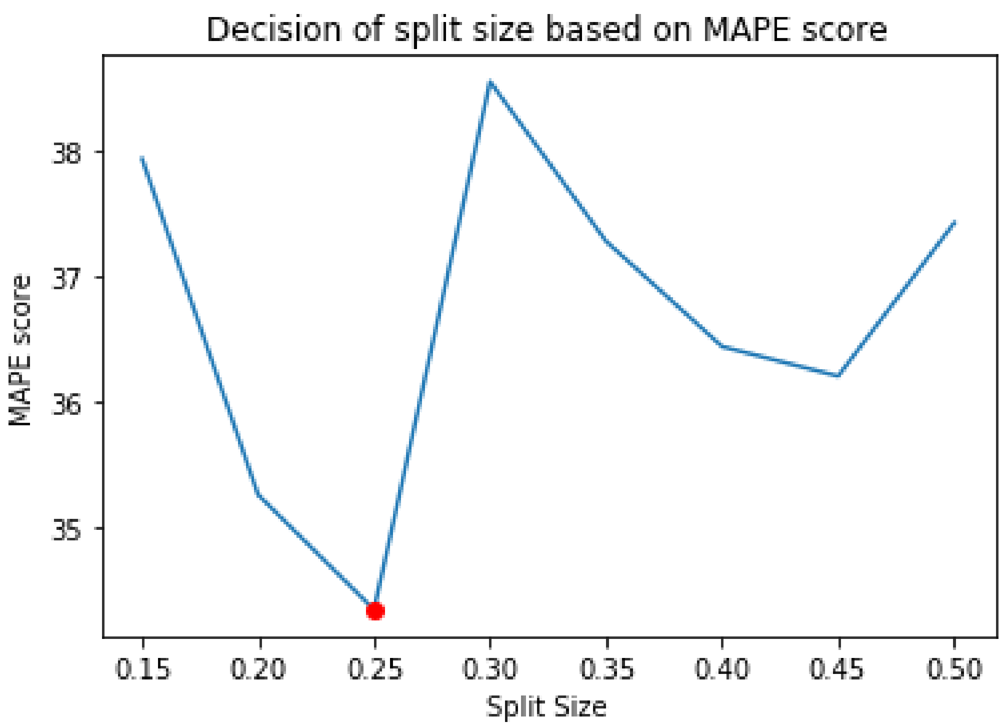
In [43]: mape

Out[43]:
```
[37.93513554291191,
 35.26653715996028,
 34.352241838494926,
 38.54861801231026,
 37.2793313106231,
 36.442601180148124,
 36.207712349922076,
 37.425393003026265]
```

In [44]:
```
plt.plot(split_ratios,rmse)
plt.xlabel('Split Size')
plt.ylabel('RMSE score')
plt.title('Decision of split size based on RMSE score')
plt.plot(split_ratios[2],rmse[2],'ro')
plt.savefig('05_spliting_decision_rmse.png')
plt.show()
```



```
In [45]: plt.plot(split_ratios,mape)
         plt.xlabel('Split Size')
         plt.ylabel('MAPE score')
         plt.title('Decision of split size based on MAPE score')
         plt.plot(split_ratios[2],mape[2],'ro')
         plt.savefig('06_spliting_decision_mape.png')
         plt.show()
```



1.2.18 Splitting Data with split size 25%

```
In [21]: n = int(len(df_filterd) * 0.25)
         df_train = df_filterd[:-n]
         df_test = df_filterd[-n:]

In [22]: df_train.shape

Out[22]: (1749808, 3)

In [23]: df_train.head()

Out[23]:    MovieID   CustID  Ratings
         0     2212  1887046        4
```

```
1      1810   2040484        3
2      2848   2552315        5
3      1877    838612        5
4       191    476765        5
```

In [24]:
```python
df_test.shape
```

Out[24]:
```
(583269, 3)
```

In [25]:
```python
df_test.head()
```

Out[25]:
```
         MovieID   CustID  Ratings
1749808     2866    44490        2
1749809      413  1089335        3
1749810     1202    82700        3
1749811     1467   419238        4
1749812     1918   800683        2
```

1.2.19 Generating sparse matrix for User - Movie

In [26]:
```python
df_p = df_train.pivot_table(index='CustID', columns='MovieID', values='Ratings')
```

In [27]:
```python
df_p.shape
```

Out[27]:
```
(12011, 378)
```

In [28]:
```python
df_p.head()
```

Out[28]:
```
MovieID  28    30    58    83    108   111   118   143   148   175   ...  \
CustID                                                               ...
769       NaN   NaN   NaN   NaN   NaN   NaN   NaN   4.0   NaN   5.0  ...
1333      NaN   NaN   NaN   3.0   NaN   2.0   NaN   NaN   2.0   3.0  ...
1442      4.0   NaN   NaN   NaN   NaN   NaN   5.0   5.0   NaN   NaN  ...
2213      5.0   NaN   5.0   3.0   NaN   NaN   4.0   NaN   NaN   NaN  ...
2455      NaN   4.0   NaN   NaN   NaN   NaN   4.0   NaN   NaN   NaN  ...

MovieID  4389  4392  4393  4402  4418  4420  4432  4472  4479  4488
CustID
769       NaN   3.0   NaN   NaN   NaN   NaN   3.0   NaN   NaN   NaN
1333      1.0   2.0   NaN   NaN   NaN   NaN   2.0   5.0   NaN   NaN
1442      NaN   NaN   4.0   NaN   NaN   NaN   NaN   4.0   NaN   NaN
2213      NaN   4.0   NaN   NaN   5.0   NaN   5.0   4.0   4.0   4.0
2455      NaN   NaN   3.0   2.0   NaN   NaN   4.0   3.0   NaN   NaN

[5 rows x 378 columns]
```

1.3 Content Based Filtering Techniques

1.3.1 Mean Rating : biased and favours movies with fewer ratings, since large numbers of ratings tend to be less extreme in its mean ratings.

```
In [29]: n = 10 # top 10 movies
         all_movie_mean = df_p.mean(axis=0).sort_values(ascending=False).rename('Rating-Mean')
         all_movie_count = df_p.count(axis=0).rename('Rating-Count').to_frame()
         ranking_mean_rating = all_movie_mean.head(n).join(all_movie_count).join(movie_titles.
         df_prediction = df_test.set_index('MovieID').join(all_movie_mean)[['Ratings', 'Rating-
         y_true = df_prediction['Ratings']
         y_pred = df_prediction['Rating-Mean']
         mean_rating_rmse = np.sqrt(mean_squared_error(y_true=y_true, y_pred=y_pred))
         mean_rating_mape = np.mean(abs((y_true - y_pred)/y_true))*100
```

```
In [30]: mean_df = pd.concat([y_true,y_pred],axis = 1)
         mean_df.head()
```

```
Out[30]:          Ratings  Rating-Mean
         MovieID
         28             4     3.730938
         28             2     3.730938
         28             5     3.730938
         28             3     3.730938
         28             3     3.730938
```

```
In [31]: mean_rating_rmse
```

```
Out[31]: 0.9968725805283982
```

```
In [32]: mean_rating_mape
```

```
Out[32]: 31.320257668022638
```

```
In [33]: ranking_mean_rating
```

```
Out[33]:          Rating-Mean  Rating-Count  \
         MovieID
         2452        4.429901          8381
         3962        4.363821          7861
         2172        4.359435          3116
         3046        4.334108          2583
         4306        4.329482          8568
         2862        4.324941          8460
         3290        4.313850          6296
         2162        4.310345          2378
         3864        4.257969          4706
         2782        4.252250          8222

                                                       Name
```

```
MovieID
2452     Lord of the Rings: The Fellowship of the Ring
3962                           Finding Nemo (Widescreen)
2172                              The Simpsons: Season 3
3046                   The Simpsons: Treehouse of Horror
4306                                     The Sixth Sense
2862                            The Silence of the Lambs
3290                                       The Godfather
2162                                       CSI: Season 1
3864                                       Batman Begins
2782                                          Braveheart
```

1.3.2 Weighted Mean Rating : Many good ratings outweigh few in this algorithm

In [34]:
```
m = 1000
C = df_p.stack().mean()
R = df_p.mean(axis=0).values
v = df_p.count().values
weighted_score = (v/ (v+m) *R) + (m/ (v+m) *C)
weighted_ranking = np.argsort(weighted_score)[::-1]
weighted_score = np.sort(weighted_score)[::-1]
weighted_movie_ids = df_p.columns[weighted_ranking]
df_prediction = df_test.set_index('MovieID').join(pd.DataFrame(weighted_score, index=w
y_true = df_prediction['Ratings']
y_pred = df_prediction['Prediction']
weighted_mean_rmse = np.sqrt(mean_squared_error(y_true=y_true, y_pred=y_pred))
weighted_mean_mape = np.mean(abs((y_true - y_pred)/y_true))*100
```

In [35]:
```
weighted_mean_df = pd.concat([y_true,y_pred],axis = 1)
weighted_mean_df.head()
```

Out[35]:

MovieID	Ratings	Prediction
28	4	3.693646
28	2	3.693646
28	5	3.693646
28	3	3.693646
28	3	3.693646

In [36]:
```
weighted_mean_rmse
```

Out[36]: 0.9989090510996999

In [37]:
```
weighted_mean_mape
```

Out[37]: 31.70895473055578

In [38]:
```
df_plot = pd.DataFrame(weighted_score[:n], columns=['Rating'])
df_plot.index = weighted_movie_ids[:10]
ranking_weighted_rating = df_plot.join(all_movie_count).join(movie_titles)
ranking_weighted_rating
```

```
Out[38]:             Rating  Rating-Count    Year  \
         MovieID
         2452      4.333274          8381  2001.0
         3962      4.268981          7861  2003.0
         4306      4.245239          8568  1999.0
         2862      4.240216          8460  1991.0
         3290      4.205516          6296  1974.0
         2782      4.173221          8222  1995.0
         2172      4.156328          3116  1991.0
         3864      4.129240          4706  2005.0
         3046      4.107855          2583  1990.0
         1905      4.099521          8520  2003.0

                                                          Name
         MovieID
         2452         Lord of the Rings: The Fellowship of the Ring
         3962                             Finding Nemo (Widescreen)
         4306                                       The Sixth Sense
         2862                              The Silence of the Lambs
         3290                                         The Godfather
         2782                                            Braveheart
         2172                                The Simpsons: Season 3
         3864                                         Batman Begins
         3046                       The Simpsons: Treehouse of Horror
         1905      Pirates of the Caribbean: The Curse of the Bla...
```

2 Colaborative Filtering Techniques

2.1 Memory based filtering techniques

2.1.1 Cosine Similarity User - User

```
In [39]: def cosine_similarity(given_df):
             user_index = 0
             n_recommendation = 15
             n_plot = 10
             df_p_imputed = df_p.T.fillna(df_p.mean(axis=1)).T
             similarity = cosine_similarity(df_p_imputed.values)
             similarity -= np.eye(similarity.shape[0])
             similar_user_index = np.argsort(similarity[user_index])[::-1]
             similar_user_score = np.sort(similarity[user_index])[::-1]
             unrated_movies = df_p.iloc[user_index][df_p.iloc[user_index].isna()].index
             mean_movie_recommendations = (df_p_imputed.iloc[similar_user_index[:n_recommendati
             best_movie_recommendations = mean_movie_recommendations[unrated_movies].sort_value
             user_id_mapping = {id:i for i, id in enumerate(df_p_imputed.index)}
             prediction = []

             for user_id in given_df['CustID'].unique():
```

```
            similar_user_index = np.argsort(similarity[user_id_mapping[user_id]])[::-1]
            similar_user_score = np.sort(similarity[user_id_mapping[user_id]])[::-1]
            for movie_id in df_test[df_test['CustID']==user_id]['MovieID'].values:
                score = (df_p_imputed.iloc[similar_user_index[:n_recommendation]][movie_i
                prediction.append([user_id, movie_id, score])
        df_pred = pd.DataFrame(prediction, columns=['CustID', 'MovieID', 'Prediction']).se
        df_pred = given_df.set_index(['CustID', 'MovieID']).join(df_pred)
        y_true = df_pred['Ratings'].values
        y_pred = df_pred['Prediction'].values
        cosine_similarity_rmse = np.sqrt(mean_squared_error(y_true=y_true, y_pred=y_pred)
        cosine_similarity_mape = np.mean(abs((y_true - y_pred)/y_true))*100
        return cosine_similarity_rmse,cosine_similarity_mape
```

In [ ]:
```
cosine_train_rmse, cosine_train_mape = cosine_similarity(df_train)
```

In [ ]:
```
cosine_test_rmse, cosine_test_mape = cosine_similarity(df_test)
```

In [57]:
```
cosine_train_rmse
```

Out[57]: 1.2103247637799341

In [58]:
```
cosine_train_mape
```

Out[58]: 41.92920389214481

In [59]:
```
cosine_test_rmse
```

Out[59]: 1.2330776623975954

In [60]:
```
cosine_test_mape
```

Out[60]: 42.70021189615626

2.1.2 For UserID 1508633

In [17]:
```
cust1508633 = df_test[df_test['CustID'] == 1508633]
cust1508633
```

Out[17]:

	MovieID	CustID	Ratings
1755200	3463	1508633	3
1764883	4432	1508633	4
1767226	2874	1508633	3
1773216	3756	1508633	3
1775980	4315	1508633	4
1776026	2192	1508633	3
1794071	692	1508633	3
1836439	3624	1508633	4
1837571	2391	1508633	3
1857338	3239	1508633	4
1861279	3544	1508633	3

```
1865485     4306  1508633        4
1867349     2580  1508633        3
1904907     1810  1508633        3
1908613     1425  1508633        3
1918869     2171  1508633        3
1937310     1180  1508633        3
1944736     2862  1508633        3
1948289     1467  1508633        3
1948787      896  1508633        3
1948893     2660  1508633        3
1958547      353  1508633        3
2000504     2499  1508633        3
2015047     4364  1508633        3
2055192     1428  1508633        3
2062806     3222  1508633        3
2063776     4341  1508633        3
2069489     2699  1508633        3
2069862     1202  1508633        2
2078372     4330  1508633        4
2082395     2913  1508633        4
2083894     3282  1508633        3
2114812     3522  1508633        3
2117219     3605  1508633        3
2117739     2009  1508633        3
2117754     4089  1508633        2
2124709     4302  1508633        4
2164026     3148  1508633        4
2170461     1861  1508633        2
2171246      143  1508633        4
2175321     2675  1508633        2
2180564      273  1508633        4
2190787      851  1508633        3
2204379     1470  1508633        4
2207715     1267  1508633        3
2215462     2186  1508633        3
2227640      357  1508633        3
2230419      550  1508633        3
2252413     2779  1508633        3
2263722      191  1508633        3
2318884     2782  1508633        3
2320109     1110  1508633        4
2323448      299  1508633        3
```

In [ ]:
```python
prediction1508633 = []
for movie_id in df_test[df_test['CustID']== 1508633]['MovieID'].values:
    # Compute predicted score
    score1508633 = (df_p_imputed.iloc[similar_user_index[:n_recommendation]][movie_id]
    prediction1508633.append([user_id, movie_id, score])
```

```
df_pred1508633 = pd.DataFrame(prediction1508633, columns=['CustID', 'MovieID', 'Predic
df_pred1508633 = cust1508633.set_index('MovieID').join(df_pred1508633,rsuffix='_right')
df_pred1508633 = df_pred1508633.drop(columns = ['CustID_right'] )
```

```
In [ ]: df_pred1508633
```

2.1.3 Movie Based Pearsons' R correlations : measure the linear correlation between review scores of all pairs of movies, then we provide the top 10 movies with highest correlation

```
In [40]: f = ['count','mean']
         df_movie_summary = df_train.groupby('MovieID')['Ratings'].agg(f)
         df_movie_summary.index = df_movie_summary.index.map(int)
         def recommend(movie_title, min_count):
             print("For movie ({})".format(movie_title))
             print("- Top 10 movies recommended based on Pearsons'R correlation - ")
             i = int(movie_titles.index[movie_titles['Name'] == movie_title][0])
             target = df_p[i]
             similar_to_target = df_p.corrwith(target)
             corr_target = pd.DataFrame(similar_to_target, columns = ['PearsonR'])
             corr_target.dropna(inplace = True)
             corr_target = corr_target.sort_values('PearsonR', ascending = False)
             corr_target.index = corr_target.index.map(int)
             corr_target = corr_target.join(movie_titles).join(df_movie_summary)[['PearsonR',
             print(corr_target[corr_target['count']>min_count][:10].to_string(index=False))
```

```
In [41]: recommend('The Mummy',0)
```

```
For movie (The Mummy)
- Top 10 movies recommended based on Pearsons'R correlation -
 PearsonR               Name  count      mean
 1.000000          The Mummy   7577  3.641283
 0.409515           Stargate   4583  3.762819
 0.390881  The Scorpion King   4771  3.327604
 0.385829    Men in Black II   7530  3.119522
 0.384842        Dragonheart   4129  3.501574
 0.384099        End of Days   5387  2.913310
 0.378859     Chain Reaction   3361  3.010414
 0.373285        Dante's Peak  4400  3.071136
 0.370338  Jurassic Park III   6044  3.131866
 0.368546         Hollow Man   6454  2.786644
```

```
In [42]: recommend('The Scorpion King',0)
```

```
For movie (The Scorpion King)
- Top 10 movies recommended based on Pearsons'R correlation -
 PearsonR                     Name  count      mean
 1.000000        The Scorpion King   4771  3.327604
 0.435755              The Rundown   4707  3.613342
```

```
0.429796        Blade: Trinity   4140  3.526812
0.422922                 I Spy   3640  2.997253
0.418851        Chain Reaction   3361  3.010414
0.418042  Domestic Disturbance   3540  3.318079
0.414334     Collateral Damage   5337  2.861907
0.413242           The Pacifier   3110  3.508039
0.410274           End of Days   5387  2.913310
0.409484           Exit Wounds   3610  2.885042
```

2.2 Model based techniques

2.2.1 Matrix Factorization using SVD

In [22]:
```
reader = Reader()

# get all rows for faster run time
data = Dataset.load_from_df(df_train[['CustID', 'MovieID', 'Ratings']][:], reader)
data.split(n_folds=3)

svd = SVD()
evaluate(svd, data, measures=['RMSE','MAE'])
```

```
/home/harshit/.local/lib/python3.6/site-packages/surprise/evaluate.py:66: UserWarning: The eval
  'model_selection.cross_validate() instead.', UserWarning)
/home/harshit/.local/lib/python3.6/site-packages/surprise/dataset.py:193: UserWarning: Using da
  UserWarning)


Evaluating RMSE, MAE of algorithm SVD.

------------
Fold 1
RMSE: 0.8458
MAE:  0.6605
------------
Fold 2
RMSE: 0.8451
MAE:  0.6603
------------
Fold 3
RMSE: 0.8446
MAE:  0.6593
------------
------------
Mean RMSE: 0.8452
Mean MAE : 0.6601
------------
```

```
Out[22]: CaseInsensitiveDefaultDict(list,
                                    {'rmse': [0.8458433749310716,
                                      0.845112174965906,
                                      0.8446200318698301],
                                     'mae': [0.6605361852163709,
                                      0.6603388076393246,
                                      0.6592977182091926]})
```

2.2.2 Matrix Factorization using gradient descent and keras

```
In [44]: def mat_factorization_gradient():
             user_id_mapping = {id:i for i, id in enumerate(df_filterd['CustID'].unique())}
             movie_id_mapping = {id:i for i, id in enumerate(df_filterd['MovieID'].unique())}

             # Create correctly mapped train- & testset
             train_user_data = df_train['CustID'].map(user_id_mapping)
             train_movie_data = df_train['MovieID'].map(movie_id_mapping)

             test_user_data = df_test['CustID'].map(user_id_mapping)
             test_movie_data = df_test['MovieID'].map(movie_id_mapping)


             # Get input variable-sizes
             users = len(user_id_mapping)
             movies = len(movie_id_mapping)
             embedding_size = 10


             ##### Create model
             # Set input layers
             user_id_input = Input(shape=[1], name='user')
             movie_id_input = Input(shape=[1], name='movie')

             # Create embedding layers for users and movies
             user_embedding = Embedding(output_dim=embedding_size,
                                        input_dim=users,
                                        input_length=1,
                                        name='user_embedding')(user_id_input)
             movie_embedding = Embedding(output_dim=embedding_size,
                                         input_dim=movies,
                                         input_length=1,
                                         name='item_embedding')(movie_id_input)

             # Reshape the embedding layers
             user_vector = Reshape([embedding_size])(user_embedding)
```

```
    movie_vector = Reshape([embedding_size])(movie_embedding)

    # Compute dot-product of reshaped embedding layers as prediction
    y = Dot(1, normalize=False)([user_vector, movie_vector])

    # Setup model
    model = Model(inputs=[user_id_input, movie_id_input], outputs=y)
    model.compile(loss='mse', optimizer='adam')


    # Fit model
    model.fit([train_user_data, train_movie_data],
              df_train['Ratings'],
              batch_size=256,
              epochs=5,
              validation_split=0.1,
              shuffle=True)

    # Test model
    y_pred_train = model.predict([train_user_data, train_movie_data])
    y_true_train = df_train['Ratings'].values

    # Test model
    y_pred = model.predict([test_user_data, test_movie_data])
    y_true = df_test['Ratings'].values


    #  Compute RMSE
    mf_gd_rmse_train = np.sqrt(mean_squared_error(y_pred=y_pred_train, y_true=y_true_
    mf_gd_mape_train = np.mean(abs((y_true_train[:1000] - y_pred_train[:1000])/y_true_
    print('Training RMSE & MAPE')
    print('Training Result With MF with GD and Keras: {:.4f} RMSE'.format(mf_gd_rmse_
    print('Training Result With MF with GD and Keras: {:.4f} MPAE'.format(mf_gd_mape_

    #  Compute RMSE
    print('Testing RMSE & MAPE')
    mf_gd_rmse_test = np.sqrt(mean_squared_error(y_pred=y_pred, y_true=y_true))
    mf_gd_mape_test = np.mean(abs((y_true[:1000] - y_pred[:1000])/y_true[:1000]))*100
    print('Testing Result With Keras Matrix-Factorization: {:.4f} RMSE'.format(mf_gd_
    print('Testing Result With Keras Matrix-Factorization: {:.4f} MPAE'.format(mf_gd_
    return mf_gd_rmse_train,mf_gd_mape_train,mf_gd_rmse_test,mf_gd_mape_test
```

In [45]:
```
mf_gd_rmse_train,mf_gd_mape_train,mf_gd_rmse_test,mf_gd_mape_test = mat_factorization
```

```
WARNING:tensorflow:From /home/harshit/.local/lib/python3.6/site-packages/tensorflow/python/fram
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From /home/harshit/.local/lib/python3.6/site-packages/tensorflow/python/ops/
```

```
Instructions for updating:
Use tf.cast instead.
Train on 1574827 samples, validate on 174981 samples
Epoch 1/5
1574827/1574827 [==============================] - 17s 11us/step - loss: 3.2986 - val_loss: 0.8
Epoch 2/5
1574827/1574827 [==============================] - 15s 10us/step - loss: 0.8515 - val_loss: 0.8
Epoch 3/5
1574827/1574827 [==============================] - 16s 10us/step - loss: 0.8491 - val_loss: 0.8
Epoch 4/5
1574827/1574827 [==============================] - 15s 10us/step - loss: 0.8386 - val_loss: 0.8
Epoch 5/5
1574827/1574827 [==============================] - 15s 10us/step - loss: 0.8030 - val_loss: 0.7
Training RMSE & MAPE
Training Result With MF with GD and Keras: 0.8757 RMSE
Training Result With MF with GD and Keras: 32.2090 MPAE
Testing RMSE & MAPE
Testing Result With Keras Matrix-Factorization: 0.8874 RMSE
Testing Result With Keras Matrix-Factorization: 35.5629 MPAE
```

In [46]:
```
mf_gd_rmse_train
```

Out[46]: 0.8757196066479144

In [47]:
```
mf_gd_mape_train
```

Out[47]: 32.20895914219936

In [48]:
```
mf_gd_rmse_test
```

Out[48]: 0.8874479549344085

In [49]:
```
mf_gd_mape_test
```

Out[49]: 35.562869051599506

2.2.3 Deep Learning model using keras

In [50]:
```
def deep_learning_model():
    user_embedding_size = 20
    movie_embedding_size = 10
    user_id_mapping = {id:i for i, id in enumerate(df_filterd['CustID'].unique())}
    movie_id_mapping = {id:i for i, id in enumerate(df_filterd['MovieID'].unique())}
    users = len(user_id_mapping)
    movies = len(movie_id_mapping)
    user_id_input = Input(shape=[1], name='user')
    movie_id_input = Input(shape=[1], name='movie')
    user_embedding = Embedding(output_dim=user_embedding_size,
```

```
                                 input_dim=users,
                                 input_length=1,
                                 name='user_embedding')(user_id_input)
        movie_embedding = Embedding(output_dim=movie_embedding_size,
                                  input_dim=movies,
                                  input_length=1,
                                  name='item_embedding')(movie_id_input)
        train_user_data = df_train['CustID'].map(user_id_mapping)
        train_movie_data = df_train['MovieID'].map(movie_id_mapping)
        test_user_data = df_test['CustID'].map(user_id_mapping)
        test_movie_data = df_test['MovieID'].map(movie_id_mapping)
        user_vector = Reshape([user_embedding_size])(user_embedding)
        movie_vector = Reshape([movie_embedding_size])(movie_embedding)
        concat = Concatenate()([user_vector, movie_vector])
        dense = Dense(256)(concat)
        y = Dense(1)(dense)
        model = Model(inputs=[user_id_input, movie_id_input], outputs=y)
        model.compile(loss='mse', optimizer='adam')
        model.fit([train_user_data, train_movie_data],
                  df_train['Ratings'],
                  batch_size=256,
                  epochs=5,
                  validation_split=0.1,
                  shuffle=True)
        y_pred_train = model.predict([train_user_data, train_movie_data])
        y_true_train = df_train['Ratings'].values
        y_pred_test = model.predict([test_user_data, test_movie_data])
        y_true_test = df_test['Ratings'].values
        dl_rmse_train = np.sqrt(mean_squared_error(y_pred=y_pred_train, y_true=y_true_tra
        dl_mape_train = np.mean(abs((y_true_train[:1000] - y_pred_train[:1000])/y_true_tra
        print('Training Result With Keras Deep Learning: {:.4f} RMSE'.format(dl_rmse_trai
        print('Training Result With Keras Deep Learning: {:.4f} MPAE'.format(dl_mape_trai
        dl_rmse_test = np.sqrt(mean_squared_error(y_pred=y_pred_test, y_true=y_true_test)
        dl_mape_test = np.mean(abs((y_true_test[:1000] - y_pred_test[:1000])/y_true_test[
        print('Testing Result With Keras Deep Learning: {:.4f} RMSE'.format(dl_rmse_test)
        print('Testing Result With Keras Deep Learning: {:.4f} MPAE'.format(dl_mape_test)
        return dl_rmse_train,dl_mape_train,dl_rmse_test,dl_mape_test
```

In [51]:
```
dl_rmse_train,dl_mape_train,dl_rmse_test,dl_mape_test = deep_learning_model()
```

```
Train on 1574827 samples, validate on 174981 samples
Epoch 1/5
1574827/1574827 [==============================] - 26s 16us/step - loss: 0.9171 - val_loss: 0.8
Epoch 2/5
1574827/1574827 [==============================] - 25s 16us/step - loss: 0.8294 - val_loss: 0.8
Epoch 3/5
1574827/1574827 [==============================] - 25s 16us/step - loss: 0.8248 - val_loss: 0.8
Epoch 4/5
```

```
1574827/1574827 [==============================] - 25s 16us/step - loss: 0.8221 - val_loss: 0.8
Epoch 5/5
1574827/1574827 [==============================] - 25s 16us/step - loss: 0.8201 - val_loss: 0.8
Training Result With Keras Deep Learning: 0.8996 RMSE
Training Result With Keras Deep Learning: 31.9017 MPAE
Testing Result With Keras Deep Learning: 0.9070 RMSE
Testing Result With Keras Deep Learning: 35.1764 MPAE
```

In [52]:
```
dl_rmse_train
```

Out[52]:
```
0.8996414552349238
```

In [53]:
```
dl_mape_train
```

Out[53]:
```
31.901738270246994
```

In [54]:
```
dl_rmse_test
```

Out[54]:
```
0.9069663213662184
```

In [55]:
```
dl_mape_test
```

Out[55]:
```
35.17644097804546
```

In [73]:
```
methods = ['Mean Rating', 'Weighted Mean Rating','Cosine Similarity','Matrix Factoriza
rmse = pd.Series(np.array([mean_rating_rmse,weighted_mean_rmse,cosine_test_rmse,mf_svd
mape = pd.Series(np.array([mean_rating_mape,weighted_mean_mape,cosine_test_mape,mf_svd
error_df = pd.concat([rmse,mape],axis = 1)
error_df.columns = ['RMSE','MAPE']
#error_df.set_index(methods, inplace = True)
error_df
```

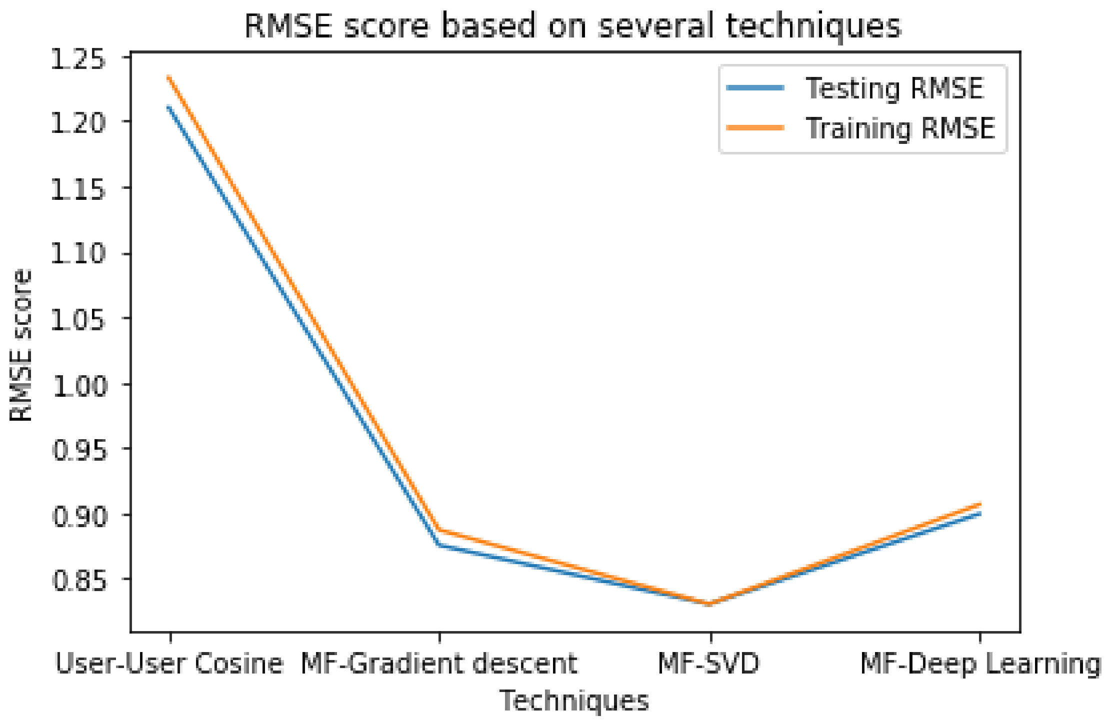
Out[73]:

	RMSE	MAPE
Mean Rating	0.996873	31.320258
Weighted Mean Rating	0.998909	31.708955
Cosine Similarity	1.233078	42.700212
Matrix Factorization with SVD	0.845200	66.010000
Matrix Factorization with Keras & GD	0.887448	35.562869
Deep Learning Dense Model with Keras	0.906966	35.176441

2.2.4 RMSE comparisions of various colaborative techniques

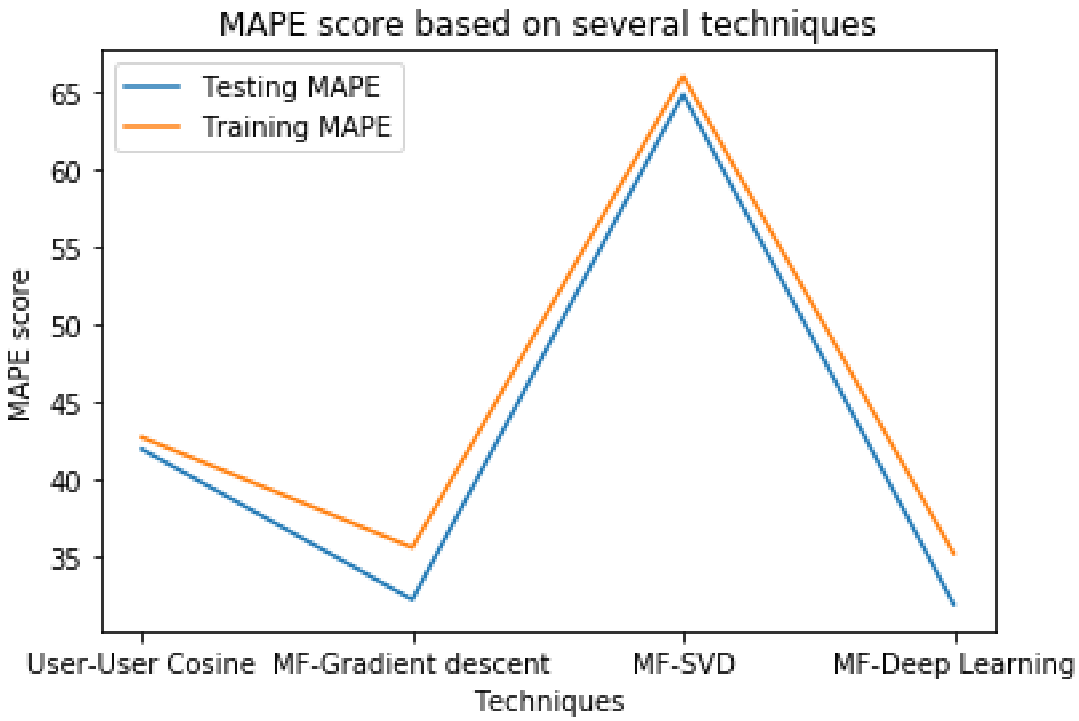
In [64]:
```
columns = ["User-User Cosine","MF-Gradient descent","MF-SVD","MF-Deep Learning"]
training_result_rmse = [cosine_train_rmse,mf_gd_rmse_train,mf_svd_train_rmse ,dl_rmse_
testing_result_rmse = [cosine_test_rmse,mf_gd_rmse_test,mf_svd_train_rmse ,dl_rmse_tes
plt.plot(columns,training_result_rmse,label='Testing RMSE')
plt.plot(columns,testing_result_rmse,label = 'Training RMSE')
plt.xlabel('Techniques')
plt.ylabel('RMSE score')
```

```
plt.title('RMSE score based on several techniques')
plt.legend()
#plt.plot(split_ratios[2],rmse[2],'ro')
plt.savefig('07_rmse_overall.png')
```



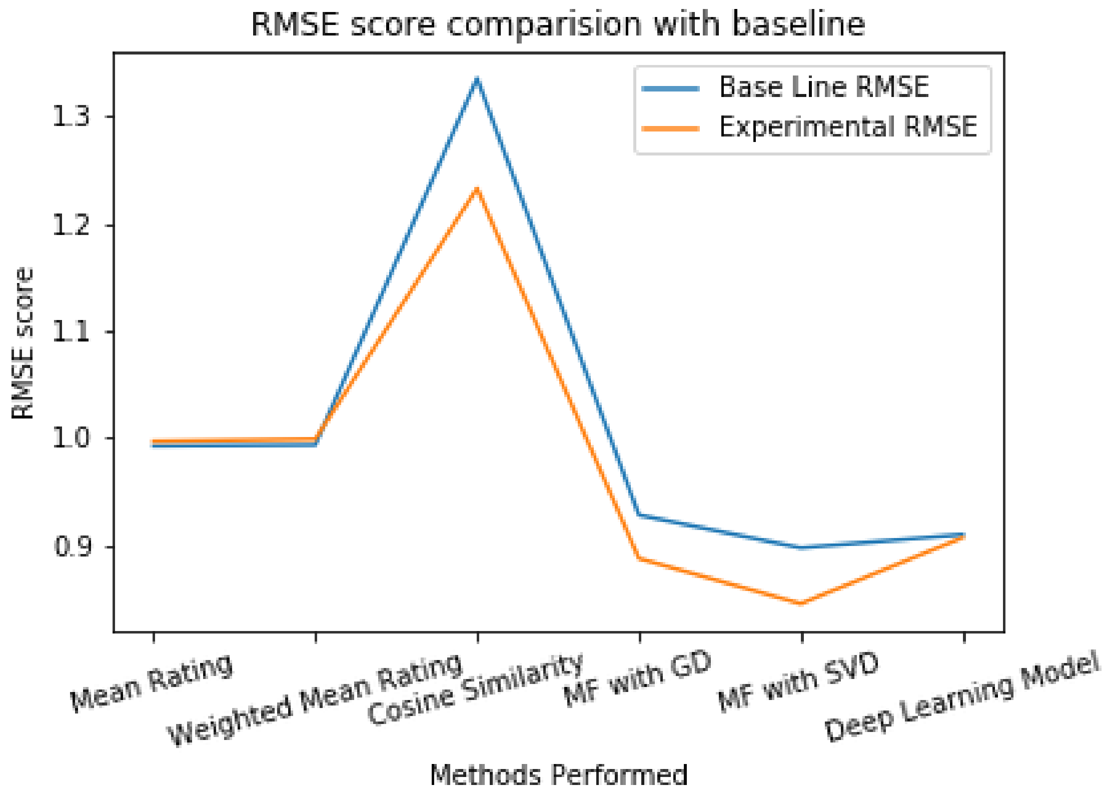
2.2.5 MAPE comparisions of various colaborative techniques

```
In [65]: training_result_mape = [cosine_train_mape,mf_gd_mape_train,mf_svd_train_mape ,dl_mape_
         testing_result_mape = [cosine_test_mape,mf_gd_mape_test,mf_svd_test_mape ,dl_mape_tes
         plt.plot(columns,training_result_mape,label='Testing MAPE')
         plt.plot(columns,testing_result_mape,label = 'Training MAPE')
         plt.xlabel('Techniques')
         plt.ylabel('MAPE score')
         plt.title('MAPE score based on several techniques')
         plt.legend()
         #plt.plot(split_ratios[2],rmse[2],'ro')
         plt.savefig('08_mape_overall.png')
```



2.2.6 Comparision experimental RMSE with baseline results

In [68]:
```
baseline = [0.9925,0.9935,1.3357,0.9280,0.8974,0.9098]
our = [mean_rating_rmse,weighted_mean_rmse,cosine_test_rmse,mf_gd_rmse_test,mf_svd_tes
columns = ["Mean Rating","Weighted Mean Rating","Cosine Similarity","MF with GD","MF w
plt.plot(columns,baseline,label='Base Line RMSE')
plt.plot(columns,our,label = 'Experimental RMSE')
plt.xlabel('Methods Performed')
plt.ylabel('RMSE score')
plt.title('RMSE score comparision with baseline')
plt.legend()
plt.xticks(rotation = 15)
#plt.plot(split_ratios[2],rmse[2],'ro')
plt.savefig('rmse_baseline_vs_exp.png')
```


RMSE score comparision with baseline
Base Line RMSE
Experimental RMSE
RMSE score
1.3
1.2
1.1
1.0
0.9
Mean Rating
Weighted Mean Rating
Cosine Similarity
MF with GD
MF with SVD
Deep Learning Model
Methods Performed

```
In [ ]:
```