```
import os
import numpy as np
import pandas as pd

import random

import seaborn as sns
#import plotly.express as px
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from sklearn.metrics import euclidean_distances
from scipy.spatial.distance import cdist
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.metrics import mean_squared_error

import warnings
warnings.filterwarnings("ignore")
```

Import data set.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
    Mounted at /content/drive
```

```
df_count=pd.read_csv('/content/drive/MyDrive/Capstone Project/count_data.csv')
```

```
df_song=pd.read_csv('/content/drive/MyDrive/Capstone Project/song_data.csv')
```

Data preprocessing.

```
df_count.head()
```

	Unnamed: 0	user_id	song_id	play_count
0	0	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOAKIMP12A8C130995	1
1	1	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBBMDR12A8C13253B	2
2	2	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBXHDL12A81C204C0	1
3	3	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBYHAJ12A6701BF1D	1

```
df_song.head()
```

	song_id	title	release	artist_name	year
0	SOQMMHC12AB0180CB8	Silent Night	Monster Ballads X-Mas	Faster Pussy cat	2003
1	SOVFVAK12A8C1350D9	Tanssi vaan	Karkuteillä	Karkkiautomaatti	1995
2	SOGTUKN12AB017F4F1	No One Could Ever	Butter	Hudson Mohawke	2006
3	SOBNYVR12A8C13558C	Si Vos Querés	De Culo	Yerba Brava	2003
4	SOHSBXH12A8C13B0DF	Tangle Of	Rene Ablaze Presents Winter	Der Mystic	0

```
#delete the first column of df_count
df_count=df_count.iloc[: , 1:]
```

```
df_count.head()
```

	user_id	song_id	play_count
0	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOAKIMP12A8C130995	1
1	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBBMDR12A8C13253B	2
2	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBXHDL12A81C204C0	1
3	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBYHAJ12A6701BF1D	1
4	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SODACBL12A8C13C273	1

```
df_count['user_id'].count()
```

```
2000000
```

Understanding the data sets.

Next, check the length of song_id in both datasets.

```
df_count['song_id'].count()
```

```
2000000
```

```
df_count.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000000 entries, 0 to 1999999
Data columns (total 3 columns):
 #   Column      Dtype
---  ------      -----
 0   user_id     object
 1   song_id     object
 2   play_count  int64
dtypes: int64(1), object(2)
memory usage: 45.8+ MB
```

```
df_song['song_id'].count()
```

```
1000000
```

```
df_song.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 5 columns):
 #   Column       Non-Null Count    Dtype
---  ------       --------------    -----
 0   song_id      1000000 non-null  object
 1   title        999985 non-null   object
 2   release      999995 non-null   object
 3   artist_name  1000000 non-null  object
 4   year         1000000 non-null  int64
dtypes: int64(1), object(4)
memory usage: 38.1+ MB
```

```
df_count['play_count'].astype(np.int32,copy=True)
```

```
0          1
1          2
2          1
3          1
4          1
          ..
1999995    2
1999996    4
1999997    3
1999998    1
1999999    1
Name: play_count, Length: 2000000, dtype: int32
```

```
df_count.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000000 entries, 0 to 1999999
Data columns (total 3 columns):
 #   Column      Dtype 
---  ------      ----- 
 0   user_id     object
 1   song_id     object
 2   play_count  int64 
dtypes: int64(1), object(2)
memory usage: 45.8+ MB
```

There are 2000000 'song_id' in df_count, but 1000000 'song_id' in df_song. So there are at least 2000000-1000000 songs without information about 'title', 'Release', 'Artist_name' and 'year'.

Check the length of "title" in df_song. Ideally, it should match with the length of "song_id" in the same dataframe.

```
df_song['title'].count()
```

```
999985
```

We have 15 less "titles" than "song_id" in df_song.

Let's check for missing values in each data set.

Find missing values in each column of df_count.

```
# Find number of missing values in each column
df_count.isna().sum()
```

```
user_id       0
song_id       0
play_count    0
dtype: int64
```

There is **no** missing values in df_count.

Find missing values in each column of df_song

```
# Find number of missing values in each column
df_song.isna().sum()
```

```
song_id         0
title          15
```

```
release         5
artist_name     0
year            0
dtype: int64
```

There are missing value in df_song: column "title" has 15 missing values, column "Release" has 5 missing values.

Now, let's check for any duplicates in the two data sets.

```
# Check df_count
print("The number of unique song_id in df_count is", df_count.song_id.nunique())# Count the n
print("The number of unique user_id in df_count is", df_count.user_id.nunique())# # Count the
```

```
The number of unique song_id in df_count is 10000
The number of unique user_id in df_count is 76353
```

```
# Check df_song
print("The number of unique song_id in df_song is", df_song.song_id.nunique())# Count the num
print("The number of unique title in df_song is", df_song.title.nunique())# Count the number
```

```
The number of unique song_id in df_song is 999056
The number of unique title in df_song is 702428
```

Notice the number of unique song_id (999056) is more than the number of unique title (702428).

- One possibility is that the same song was incorrectly marked with different song_id. Another possibility is that different songs have the same title. This is something that can be investigated further.
- Also notice that the number of unique songs in df_count (10000) is significantly smaller than the number of unique songs in df_song (999056). df_count is the only data set that contains information about play_count. Since play_count is a good indicator of the likelihood of a user listening to the song, I will combine data set df_count with df_song. By doing so, there is going to be only 10000 unique songs left in the combine data set. This reduces the size of the data significantly and also reduces sparseness of the interaction matrix. In addition, there is no need to investigate how the number of "song_id" is different from the number of "title".
- However, it is worth noticing that if the recommendation system is based on for example "Artist", then it may not be a good idea to combine the two data sets early on since we will lose a lot of information about the "song_id" and "title". Also, in that recommendation system, we need to understand why the number of unique song_id is different from the number of unique "title".

Preprocessing.

Merge the two data sets using "song_id".

```
df=pd.merge(df_count, df_song.drop_duplicates(['song_id']), on="song_id", how="left")
```

```
df.head(50)
```

	user_id	song_id	play_count	
0	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOAKIMP12A8C130995	1	The
1	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBBMDR12A8C13253B	2	Ent
2	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBXHDL12A81C204C0	1	St
3	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBYHAJ12A6701BF1D	1	Constel
4	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SODACBL12A8C13C273	1	Learn

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2000000 entries, 0 to 1999999
Data columns (total 7 columns):
 #   Column       Dtype
---  ------       -----
 0   user_id      object
 1   song_id      object
 2   play_count   int64
 3   title        object
 4   release      object
 5   artist_name  object
 6   year         int64
dtypes: int64(2), object(5)
memory usage: 122.1+ MB
```

Which one is the most interacted song in the dataset?

```
df['song_id'].value_counts()
```

```
SOFRQTD12A81C233C0    8277
SOAUWYT12A81C206F1    7032
SOAXGDH12A8C13F8A1    6949
SOBONKR12A58A7A7E0    6412
SOSXLTC12AF72A7F54    6145
                      ...
SOWNLZF12A58A79811      51
SOLIGVL12AB017DBAE      51
SOBPGWB12A6D4F7EF3      50
SOYYBJJ12AB017E9FD      48
SOGSPGJ12A8C134FAA      48
Name: song_id, Length: 10000, dtype: int64
```

Song with id SOFRQTD12A81C233C0 is the most interacted song in df. It has been played for 8277 times.

16	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOMGIYR12AB0187973	6	Sea I

Which user listened to the most number of songs?

```
df['user_id'].value_counts()
```

```
6d625c6557df84b60d90426c0116138b617b9449    711
fbee1c8ce1a346fa07d2ef648cec81117438b91f    643
4e11f45d732f4861772b2906f81a7d384552ad12    556
24b98f8ab023f6e7a1c37c7729c623f7b821eb95    540
1aa4fd215aadb160965110ed8a829745cde319eb    533
                                           ...
91fb68459d5f963696d9a8c8bed0556c0a84086a      1
8ede5adc7f577c3c410d66e0b4fe8c230dad807e      1
836071687850be292b31b2af3c4b6b7ca8b52cbd      1
7cc6b08cdc660dc3ffbe063828e2cd1d352f2529      1
9fa8834c229ad20b103783ee0d756420ae0c7c44      1
Name: user_id, Length: 76353, dtype: int64
```

24 b80344d063b5ccb3212f76538f3d9e43d87dca9e SOQLCKR12A81C22440 1 Jewe

- The user with **userId: 6d625c6557df84b60d90426c0116138b617b9449** has listened to the most number of songs. **711** times.

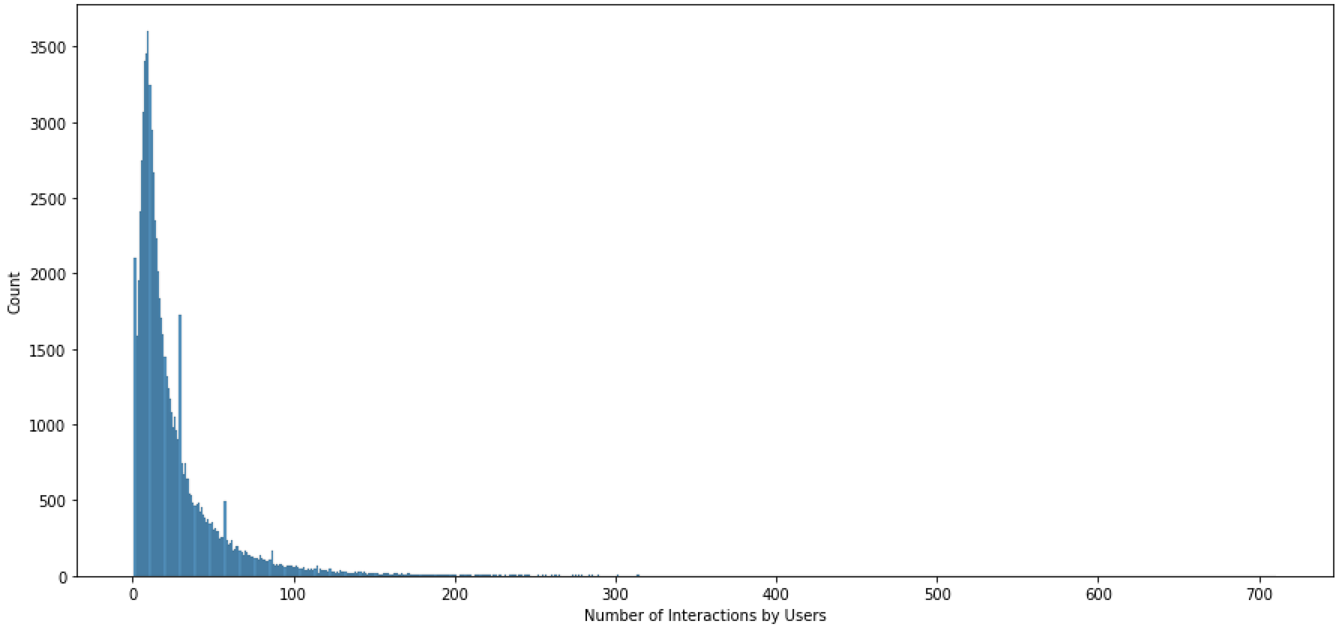
What is the distribution of the user-song interactions in this dataset?

```
#Finding user-song interactions distribution

count_interactions = df.groupby('user_id').count()['song_id']
count_interactions
```

```
user_id
00003a4459f33b92906be11abe0e93efc423c0ff     7
00005c6177188f12fb5e2e82cdbd93e8a3f35e64     5
00030033e3a2f904a48ec1dd53019c9969b6ef1f     9
0007235c769e610e3d339a17818a5708e41008d9    10
0007c0e74728ca9ef0fe4eb7f75732e8026a278b     9
                                            ..
fffce9c1537fbc350ea68823d956eaa8f5236dbe    44
fffd6a2bdef646ce9898b628d5dd56c43df69a9d    11
fffd9635b33f412de8ed02e44e6564e3644cf3c6    17
fffe6d1d8500f1c1f31bd63abce35c0f975a86bf     7
fffea3d509760c984e7d40789804c0e5e289cc86    23
Name: song_id, Length: 76353, dtype: int64
```

```
#Plotting user-song interactions distribution

plt.figure(figsize=(15,7))

sns.histplot(count_interactions)

plt.xlabel('Number of Interactions by Users')
```

```
plt.show()
```



- The user-song distribution has a long tail to the right. Few users played the songs more than one time. As we can not see the detailed numbers from this graph. I am going to look at the statistics of the distribution in details next.

Statistics of the user-song interactions.

```
count_interactions.describe().T
```

```
count    76353.000000
mean        26.194125
std         31.625078
min          1.000000
25%          9.000000
50%         16.000000
75%         31.000000
max        711.000000
Name: song_id, dtype: float64
```

- We see that the mean number of songs a user played is 26 which is higher than the median which is 16. This is expected because the distribution has a long tail to the right to skew the mean to the right. Median is resistant to outliers and therefore is smaller than the mean.

- 25% of users listened to less than 9 songs and 75% users listened to less than 31 songs.

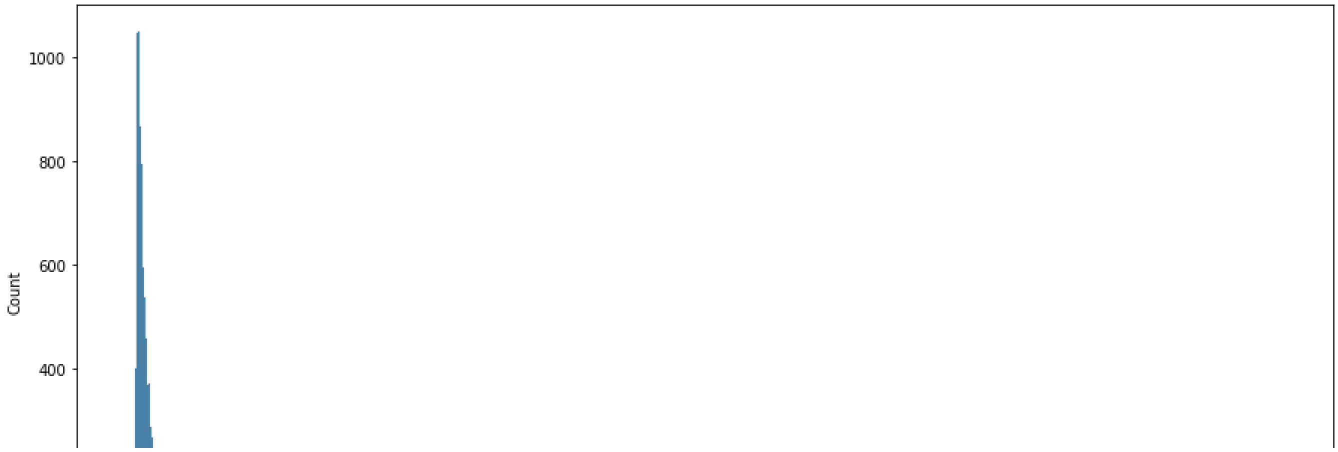
For each song, how many times does it got to be played?

```
#Finding song playcount interactions distribution

play_count_interactions = df.groupby('song_id').count()['play_count']
play_count_interactions.describe().T
```

```
count    10000.000000
mean       200.000000
std        317.715673
min         48.000000
25%         89.000000
50%        124.000000
75%        201.000000
max       8277.000000
Name: play_count, dtype: float64
```

- The mean number of times a song is played is 200 which is bigger than the median which is 124 due to the distribution has a long tail to the right.
- The minimum number of times a song is played is 48. 25% of the songs got played for less than 89 times and 75% of the songs got played for less than 201 times. The maximum number of times a songs is played is 8277 which we have observed ealier.
- The graph below is a visualization of the distribution.

```
#Plotting playcount interactions distribution

plt.figure(figsize=(15,7))

sns.histplot(play_count_interactions)

plt.xlabel('Number of playcount by song_id')

plt.show()
```



For each user, how many times do they listen to songs?

```
#Finding user playcount interactions distribution

user_count_interactions = df.groupby('user_id').count()['play_count']
user_count_interactions.describe().T
```

```
count    76353.000000
mean        26.194125
std         31.625078
min          1.000000
25%          9.000000
50%         16.000000
75%         31.000000
max        711.000000
Name: play_count, dtype: float64
```

- Note this summary statistics is exactly the same as the user-song interactions summary statistics.
- Let's look at the play_count distribution in the dataframe to see why the distribution of the number of songs a user listened to is the same as the total number of times a user listened to the songs.

Look at the summary statistics of play_count from df.

```
df.describe().T
```

	count	mean	std	min	25%	50%	75%	max
play_count	2000000.0	3.045485	6.579720	1.0	1.0	1.0	3.0	2213.0
year	2000000.0	1628.644749	778.728286	0.0	1984.0	2002.0	2007.0	2010.0

- For every song, half of the users only played it one time.

- For the same song, 75% of users played it less than 3 times.
- This explains why the distribution of the number of songs a user listend to is the same as the total number fo times a user listened to the songs. Becauase half of the users only played each song they listened to one time, the number of songs a user listened to is the same as the total number of times a user listened to the songs for these users. Since 75% of users has play count less than 3. So the number of songs a user listened to is very similar to the total number of times a user listened to the songs.
- Therefore whenever it is convenient, I will either user the number of times a song_id appears in the dataset or the total number of play_count for a song_id. They are interchangable for our case.

Method 1: Rank Based Recommendation System

- Rank-based recommendation systems provide recommendations based on the most popular items. This kind of recommendation system is useful when we have **cold start** problems. Cold start refers to the issue when we get a new user into the system and the machine is not able to recommend songs to the new user, as the user did not have any historical interactions in the dataset. In those cases, we can use rank-based recommendation system to recommend songs to the new user.
- Traditional rank-based recommendation will only recommend those popular items based on the assumption that a new user would be the same as the majority of the previous users. This generates a problem: many songs will never be recommended to anyone.
- For this reason, I decided to **combine rank-based recommendation with one recommendation randomly selected from the songs that are not in the popular list**. The rationale behind this algorithm is to give less popular songs a chance to be listened by users. In addition, a new user may very well not be the same as the majority of the existing users. Since we don't have any information about the new user, it is reasonable to assume that the probability of a new user be **different** from the majority of the existing users is the **same** as she/he be the same as the existing users.

To build the rank-based recommendation system, we take **play_count** for each song and then rank them in descending order.

```
#Calculating the play_count of each song
count = df.groupby('song_id').count()['play_count']
#Making a dataframe with the play_count
df_play_count = pd.DataFrame({'play_count':count})
```

```
df_play_count_sort=df_play_count.sort_values(by='play_count', ascending=False)
```

```
df_play_count_sort
```

	play_count
song_id	
SOFRQTD12A81C233C0	8277
SOAUWYT12A81C206F1	7032
SOAXGDH12A8C13F8A1	6949
SOBONKR12A58A7A7E0	6412
SOSXLTC12AF72A7F54	6145
...	...
SOWNLZF12A58A79811	51
SOLIGVL12AB017DBAE	51
SOBPGWB12A6D4F7EF3	50
SOYYBJJ12AB017E9FD	48
SOGSPGJ12A8C134FAA	48

10000 rows × 1 columns

Add index to df_play_count_sort_reset.

```
df_play_count_sort_reset=df_play_count_sort.reset_index()
df_play_count_sort_reset
```

	song_id	play_count
3	SOBONKR12A58A7A7E0	6412

```
#If we want the song_id for the third most played songs, we can do the following.
df_play_count_sort_reset.iat[2,0]
```

    'SOAXGDH12A8C13F8A1'

Now create a function that gives the recommendation of n songs in which n-1 are the most popular songs in terms of play count and one is randomly selected from the rest of the 10000-n songs.

```
def top_n_songs(n):
    idx=[i for i in range(n,len(df_play_count_sort_reset))] # Create a list with numbers star
    rand_idx = random.randrange(0,len(df_play_count_sort_reset)-n)#Randomly select a number f
    random_num = idx[rand_idx]
    popular=[i for i in range(0,n-1)]
    popular.append(random_num)


    return df_play_count_sort_reset.iloc[popular,0]
```

We can **use this function with different n's** to get songs to recommend.

Recommending top 10 songs based on popularity and promotion of unpopular songs.

```
top_n_songs(10)
```

```
0       SOFRQTD12A81C233C0
1       SOAUWYT12A81C206F1
2       SOAXGDH12A8C13F8A1
3       SOBONKR12A58A7A7E0
4       SOSXLTC12AF72A7F54
5       SONYKOW12AB01849C9
6       SOEGIYH12A6D4FC0E3
7       SOLFXKT12AB017E3E0
8       SODJWHY12A8C142CCE
4980    SOYQEXD12AB0186267
Name: song_id, dtype: object
```

Summary:

- For this recommendation system, the first 9 songs are based on popularity and the last one is based on a random selection.

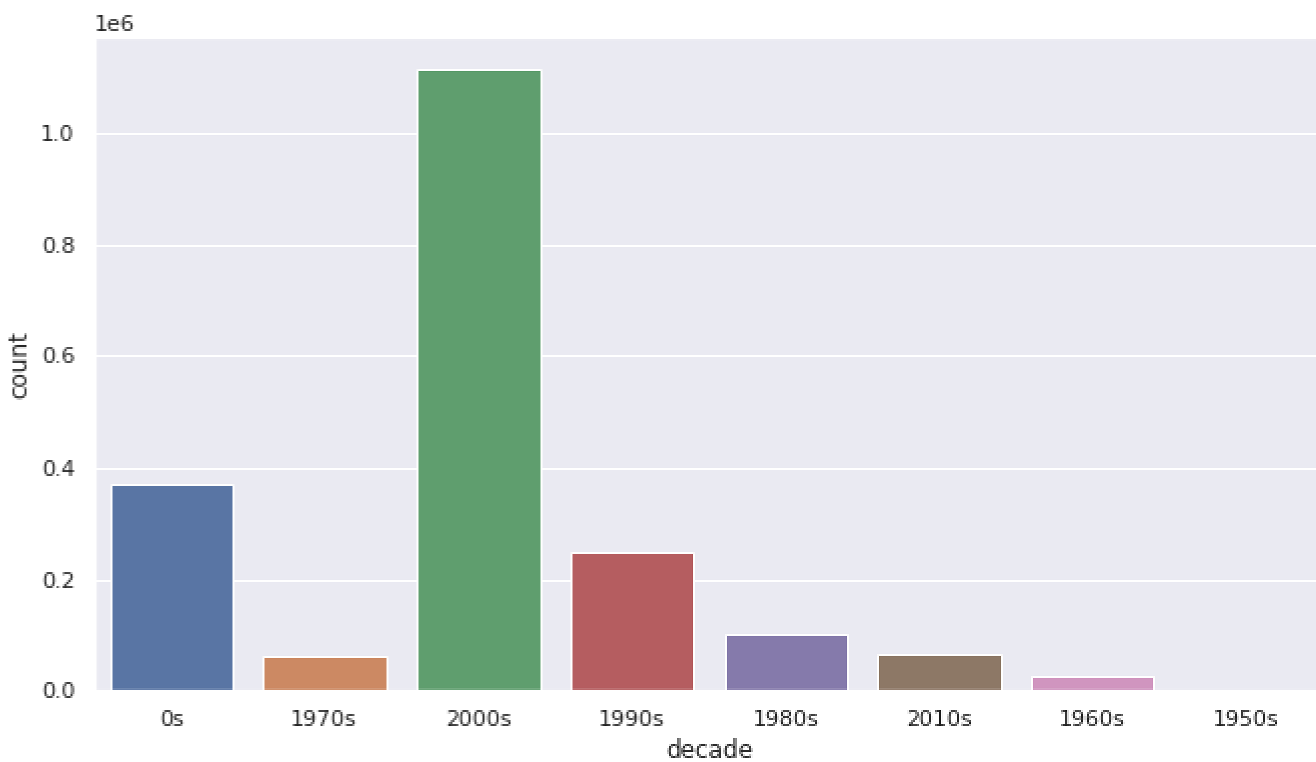
- This type of recommendation works for new users of whom we have no information about. My method helps to give songs that are less played a chance to be recommended. The rationale behind it is that a new user from whom we have no information about has a good probability of not being the same as the majority of the original users

Now build a **collaborative filtering based recommendation system**.

- In this type of recommendation system, `we do not need any information` about the users or items. We only need user item interaction data to build a collaborative recommendation system.
- There are mainly two types of collaborative filtering based recommendation systems: **similarity/Neighborhood based** and **model based**.

```
def get_decade(year):
    period_start = int(year/10) * 10
    decade = '{}s'.format(period_start)
    return decade

df['decade'] = df['year'].apply(get_decade)

sns.set(rc={'figure.figsize':(11 ,6)})
sns.countplot(df['decade'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f04e3dc1d10>
```



Method 2: Similarity/Neighborhood Based Collaborative Filtering

First, we compute the **user-item interactions matrix**

```
interactions_matrix = df.pivot(index='user_id', columns='song_id', values='play_count')
interactions_matrix
```

song_id	SOAAAGQ12A8C1420C8	SOACPJ12A81C21360	SOAA
user_id			
00003a4459f33b92906be11abe0e93efc423c0ff	NaN	NaN	
00005c6177188f12fb5e2e82cdbd93e8a3f35e64	NaN	NaN	
00030033e3a2f904a48ec1dd53019c9969b6ef1f	NaN	NaN	
0007235c769e610e3d339a17818a5708e41008d9	NaN	NaN	
0007c0e74728ca9ef0fe4eb7f75732e8026a278b	NaN	NaN	
...	...	...	
fffce9c1537fbc350ea68823d956eaa8f5236dbe	NaN	NaN	
fffd6a2bdef646ce9898b628d5dd56c43df69a9d	NaN	NaN	
fffd9635b33f412de8ed02e44e6564e3644cf3c6	NaN	NaN	
fffe6d1d8500f1c1f31bd63abce35c0f975a86bf	NaN	NaN	
fffea3d509760c984e7d40789804c0e5e289cc86	NaN	NaN	

76353 rows × 10000 columns

- This is a very **sparse matrix** with 76353 rows and 10000 columns.
- Since the matrix is extremely sparse, there is not enough ram to fill all the NaN by 0s using Google colab. Therefore, I picked 10000 most played songs to analyze.
- I can extend it to more number of songs later when there is better computational power.

Now we want to figure out what items have been played per customer, using groupby:

```
DataGrouped = df.groupby(['user_id','song_id']).sum().reset_index()
DataGrouped.head()
```

	user_id	song_id	play_count	year
0	00003a4459f33b92906be11abe0e93efc423c0ff	SOJJRVI12A6D4FBE49	1	2003
1	00003a4459f33b92906be11abe0e93efc423c0ff	SOKJWZB12A6D4F9487	4	0

```
DataGrouped.nunique()
```

```
user_id       76353
song_id       10000
play_count      295
year             58
dtype: int64
```

**Recall we have 76353 distinct number of user_id and 10000 distinct number of song_id. The interesting question is how many different songs our customers actually played? I will use Numpy's .list function to get the play count per distinct number of Customers and Songs:

```
customers = list(np.sort(DataGrouped.user_id.unique()))
# a list of values, sotre 76353 unique customers
products = list(DataGrouped.song_id.unique())
quantity = list(DataGrouped.play_count)
```

```
from pandas import DataFrame
DfCustomerUnique = DataFrame(customers,columns=['user_id'])
DfCustomerUnique.head()
DfCustomerUnique.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 76353 entries, 0 to 76352
Data columns (total 1 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   user_id  76353 non-null  object
dtypes: object(1)
memory usage: 596.6+ KB
```

Now we can build our sparse matrix:

```
from scipy import sparse
from pandas.api.types import CategoricalDtype
rows = DataGrouped.user_id.astype(CategoricalDtype(categories=customers)).cat.codes
# We have got 76353 unique customers, which make up 2000000 data rows (index)
# Get the associated row indices
cols = DataGrouped.song_id.astype(CategoricalDtype(categories= products)).cat.codes
# We have got unique 10000 song_id, making up 2000000 data rows (index)
# Get the associated column indices
#Compressed Sparse Row matrix
```

```
SongSparse = sparse.csr_matrix((quantity, (rows, cols)), shape=(len(customers), len(products)
#len of customers=76353, len of products=10000
#csr_matrix((df, (row_ind, col_ind)), [shape=(M, N)])
#where data, row_ind and col_ind satisfy the relationship a[row_ind[k], col_ind[k]] = data[k]
SongSparse
```

```
<76353x10000 sparse matrix of type '<class 'numpy.longlong'>'
        with 2000000 stored elements in Compressed Sparse Row format>
```

```
#We have 76353 customers with 10000 songs.
#In terms of sparsity of the matrix, that makes:
MatrixSize = SongSparse.shape[0]*SongSparse.shape[1] # 763530000 possible interactions in the
SongAmount = len(SongSparse.nonzero()[0]) # 13837 SalesItems interacted with;
sparsity = 100*(1 - (SongAmount/MatrixSize))
sparsity
```

```
99.73805875342161
```

This means that our data is extremely sparse. 99.7% of the entries are not filled. Only 0.3% of the interaction matrix is filled.

Our Collaborative Filtering will be based on binary data (a set of just two values), which is an important special case of categorical data. For every dataset I will add a 1 as played. That means, that this customer has played this song, no matter how many times the customer actually has played in the past. I decided to use this binary data approach for our recommending because the question asks for the probability of a customer playing a song. Another approach would be to use the play_count and normalize it if we want to treat the number of play_count as a kind of taste factor, meaning that someone who listened to song_id 100 times-loves the song; while another Customer listened to the same song_id only 5 times- does not like it as much.

```
def create_DataBinary(DataGrouped):
    DataBinary = DataGrouped.copy()
    DataBinary['playedYes'] = 1
    return DataBinary
DataBinary = create_DataBinary(DataGrouped)
DataBinary.head()
```

	user_id	song_id	play_count	year	pla
1	00003a4459f33b92906be11abe0e93efc423c0ff	SOKJWZB12A6D4F9487	4	0	

Finally, let’s get rid of the column play_count:

```
play_data=DataBinary.drop(['play_count'], axis=1)
play_data.head()
```

	user_id	song_id	year	playedYes
0	00003a4459f33b92906be11abe0e93efc423c0ff	SOJJRVI12A6D4FBE49	2003	1
1	00003a4459f33b92906be11abe0e93efc423c0ff	SOKJWZB12A6D4F9487	0	1
2	00003a4459f33b92906be11abe0e93efc423c0ff	SOMZHIH12A8AE45D00	2007	1
3	00003a4459f33b92906be11abe0e93efc423c0ff	SONFEUF12AAF3B47E3	0	1
4	00003a4459f33b92906be11abe0e93efc423c0ff	SOVMGXI12AF72A80B0	2003	1

Our recommendation is based on the assumption that customers who played songs in a similar quantity share one or more hidden preferences. Due to this shared latent or hidden features customers will likely played similar songs.

```
# Top 10 users based on number of songs they played.
most_rated = df.groupby('user_id').size().sort_values(ascending=False)[:5000].to_frame()

most_rated=most_rated.reset_index()
most_rated.shape
```

```
(5000, 2)
```

```
most_rated.head()
```

	user_id	0
0	6d625c6557df84b60d90426c0116138b617b9449	711
1	fbee1c8ce1a346fa07d2ef648cec81117438b91f	643
2	4e11f45d732f4861772b2906f81a7d384552ad12	556
3	24b98f8ab023f6e7a1c37c7729c623f7b821eb95	540
4	1aa4fd215aadb160965110ed8a829745cde319eb	533

```
# Rename the column of most_rated
most_rated.rename(columns = {0: 'songs_play'}, inplace = True)
most_rated
```

	user_id	songs_play
0	6d625c6557df84b60d90426c0116138b617b9449	711
1	fbee1c8ce1a346fa07d2ef648cec81117438b91f	643
2	4e11f45d732f4861772b2906f81a7d384552ad12	556
3	24b98f8ab023f6e7a1c37c7729c623f7b821eb95	540
4	1aa4fd215aadb160965110ed8a829745cde319eb	533
...	...	...
4995	5a9eccc83bcfd207e101c018e73bc9869dfdfadb	72
4996	3c51f21932017a30fe9d7441a6ed6dbc6e9768b1	72
4997	a906b0e81b1ceb8e15fdaab487073c5109da3a80	72
4998	90d2951cd918fa4630a4121ebe64ae069060983d	72
4999	5f89ba1ce3ad55768b715ec1741d7606aa48c91f	72

5000 rows × 2 columns

```
most_rated.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   user_id     5000 non-null   object
 1   songs_play  5000 non-null   int64
dtypes: int64(1), object(1)
memory usage: 78.2+ KB
```

```
user_names=most_rated.user_id.values.tolist()
print(user_names[0:1])
```

```
['6d625c6557df84b60d90426c0116138b617b9449']
```

```
df_new=df[df['user_id'].isin(user_names)]
```

```
df_new.nunique()
```

```
user_id         5000
song_id        10000
play_count       175
title           9567
release         5388
artist_name     3375
year              58
```

```
decade             8
dtype: int64
```

```
%%time
interactions_matrix_new = df_new.pivot(index='user_id', columns='song_id', values='play_count
print(interactions_matrix_new)
#print(df_new.pivot(index='index', columns='song_id', values='play_count')['song_id'])
interactions_matrix_new.info()
```

```
song_id                                   SOAAAGQ12A8C1420C8  ...  SOZZZPV12A8C1444B5
user_id                                                       ...
000ebc858861aca26bac9b49f650ed424cf882fc                 NaN  ...                 NaN
00185e316f07f0f00c325ca034be59c15b362401                 NaN  ...                 NaN
0028292aa536122c1f86fd48a39bd83fe582d27f                 NaN  ...                 NaN
0031572620fa7f18487d3ea22935eb28410ecc4c                 NaN  ...                 NaN
003d0f3aac94fd261bb74c0124a90750579972d4                 NaN  ...                 NaN
...                                                      ...  ...                 ...
ffdaab327f2fc6b9fa01a4e3e7f41fdd0e468046                 NaN  ...                 NaN
fff03efd1550136063389fa71125194614e1c68f                 NaN  ...                 NaN
fff0b1ab076f0b71cbde9c7dcbcfca400708d845                 NaN  ...                 NaN
fff6c30c773e6ffafcac213c9afd9666afaf6d63                 NaN  ...                 NaN
fffc0df75a48d823ad5abfaf2a1ee61eb1e3302c                 NaN  ...                 NaN

[5000 rows x 10000 columns]
<class 'pandas.core.frame.DataFrame'>
Index: 5000 entries, 000ebc858861aca26bac9b49f650ed424cf882fc to fffc0df75a48d823ad5abfa
Columns: 10000 entries, SOAAAGQ12A8C1420C8 to SOZZZPV12A8C1444B5
dtypes: float64(10000)
memory usage: 381.5+ MB
CPU times: user 959 ms, sys: 17.7 ms, total: 977 ms
Wall time: 968 ms
```

Below is one of the techniques to find out similar users. Here each user is denoted by a **vector of 10000 dimensions**. Then we will find out pairwise **cosine similarities** for all the users. If two vectors i.e. users are exactly same or lie on top each other, then they are most similar and cosine similarity will be 1

Compute interaction matrix for this subset df_new.

Get the column names.

```
#Get the column names and save it in col_names.
col_names=interactions_matrix_new.columns.values.tolist()
```

Get the row names.

```
print(col_names)
```

```
['SOAAAGQ12A8C1420C8', 'SOAACPJ12A81C21360', 'SOAACSG12AB018DC80', 'SOAAEJI12AB0188AB5',
```

```
len(col_names)
```

```
10000
```

10000 columns (songs).

```
#Get the column names and save it in row_names.
row_names=interactions_matrix_new.index.values.tolist()
```

```
print(row_names)
len(row_names)
```

```
['000ebc858861aca26bac9b49f650ed424cf882fc', '00185e316f07f0f00c325ca034be59c15b362401',
5000
```

5000 rows (users)

Change the column names and row names to their corresponding index and rebuild the interactions_matrix_new. Now the rows and columns are all indexed by numbers.

```
for i in range(0,10000):
  col_names[i]=i
print(col_names)
interactions_matrix_new.columns=col_names
```

```
for i in range(0,5000):
  row_names[i]=i
interactions_matrix_new.index=row_names

print(interactions_matrix_new)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 
      0     1     2     3     4     5     ...  9994  9995  9996  9997  9998  9999
0      NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN   NaN   NaN   NaN   NaN   NaN
1      NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN   NaN   NaN   NaN   NaN   NaN
2      NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN   NaN   NaN   NaN   NaN   NaN
3      NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN   NaN   NaN   NaN   NaN   NaN
4      NaN   NaN   NaN   NaN   NaN   5.0  ...   NaN   NaN   NaN   NaN   NaN   NaN
...    ...   ...   ...   ...   ...   ...  ...   ...   ...   ...   ...   ...   ...
4995   NaN   NaN   NaN   1.0   NaN   NaN  ...   NaN   NaN   NaN   NaN   NaN   NaN
4996   NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN   NaN   NaN   NaN   NaN   NaN
4997   NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN   NaN   NaN   NaN   NaN   NaN
```

```
4998   NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN   NaN   NaN   NaN   NaN   NaN
4999   NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN   NaN   NaN   NaN   NaN   NaN

[5000 rows x 10000 columns]
```

Cosine similarity can't take missing values in its vectors while computing, hence we need to fill those NaN values with zeros

```
%%time
interactions_matrix_new.fillna(0, inplace=True)
print(interactions_matrix_new.head())
print(interactions_matrix_new.info())
#print('rows is', interactions_matrix_new_reset.shape[0])
#print('column is', interactions_matrix_new_reset.shape[1])
#vector=interactions_matrix_new_reset.iloc[0,1:5176]

#print('length of vector is', len(vector))

#print(vector)
# reconstruct dense matrix
#interactions_matrix_new = S.todense()
#interactions_matrix_new
```

```
     0     1     2     3     4     5     ...  9994  9995  9996  9997  9998  9999
0   0.0   0.0   0.0   0.0   0.0   0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0
1   0.0   0.0   0.0   0.0   0.0   0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0
2   0.0   0.0   0.0   0.0   0.0   0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0
3   0.0   0.0   0.0   0.0   0.0   0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0
4   0.0   0.0   0.0   0.0   0.0   5.0  ...   0.0   0.0   0.0   0.0   0.0   0.0

[5 rows x 10000 columns]
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5000 entries, 0 to 4999
Columns: 10000 entries, 0 to 9999
dtypes: float64(10000)
memory usage: 381.5 MB
None
CPU times: user 2.47 s, sys: 16.9 ms, total: 2.49 s
Wall time: 2.49 s
```

```
interactions_matrix_new.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5000 entries, 0 to 4999
Columns: 10000 entries, 0 to 9999
dtypes: float64(10000)
memory usage: 381.5 MB
```

Now, create a **function to find similar users and similarity scores for given user_id and interaction matrix**

```
def similar_users(j, interactions_matrix_new):
    similarity = []
    for user in range(0, interactions_matrix_new.shape[0]):

        #finding cosine similarity between the user_id and each user

        sim = cosine_similarity([interactions_matrix_new.loc[j]], [interactions_matrix_new.lo

            #Appending the user and the corresponding similarity score with user_id as a tupl
        similarity.append((user, sim))

    similarity.sort(key=lambda x: x[1], reverse=True)   #Sorting the list on the basis of sim

    most_similar_users = [tup[0] for tup in similarity] #Extracting the user from each tuple

    similarity_score = [tup[1] for tup in similarity]   #Extracting the similarity score from

    most_similar_users.remove(j)                  #Removing the original user and keeping onl
    similarity_score.remove(similarity_score[0])

    return most_similar_users, similarity_score
```

We can use this function to find similar users for different user_id.

Find out top 10 similar users to the user_id corresponding to index=3 and their similarity score.

```
similar_users(3, interactions_matrix_new)[0][:10]
```

```
[786, 3699, 4365, 2735, 1611, 3552, 3250, 4340, 280, 2763]
```

Find out top 10 similar users to the user_id corresponding to index=200 and their similarity score.

```
similar_users(200, interactions_matrix_new)[0][:10]
```

```
[410, 4660, 1429, 156, 3612, 3697, 517, 3839, 1374, 3283]
```

- These are the top 10 users that are most similar to the user with index 200.

We can also find the level of similarity between similar users and the user with user with index n.

```
similar_users(200, interactions_matrix_new)[1][:10]
```

```
[array([[0.32822587]]),
 array([[0.32395614]]),
 array([[0.31434613]]),
 array([[0.3069348]]),
 array([[0.30335633]]),
 array([[0.29964655]]),
 array([[0.29824361]]),
 array([[0.29425153]]),
 array([[0.29227382]]),
 array([[0.27860285]])]
```

We have learned how to find similar users for a given user but how do we find which **songs to recommend to a particular user?** This is done by finding the **songs which have been played the most by similar users** but not by the user of interest.

Let's create a function to do it.

```
def recommendations(j, num_of_songs, interactions_matrix_new):

    #Saving similar users using the function similar_users defined above
    most_similar_users = similar_users(j, interactions_matrix_new)[0]

    #Finding song_id s with which the user j has interacted
    song_ids = set(list(interactions_matrix_new.columns[np.where(interactions_matrix_new.loc[
    recommendations = []

    observed_interactions = song_ids.copy()
    for similar_user in most_similar_users:
        if len(recommendations) < num_of_songs:
            #Finding 'n' songs which have been played by similar users but not by the user_id
            similar_user_song_ids = set(list(interactions_matrix_new.columns[np.where(interac
            recommendations.extend(list(similar_user_song_ids.difference(observed_interaction
            observed_interactions = observed_interactions.union(similar_user_song_ids)
        else:
            break

    return recommendations[:num_of_songs]
```

Finally, we can recommend `n` number of songs to any user using the function defined above

Recommend 10 songs to user with index 200 based on similarity based collaborative filtering

```
recommendations(200, 10, interactions_matrix_new)
```

```
[3074, 7173, 6670, 4631, 7709, 3614, 6175, 2080, 31, 1058]
```

3. Model Based Collaborative Filtering - Matrix Factorization.

- Model-based Collaborative Filtering is a **personalized recommendation system**, the recommendations are based on the past behavior of the user and it is not dependent on any additional information. We use **latent features** to find recommendations for each user.
- I will use **SVD** to compute the latent features from the user-item matrix that we already learned earlier. But SVD does not work when we missing values in the user-item matrix.
- I already found the user-item matrix when computing cosine similarity. The matrix is given by **interactions_matrix_new_reset**.

Perform the decomposition using the `svd() function` from the linalg module of the NumPy library.

```
u, s, vt = np.linalg.svd(interactions_matrix_new, full_matrices=False)
```

```
u.shape
```

```
(5000, 5000)
```

```
s.shape
```

```
(5000,)
```

```
vt.shape
```

```
(5000, 10000)
```

Splitting the dataset and selecting optimal latent variables Now, we need to find the appropriate K to use in order to re-generate the interaction matrix and make predictions. We will split the data into **train and test data** and make predictions for different value of K. We will choose the K which gives good performance on the train and test data.

Split the data into train and test data.

```
X_train, X_test = train_test_split(df_new, test_size=0.2, random_state=42)
```

```
X_train.shape # want to do this to get the best version of number of ...
```

```
(468877, 8)
```

```
X_test.shape
```

```
(117220, 8)
```

Create the **train and test interactions matrices**.

```
#Train interaction matrix
interactions_matrix_train = X_train.pivot(index='user_id', columns='song_id', values='play_co
interactions_matrix_train.fillna(0, inplace=True)
```

```
interactions_matrix_train.shape
```

```
(5000, 10000)
```

```
#Test interaction matrix
interactions_matrix_test = X_test.pivot(index='user_id', columns='song_id', values='play_coun
interactions_matrix_test.fillna(0, inplace=True)
```

```
interactions_matrix_test.shape
```

```
(5000, 9876)
```

I will use the **interactions_matrix_train** to find U, S, and V transpose using SVD. Then find the subset of rows in the **interactions_matrix_test** dataset which I can predict using this matrix decomposition with different numbers of latent features.

```
#Finding unique users in train and test data and then taking their intersection i.e. common u
train_idx = set(interactions_matrix_train.index)
test_idx = set(interactions_matrix_test.index)
match_idx = train_idx.intersection(test_idx)

#Finding unique songs in train and test data and then taking their intersection
 i.e. common songs in train and test data
train_songs = set(interactions_matrix_train.columns)
test_songs = set(interactions_matrix_test.columns)
match_cols = train_songs.intersection(test_songs)

#Selecting only common users and movies from the test interaction matrix
interactions_matrix_test = interactions_matrix_test.loc[match_idx, match_cols]
```

```
interactions_matrix_test.shape
```

```
(5000, 9876)
```

Now, let's decompose the **interactions_matrix_train** and **find the U and Vt for the test data** using the common users and movies in the train and test data.

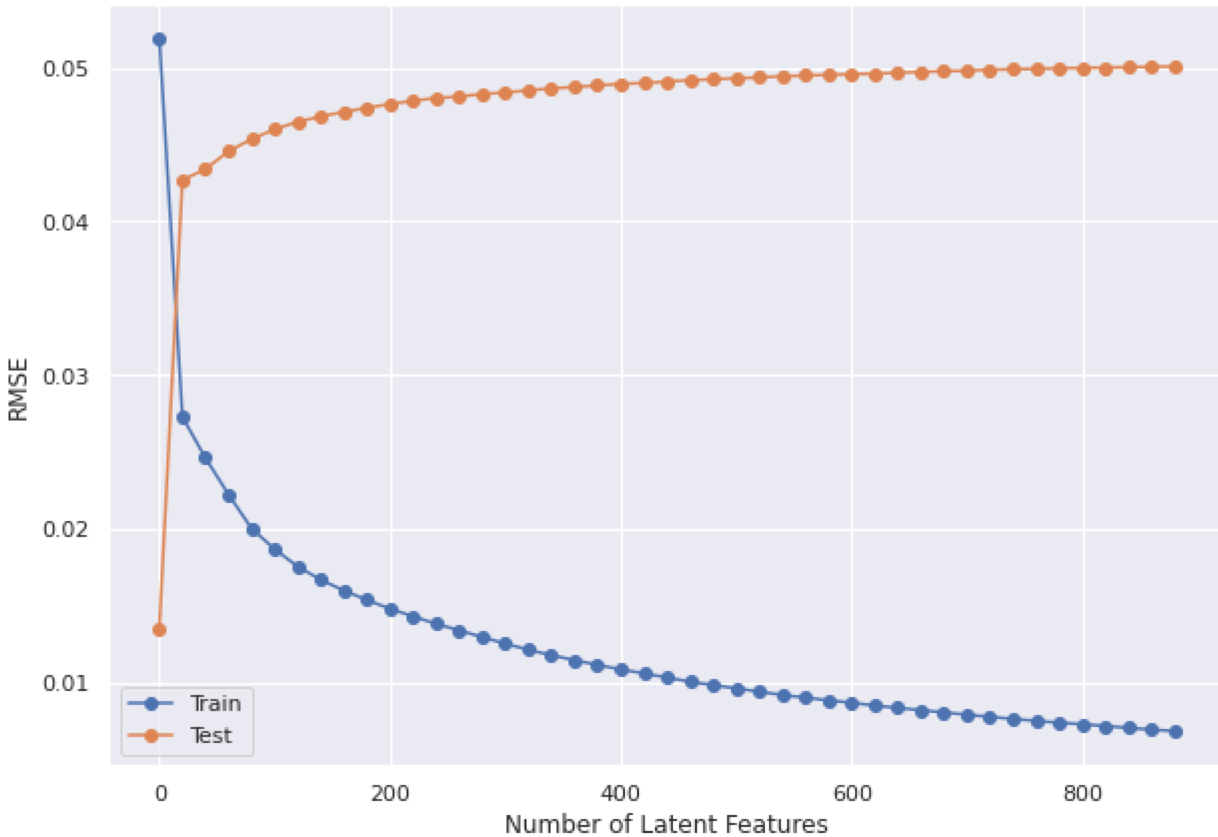
```
u_train, s_train, vt_train = np.linalg.svd(interactions_matrix_train, full_matrices=False)
```

```
#Finding u_test and vt_test matrices using u_train, vt_train and common user/movies in train
row_idxs = interactions_matrix_train.index.isin(test_idx)
col_idxs = interactions_matrix_train.columns.isin(test_songs)
u_test = u_train[row_idxs, :]
vt_test = vt_train[:, col_idxs]
```

We have calculated U and Vt matrices for the train as well as test data. Now, we need to find the number of latent features that give us the **lowest RMSE on the train and the test data**.

```
#Creating array for number of latent features and empty lists to store train and test errors
latent_features = np.arange(0, 900, 20)
train_error = []
test_error = []

for k in latent_features:
    #Slicing the U, S, and Vt matrices to get k latent features from train and test data
    s_train_lat, u_train_lat, vt_train_lat = np.diag(s_train[:k]), u_train[:, :k], vt_train[:
    u_test_lat, vt_test_lat = u_test[:, :k], vt_test[:k, :]

    #Regenerating train and test interaction matrices using k latent features
    interactions_matrix_train_preds = np.around(np.matmul(np.matmul(u_train_lat, s_train_lat)
    interactions_matrix_test_preds = np.around(np.matmul(np.matmul(u_test_lat, s_train_lat),

    #Calculating the actual and predicted average play_count for each song in the training da
    avg_play_count_train = interactions_matrix_train.mean(axis=0)
    avg_play_count_train_pred = interactions_matrix_train_preds.mean(axis=0)

    #Calculating the actual and predicted average play_count for each movie in the test data
    avg_play_count_test = interactions_matrix_test.mean(axis=0)
    avg_play_count_test_pred = interactions_matrix_test_preds.mean(axis=0)

    #Calculating train and test RMSE
    train_rmse = mean_squared_error(avg_play_count_train, avg_play_count_train_pred, squared=
    test_rmse = mean_squared_error(avg_play_count_test, avg_play_count_test_pred, squared=Fal

    train_error.append(train_rmse)
    test_error.append(test_rmse)

#Plotting train and test RMSE
plt.figure(figsize=(10,7))
plt.plot(latent_features, train_error, label='Train', marker='o');
plt.plot(latent_features, test_error, label='Test', marker='o');
plt.xlabel('Number of Latent Features');
```

```
plt.xlabel('Number of Latent Features');
plt.ylabel('RMSE');
plt.legend();
```



From the above plot we can see that we got a **reasonable RMSE in both train and test dataset has latent features be less than 100.** If we increase the latent features it will be overfitted and decreasing the latent features will underfit the model.

Reconstruct the original interaction matrix using latent features = 10 in the same way as above

```
s_final, u_final, vt_final = np.diag(s[:10]), u[:, :10], vt[:10, :]
songs_predicted_play_count = np.around(np.matmul(np.matmul(u_final, s_final), vt_final))
songs_predicted_play_count = pd.DataFrame(abs(songs_predicted_play_count), columns = interact
songs_predicted_play_count.head()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 10000 columns

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

We have the predictions of play counts but we need to create a **function to recommend songs** to the users on the basis of predicted play counts for each song.

```
# Recommend the songs with the highest predicted play counts

def recommend_songs(user_idx, interactions_matrix_new, preds_df, num_recommendations):

    # Get and sort the user's ratings from the actual and predicted interaction matrix
    sorted_user_play_count = interactions_matrix_new.loc[user_idx].sort_values(ascending=Fals
    sorted_user_predictions = preds_df.loc[user_idx].sort_values(ascending=False)

    #Creating a dataframe with actual and predicted ratings columns
    temp = pd.concat([sorted_user_play_count, sorted_user_predictions], axis=1)
    temp.index.name = 'Recommended Songs'
    temp.columns = ['user_play_count', 'user_predictions']

    #Filtering the dataframe where actual play count are 0 which implies that the user has no
    temp = temp.loc[temp.user_play_count == 0]

    #Recommending movies with top predicted play counts.
    temp = temp.sort_values('user_predictions', ascending=False)
    print('\nBelow are the recommended songs for user(user_id = {}):\n'.format(user_index))
    print(temp['user_predictions'].head(num_recommendations))
```

Now, let's use the function defined above to **recommend songs to a user**

```
user_index = 121
num_recommendations = 10
recommend_songs(user_index, interactions_matrix_new, songs_predicted_play_count, num_recommen
```

Finally, we can **calculate the RMSE** for the final play counts predicted using the model-based recommendation system.

```
#Create a dataframe containing average actual play counts and avearge predicted play counts o
rmse_df = pd.concat([interactions_matrix_new.mean(), songs_predicted_play_count.mean()], axis
rmse_df.columns = ['Avg_actual_play_count', 'Avg_predicted_play_count']
rmse_df
```

	Avg_actual_play_count	Avg_predicted_play_count
0	0.0146	0.0000
1	0.0378	0.0002
2	0.0082	0.0002
3	0.0086	0.0000
4	0.0242	0.0000
...	...	...
9995	0.0204	0.0000
9996	0.0164	0.0000
9997	0.0204	0.0000
9998	0.0098	0.0000
9999	0.0030	0.0000

10000 rows × 2 columns

	Avg_actual_play_count	Avg_predicted_play_count
0	0.0146	0.0000
1	0.0378	0.0002
2	0.0082	0.0002
3	0.0086	0.0000
4	0.0242	0.0000
...	...	...
9995	0.0204	0.0000
9996	0.0164	0.0000
9997	0.0204	0.0000
9998	0.0098	0.0000
9999	0.0030	0.0000

10000 rows × 2 columns


```
RMSE = mean_squared_error(rmse_df['Avg_actual_play_count'], rmse_df['Avg_predicted_play_count
print('\nRMSE SVD Model = {} \n'.format(RMSE))
```

RMSE for the SVD model with **10 latent features** is **0.050100743137434615**.

Use latent feature 5.

```
s_final, u_final, vt_final = np.diag(s[:5]), u[:, :5], vt[:5, :]
songs_predicted_play_count = np.around(np.matmul(np.matmul(u_final, s_final), vt_final))
songs_predicted_play_count = pd.DataFrame(abs(songs_predicted_play_count), columns = interact
songs_predicted_play_count.head()
```

```
# Recommend the songs with the highest predicted play counts

def recommend_songs(user_idx, interactions_matrix_new, preds_df, num_recommendations):

    # Get and sort the user's ratings from the actual and predicted interaction matrix
    sorted_user_play_count = interactions_matrix_new.loc[user_idx].sort_values(ascending=Fals
    sorted_user_predictions = preds_df.loc[user_idx].sort_values(ascending=False)

    #Creating a dataframe with actual and predicted ratings columns
    temp = pd.concat([sorted_user_play_count, sorted_user_predictions], axis=1)
    temp.index.name = 'Recommended Songs'
    temp.columns = ['user_play_count', 'user_predictions']

    #Filtering the dataframe where actual play count are 0 which implies that the user has no
    temp = temp.loc[temp.user_play_count == 0]

    #Recommending movies with top predicted play counts.
    temp = temp.sort_values('user_predictions', ascending=False)
    print('\nBelow are the recommended songs for user(user_id = {}):\n'.format(user_index))
    print(temp['user_predictions'].head(num_recommendations))
```

```
user_index = 121
num_recommendations = 10
recommend_songs(user_index, interactions_matrix_new, songs_predicted_play_count, num_recommen
```

```
#Create a dataframe containing average actual play counts and avearge predicted play counts o
rmse_df = pd.concat([interactions_matrix_new.mean(), songs_predicted_play_count.mean()], axis
rmse_df.columns = ['Avg_actual_play_count', 'Avg_predicted_play_count']
rmse_df
```

```
RMSE = mean_squared_error(rmse_df['Avg_actual_play_count'], rmse_df['Avg_predicted_play_count
print('\nRMSE SVD Model = {} \n'.format(RMSE))
```

Double-click (or enter) to edit

RMSE for the SVD model with **5 latent features** is **0.05335645683874617**.

Try latent feature 7.

```
s_final, u_final, vt_final = np.diag(s[:7]), u[:, :7], vt[:7, :]
songs_predicted_play_count = np.around(np.matmul(np.matmul(u_final, s_final), vt_final))
songs_predicted_play_count = pd.DataFrame(abs(songs_predicted_play_count), columns = interact
songs_predicted_play_count.head()
```

```
# Recommend the songs with the highest predicted play counts

def recommend_songs(user_idx, interactions_matrix_new, preds_df, num_recommendations):

    # Get and sort the user's ratings from the actual and predicted interaction matrix
    sorted_user_play_count = interactions_matrix_new.loc[user_idx].sort_values(ascending=Fals
    sorted_user_predictions = preds_df.loc[user_idx].sort_values(ascending=False)

    #Creating a dataframe with actual and predicted ratings columns
    temp = pd.concat([sorted_user_play_count, sorted_user_predictions], axis=1)
    temp.index.name = 'Recommended Songs'
    temp.columns = ['user_play_count', 'user_predictions']

    #Filtering the dataframe where actual play count are 0 which implies that the user has no
    temp = temp.loc[temp.user_play_count == 0]

    #Recommending movies with top predicted play counts.
    temp = temp.sort_values('user_predictions', ascending=False)
    print('\nBelow are the recommended songs for user(user_id = {}):\n'.format(user_index))
    print(temp['user_predictions'].head(num_recommendations))
```

```
user_index = 121
num_recommendations = 10
recommend_songs(user_index, interactions_matrix_new, songs_predicted_play_count, num_recommen
```

```
#Create a dataframe containing average actual play counts and avearge predicted play counts o
rmse_df = pd.concat([interactions_matrix_new.mean(), songs_predicted_play_count.mean()], axis
rmse_df.columns = ['Avg_actual_play_count', 'Avg_predicted_play_count']
rmse_df
```

```
RMSE = mean_squared_error(rmse_df['Avg_actual_play_count'], rmse_df['Avg_predicted_play_count
print('\nRMSE SVD Model = {} \n'.format(RMSE))
```

RMSE for the SVD model with **7 latent features** is **0.05316204062167089**.

Try 16 latent features.

```
s_final, u_final, vt_final = np.diag(s[:16]), u[:, :16], vt[:16, :]
songs_predicted_play_count = np.around(np.matmul(np.matmul(u_final, s_final), vt_final))
songs_predicted_play_count = pd.DataFrame(abs(songs_predicted_play_count), columns = interact
songs_predicted_play_count.head()
```

```
# Recommend the songs with the highest predicted play counts

def recommend_songs(user_idx, interactions_matrix_new, preds_df, num_recommendations):

    # Get and sort the user's ratings from the actual and predicted interaction matrix
    sorted_user_play_count = interactions_matrix_new.loc[user_idx].sort_values(ascending=Fals
    sorted_user_predictions = preds_df.loc[user_idx].sort_values(ascending=False)

    #Creating a dataframe with actual and predicted ratings columns
    temp = pd.concat([sorted_user_play_count, sorted_user_predictions], axis=1)
    temp.index.name = 'Recommended Songs'
    temp.columns = ['user_play_count', 'user_predictions']

    #Filtering the dataframe where actual play count are 0 which implies that the user has no
    temp = temp.loc[temp.user_play_count == 0]

    #Recommending movies with top predicted play counts.
    temp = temp.sort_values('user_predictions', ascending=False)
    print('\nBelow are the recommended songs for user(user_id = {}):\n'.format(user_index))
    print(temp['user_predictions'].head(num_recommendations))
```

```
user_index = 121
num_recommendations = 10
recommend_songs(user_index, interactions_matrix_new, songs_predicted_play_count, num_recommen
```

```
#Create a dataframe containing average actual play counts and avearge predicted play counts o
rmse_df = pd.concat([interactions_matrix_new.mean(), songs_predicted_play_count.mean()], axis
rmse_df.columns = ['Avg_actual_play_count', 'Avg_predicted_play_count']
rmse_df
```

```
RMSE = mean_squared_error(rmse_df['Avg_actual_play_count'], rmse_df['Avg_predicted_play_count
print('\nRMSE SVD Model = {} \n'.format(RMSE))
```

RMSE **decreases** as I change the number of latent features from 10 to 16. Now increase latent features to try.

```
for i in range(10,100):
    s_final, u_final, vt_final = np.diag(s[:i]), u[:, :i], vt[:i, :]
    songs_predicted_play_count = np.around(np.matmul(np.matmul(u_final, s_final), vt_final))
    songs_predicted_play_count = pd.DataFrame(abs(songs_predicted_play_count), columns = inte
```

```
    RMSE = mean_squared_error(rmse_df['Avg_actual_play_count'], rmse_df['Avg_predicted_play_c
    print('\nRMSE SVD Model = {} \n'.format(RMSE))
    def recommend_songs(user_idx, interactions_matrix_new, preds_df, num_recommendations):

    # Get and sort the user's ratings from the actual and predicted interaction matrix
        sorted_user_play_count = interactions_matrix_new.loc[user_idx].sort_values(ascending=
        sorted_user_predictions = preds_df.loc[user_idx].sort_values(ascending=False)

    #Creating a dataframe with actual and predicted ratings columns
        temp = pd.concat([sorted_user_play_count, sorted_user_predictions], axis=1)
        temp.index.name = 'Recommended Songs'
        temp.columns = ['user_play_count', 'user_predictions']

    #Filtering the dataframe where actual play count are 0 which implies that the user has no
        temp = temp.loc[temp.user_play_count == 0]

    #Recommending movies with top predicted play counts.
        temp = temp.sort_values('user_predictions', ascending=False)
        print('\nBelow are the recommended songs for user(user_id = {}):\n'.format(user_index
        print(temp['user_predictions'].head(num_recommendations))
#Create a dataframe containing average actual play counts and avearge predicted play counts o
    rmse_df = pd.concat([interactions_matrix_new.mean(), songs_predicted_play_count.mean()],
    rmse_df.columns = ['Avg_actual_play_count', 'Avg_predicted_play_count']
```

This gives RMSE for different number features ranging from 10 to 100.