Music Recommendation System

Problem Definition

The Context:

Music has always been a staple in society. For almost anyone, music can be an escape, an outlet, or an expression. It is probably one of, if not the, most interacted with form of media for many people. With the advent of streaming platforms, more music is accessible to people now than ever before.

As music becomes more accessible, it becomes even more paramount that companies can key on what will keep people interacting with their app and the music on their app more than ever. Not only companies like Spotify and Apple with Apple music, but artists as well rely on people interacting with these streaming platforms to generate revenue. With so many more songs becoming available, it's become quite tedious to continue to find music similar to one's tastes. That makes it **even more important that users have functionality available to them that streamlines the process of them finding music within the app**.

It's become quite clear throughout history that makng something more convenient for your user is the best way to drive engagement, and that's what makes recommendations on streaming apps like Spotify and Apple Music so important.

To that end, it becomes important to be able to ensure that their platforms do a good job of both making sure that users can interact with the music they enjoy as easily as possible, but also leveraging algorithms to **be able to recommend new content to users to maintain and even increase engagement, both improving the app's quality and ease of use for the listener. The more time people spend on the app, the more revenue they generate and the more artists benefit as well.**

The objective:

To that end, the **objective of this project is to develop a recommendation system leveraging data science techniques to predict the top 10 songs for a user based on how likely they are to listen to those songs**.

This is based on a number of key factors, including **what kinds of songs they've listened to the most, and their tendencies when it comes to listening to music in general**.

This will increase user satisfaction and engagement by delivering personalized and accurate music recommendations, enhance the experience of a user by helping them navigating and ever increasing library of music.

The key questions:

In general, we want to ensure that our recommendation system is as accurate as possible for a company to deploy to their users so that they can be connfident in the recommendations they're getting. A poor recommendation system would not only not benefit the users and the company, but would damage the reputation of the product and make users less likely to listen to music on that app. To that end, here are some key questions we want to answer?

- What key factors contribute to a user's music preferences?
  - What data can we use to understand user preferences?
- How do users display their like/dislike of music?
  - Do they not interact with old kinds of music?
  - Are there certain genres that users don't interact with?
- How can we ensure that our recommendations are not only accurate, but precise?
  - How much error do we have in our recommendation system, and how egregious is said error?

The problem formulation:

- What is it that we are trying to solve using data science?

First, we will load, clean/process and understand the data we've been given in our datasets.

Then, we will use exploratory data analysis to try and identify key features that allow us to answer our first question above (what features can we identify that will allow us to understand a user's music preference).

Then, we will leverage the different kinds of Recommendation Systems we've learned in the past few weeks:

- Rank/Popularity based
- User/User collaborative filtering
- Item/Item collaborative filtering
- Model Based/Matrix Factorization
- Clustering based
- Content based

We will apply these models to the dataset, evaluate them, and try to identify which one is most effective in recommendation. We will use F1 score, RMSE, precision and recall values to evaluate the models.

We will internalize our results make a final recommendation on which recommendation system would be the best approach in improving user experience, thus improving the product.

Data Dictionary

The core data is the Taste Profile Subset released by the Echo Nest as part of the Million Song Dataset. There are two files in this dataset. The first file contains the details about the song id, titles, release, artist name, and the year of release. The second file contains the user id, song id, and the play count of users.

song_data

song_id - A unique id given to every song

title - Title of the song

Release - Name of the released album

Artist_name - Name of the artist

year - Year of release

count_data

user _id - A unique id given to the user

song_id - A unique id given to the song

play_count - Number of times the song was played

Data Source

http://millionsongdataset.com/

Importing Libraries and the Dataset

In [ ]:
```
# Mounting the drive
from google.colab import drive
drive.mount('/content/drive')
```

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

In [ ]:
```
# Used to ignore the warning given as output of the code
import warnings
warnings.filterwarnings('ignore')

# Basic libraries of python for numeric and dataframe computations
import numpy as np
import pandas as pd

# Import Matplotlib the Basic library for data visualization
import matplotlib.pyplot as plt

# Import seaborn - Slightly advanced library for data visualization
import seaborn as sns

# Import the required library to compute the cosine similarity between two vectors
from sklearn.metrics.pairwise import cosine_similarity
```

```
# Import defaultdict from collections A dictionary output that does not raise a key er
from collections import defaultdict

# Impoort mean_squared_error : a performance metrics in sklearn
from sklearn.metrics import mean_squared_error
```

Load the dataset

In [ ]:
```
# Importing the datasets
count_df = pd.read_csv('/content/drive/MyDrive/MIT Data Science & AI Course Notes/Caps
song_df = pd.read_csv('/content/drive/MyDrive/MIT Data Science & AI Course Notes/Capst
```

Understanding the data by viewing a few observations

In [ ]:
```
# See top 10 records of count_df data
count_df.head(10)
```

Out[ ]:

	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
4	4	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SODACBL12A8C13C273	1
5	5	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SODDNQT12A6D4F5F7E	5
6	6	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SODXRTY12AB0180F3B	1
7	7	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOFGUAY12AB017B0A8	1
8	8	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOFRQTD12A81C233C0	1
9	9	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOHQWYZ12A6D4FA701	1

In [ ]:
```
# See top 10 records of song_df data
song_df.head(10)
```

Out[ ]:

	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 Aspens	Rene Ablaze Presents Winter Sessions	Der Mystic	0
5	SOZVAPQ12A8C13B63C	Symphony No. 1 G minor "Sinfonie Serieuse"/All...	Berwald: Symphonies Nos. 1/2/3/4	David Montgomery	0
6	SOQVRHI12A6D4FB2D7	We Have Got Love	Strictly The Best Vol. 34	Sasha / Turbulence	0
7	SOEYRFT12AB018936C	2 Da Beat Ch'yall	Da Bomb	Kris Kross	1993
8	SOPMIYT12A6D4F851E	Goodbye	Danny Boy	Joseph Locke	0
9	SOJCFMH12A8C13B0C2	Mama_ mama can't you see ?	March to cadence with the US marines	The Sun Harbor's Chorus-Documentary Recordings	0

Let us check the data types and and missing values of each column

In [ ]:
```
# See the info of the count_df data
count_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000000 entries, 0 to 1999999
Data columns (total 4 columns):
 #   Column      Dtype 
---  ------      ----- 
 0   Unnamed: 0  int64 
 1   user_id     object
 2   song_id     object
 3   play_count  int64 
dtypes: int64(2), object(2)
memory usage: 61.0+ MB
```

In [ ]:
```
# See the info of the song_df data
song_df.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
```

Observations and Insights:_

Count_Df: 2,000,000 entries and 4 columns. Data types:

- user_id : object
- song_id: object
- play_count: int64
- Unnamed: 0: int64 It looks like the unnamed column can be dropped

Song_df: 1,000,000 entries and 5 columns. Data types:

- Song_id: object
- title: object
- release: object
- artist_name: object
- year: int64 Title and release columns have a few missing values

In [ ]:
```
# Left merge the count_df and song_df data on "song_id". Drop duplicates from song_df
df = pd.merge(count_df, song_df.drop_duplicates(['song_id']), on='song_id', how = 'lef

# Drop the column 'Unnamed: 0'
df.drop('Unnamed: 0', axis = 1, inplace = True)
## Name the obtained dataframe as "df"
df
```

Out[ ]:

	user_id	song_id	play_count	title
0	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOAKIMP12A8C130995	1	The Cove
1	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBBMDR12A8C13253B	2	Entre Dos Aguas
2	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBXHDL12A81C204C0	1	Stronger
3	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBYHAJ12A6701BF1D	1	Constellations
4	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SODACBL12A8C13C273	1	Learn To Fly
...	...	...	...	...
1999995	d8bfd4ec88f0f3773a9e022e3c1a0f1d3b7b6a92	SOJEYPO12AAA8C6B0E	2	Ignorance (Album Version)
1999996	d8bfd4ec88f0f3773a9e022e3c1a0f1d3b7b6a92	SOJJYDE12AF729FC16	4	Two Is Better Than One
1999997	d8bfd4ec88f0f3773a9e022e3c1a0f1d3b7b6a92	SOJKQSF12A6D4F5EE9	3	What I've Done (Album Version)
1999998	d8bfd4ec88f0f3773a9e022e3c1a0f1d3b7b6a92	SOJUXGA12AC961885C	1	Up
1999999	d8bfd4ec88f0f3773a9e022e3c1a0f1d3b7b6a92	SOJYOLS12A8C13C06F	1	Soil_ Soil (Album Version)

2000000 rows × 7 columns

Think About It: As the user_id and song_id are encrypted. Can they be encoded to numeric features?

In [ ]:

```
# Apply label encoding for "user_id" and "song_id"
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()

df['user_id'] = le.fit_transform(df['user_id'])
df['song_id'] = le.fit_transform(df['song_id'])
df.shape
```

Out[ ]:

```
(2000000, 7)
```

Think About It: As the data also contains users who have listened to very few songs and vice versa, is it required to filter the data so that it contains users who have listened to a good count of songs and vice versa?

A dataset of size 2000000 rows x 7 columns can be quite large and may require a lot of computing resources to process. This can lead to long processing times and can make it difficult to train and evaluate your model efficiently. In order to address this issue, it may be necessary to trim down your dataset to a more manageable size.

In [ ]:
```
# Get the column containing the users
users = df.user_id
# Create a dictionary from users to their number of songs
ratings_count = dict()
for user in users:
    # If we already have the user, just add 1 to their rating count
    if user in ratings_count:
        ratings_count[user] += 1
    # Otherwise, set their rating count to 1
    else:
        ratings_count[user] = 1
```

In [ ]:
```
# We want our users to have listened at least 90 songs
RATINGS_CUTOFF = 90
remove_users = []
for user, num_ratings in ratings_count.items():
    if num_ratings < RATINGS_CUTOFF:
        remove_users.append(user)
df = df.loc[~df.user_id.isin(remove_users)]
```

In [ ]:
```
# Get the column containing the songs
songs = df.song_id
# Create a dictionary from songs to their number of users
ratings_count = dict()
for song in songs:
    # If we already have the song, just add 1 to their rating count
    if song in ratings_count:
        ratings_count[song] += 1
    # Otherwise, set their rating count to 1
    else:
        ratings_count[song] = 1
```

In [ ]:
```
# We want our song to be listened by atleast 120 users to be considred
# We want our song to be listened by atleast 120 users to be considred
RATINGS_CUTOFF = 120
remove_songs = []
for song, num_ratings in ratings_count.items():
    if num_ratings < RATINGS_CUTOFF:
        remove_songs.append(song)
df_final= df.loc[~df.song_id.isin(remove_songs)]
```

In [ ]:
```
# Drop records with play_count more than(>) 5
df_final=df_final[df_final.play_count<=5]
```

In [ ]:
```
# Check the shape of the data
df_final.shape
```

Out[ ]:
```
(117876, 7)
```

Exploratory Data Analysis

Let's check the total number of unique users, songs, artists in the data

Total number of unique user id

In [ ]:
```
# Display total number of unique user_id
df_final['user_id'].nunique()
```

Out[ ]:
```
3155
```

Total number of unique song id

In [ ]:
```
# Display total number of unique song_id
df_final['song_id'].nunique()
```

Out[ ]:
```
563
```

Total number of unique artists

In [ ]:
```
# Display total number of unique artists
df_final['artist_name'].nunique()
```

Out[ ]:
```
232
```

Observations and Insights:__

Df_final unique counts:

- User_id: There are 3155 unique users
- Song_id: There are 563 unique songs
- Artist_name: There are 232 unique artists

Let's find out about the most interacted songs and interacted users

Most interacted songs

In [ ]:
```
df_final['song_id'].value_counts()
```

Out[ ]:
```
8582    751
352     748
2220    713
1118    662
4152    652
       ...
9048    103
6450    102
990     101
4831     97
8324     96
Name: song_id, Length: 563, dtype: int64
```

Most interacted users

In [ ]:
```
df_final['user_id'].value_counts()
```

Out[ ]:
```
61472    243
15733    227
37049    202
9570     184
23337    177
        ...
19776      1
45476      1
17961      1
14439      1
10412      1
Name: user_id, Length: 3155, dtype: int64
```

Observations and Insights:__

The most interacted with song is 8582 at 751 plays, and the most interacted with user is 61472 at 243 song plays.

Songs played in a year

In [ ]:
```
# Find out the number of songs played in a year
  # Hint: Use groupby function on the 'year' column
songs_in_a_year = df_final.groupby('year').count()['title']
songs_per_year = pd.DataFrame(songs_in_a_year)
songs_per_year.drop(songs_per_year.index[0], inplace = True)
songs_per_year.tail()
```

Out[ ]:

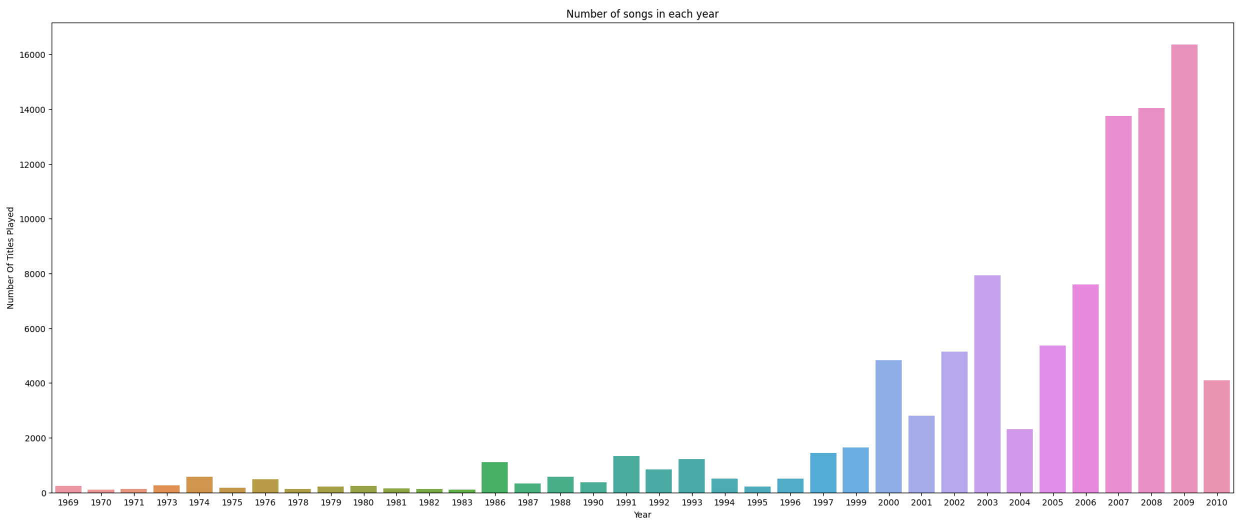
year	title
2006	7592
2007	13750
2008	14031
2009	16351
2010	4087

In [ ]:
```
# Create a barplot plot with y label as "number of titles played" and x -axis year
# Set the figure size
plt.figure(figsize = (25,10))
sns.barplot(x = songs_per_year.index, y = 'title', data = songs_per_year, estimator =

# Set the x label of the plot
plt.xlabel('Year')

# Set the y label of the plot
plt.ylabel('Number Of Titles Played')

# Show the plot
plt.title('Number of songs in each year')
plt.show()
```



Observations and Insights:__

We can see that there is a significant increase in the number of songs that were played per year, roughly post 1999 where we see a huge spike into the year 2000. In our datasets, we see that the years 2009, 2008 and 2007 had the highest number of songs played in that order. We see the lowest number of songs were played in the 1970s and early 1980s, with a pickup being seen in the 1980s and 1990s. This may be because we have less data about what songs were being playedd back then.

Think About It: What other insights can be drawn using exploratory data analysis?

Now that we have explored the data, let's apply different algorithms to build recommendation systems.

Note: Use the shorter version of the data, i.e., the data after the cutoffs as used in Milestone 1.

Building various models

Popularity-Based Recommendation Systems

Let's take the count and sum of play counts of the songs and build the popularity recommendation systems based on the sum of play counts.

The use case for this might be for someone like a new user where we don't have any pre-existing data with which to work with.

In [ ]:
```
# Calculating average play_count
       # Hint: Use groupby function on the song_id column
average_play_count = df_final.groupby('song_id').mean()['play_count']

# Calculating the frequency a song is played
      # Hint: Use groupby function on the song_id column
play_frequency = df_final.groupby('song_id').count()['play_count']
```

In [ ]:
```
# Making a dataframe with the average_count and play_freq
play_counts_df = pd.DataFrame({'average_count': average_play_count, 'play_freq': play_
# Let us see the first five records of the final_play dataset
play_counts_df.head()
```

Out[ ]:

song_id	average_count	play_freq
21	1.622642	265
22	1.492424	132
52	1.729216	421
62	1.728070	114
93	1.452174	115

Now, let's create a function to find the top n songs for a recommendation based on the average play count of song. We can also add a threshold for a minimum number of playcounts for a song to be considered for recommendation.

In [ ]:
```
# Build the function to find top n songs
def top_n_songs(data, n, min_interaction = 100):
  recommendations = data[data['play_freq'] > min_interaction]

  recommendations = recommendations.sort_values(by = 'average_count', ascending = Fals

  return recommendations.index[:n]
```

In [ ]:
```
# Recommend top 10 songs using the function defined above
list(top_n_songs(play_counts_df, 10, 50))
```

Out[ ]:
```
[7224, 8324, 6450, 9942, 5531, 5653, 8483, 2220, 657, 614]
```

User User Similarity-Based Collaborative Filtering

To build the user-user-similarity-based and subsequent models we will use the "surprise" library.

In [ ]:
```
# Install the surprise package using pip. Uncomment and run the below code to do the s
!pip install surprise
```

```
Collecting surprise
  Downloading surprise-0.1-py2.py3-none-any.whl (1.8 kB)
Collecting scikit-surprise (from surprise)
  Downloading scikit-surprise-1.1.3.tar.gz (771 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 772.0/772.0 kB 11.6 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: joblib>=1.0.0 in /usr/local/lib/python3.10/dist-packag
es (from scikit-surprise->surprise) (1.3.2)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-packag
es (from scikit-surprise->surprise) (1.23.5)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-package
s (from scikit-surprise->surprise) (1.11.4)
Building wheels for collected packages: scikit-surprise
  Building wheel for scikit-surprise (setup.py) ... done
  Created wheel for scikit-surprise: filename=scikit_surprise-1.1.3-cp310-cp310-linux
_x86_64.whl size=3163753 sha256=737f3d4efcbf2e35f9368f27d4a1d791c12bf34d09bb4071eeadf
65a753c1b42
  Stored in directory: /root/.cache/pip/wheels/a5/ca/a8/4e28def53797fdc4363ca4af740db
15a9c2f1595ebc51fb445
Successfully built scikit-surprise
Installing collected packages: scikit-surprise, surprise
Successfully installed scikit-surprise-1.1.3 surprise-0.1
```

In [ ]:
```
# Import necessary libraries

# To compute the accuracy of models
from surprise import accuracy

# This class is used to parse a file containing play_counts, data should be in structu
from surprise.reader import Reader

# Class for loading datasets
from surprise.dataset import Dataset

# For tuning model hyperparameters
from surprise.model_selection import GridSearchCV

# For splitting the data in train and test dataset
from surprise.model_selection import train_test_split

# For implementing similarity-based recommendation system
from surprise.prediction_algorithms.knns import KNNBasic

# For implementing matrix factorization based recommendation system
from surprise.prediction_algorithms.matrix_factorization import SVD

# For implementing KFold cross-validation
from surprise.model_selection import KFold

# For implementing clustering-based recommendation system
from surprise import CoClustering
```

Some useful functions

Below is the function to calculate precision@k and recall@k, RMSE, and F1_Score@k to evaluate the model performance.

Think About It: Which metric should be used for this problem to compare different models?

In [ ]:
```
# The function to calulate the RMSE, precision@k, recall@k, and F_1 score
def precision_recall_at_k(model, k = 30, threshold = 1.5):
    """Return precision and recall at k metrics for each user"""

    # First map the predictions to each user.
    user_est_true = defaultdict(list)

    # Making predictions on the test data
    predictions=model.test(testset)

    for uid, _, true_r, est, _ in predictions:
        user_est_true[uid].append((est, true_r))

    precisions = dict()
    recalls = dict()
    for uid, user_ratings in user_est_true.items():

        # Sort user ratings by estimated value
        user_ratings.sort(key = lambda x : x[0], reverse = True)

        # Number of relevant items
        n_rel = sum((true_r >= threshold) for (_, true_r) in user_ratings)

        # Number of recommended items in top k
        n_rec_k = sum((est >= threshold) for (est, _) in user_ratings[ : k])

        # Number of relevant and recommended items in top k
        n_rel_and_rec_k = sum(((true_r >= threshold) and (est >= threshold))
                              for (est, true_r) in user_ratings[ : k])

        # Precision@K: Proportion of recommended items that are relevant
        # When n_rec_k is 0, Precision is undefined. We here set Precision to 0 when n

        precisions[uid] = n_rel_and_rec_k / n_rec_k if n_rec_k != 0 else 0

        # Recall@K: Proportion of relevant items that are recommended
        # When n_rel is 0, Recall is undefined. We here set Recall to 0 when n_rel is

        recalls[uid] = n_rel_and_rec_k / n_rel if n_rel != 0 else 0

    # Mean of all the predicted precisions are calculated
    precision = round((sum(prec for prec in precisions.values()) / len(precisions)), 3

    # Mean of all the predicted recalls are calculated
    recall = round((sum(rec for rec in recalls.values()) / len(recalls)), 3)

    accuracy.rmse(predictions)

    # Command to print the overall precision
    print('Precision: ', precision)

    # Command to print the overall recall
    print('Recall: ', recall)
```

```
    # Formula to compute the F-1 score
    print('F_1 score: ', round((2 * precision * recall) / (precision + recall), 3))
```

Think About It: In the function precision_recall_at_k above the threshold value used is 1.5. How precision and recall are affected by changing the threshold? What is the intuition behind using the threshold value of 1.5?

In [ ]:
```
# Instantiating Reader scale with expected rating scale
 #use rating scale (0, 5)
reader = Reader(rating_scale = (0, 5))

# Loading the dataset
 # Take only "user_id","song_id", and "play_count"
data = Dataset.load_from_df(df_final[['user_id', 'song_id', 'play_count']], reader)

# Splitting the data into train and test dataset
 # Take test_size = 0.4, random_state = 42
trainset, testset = train_test_split(data, test_size = 0.4, random_state = 42)
```

Think About It: How changing the test size would change the results and outputs?

In [ ]:
```
# Build the default user-user-similarity model
sim_options = {'name': 'cosine',
               'user_based': True}

# KNN algorithm is used to find desired similar items
 # Use random_state = 1
sim_user_user = KNNBasic(sim_options = sim_options, verbose = False, random_state = 1)

# Train the algorithm on the trainset, and predict play_count for the testset
sim_user_user.fit(trainset)

# Let us compute precision@k, recall@k, and f_1 score with k = 30
 # Use sim_user_user model
precision_recall_at_k(sim_user_user)
```

```
RMSE: 1.0878
Precision:  0.396
Recall:  0.692
F_1 score:  0.504
```

Observations and Insights:_ RMSE: 1.0878

- This indicates how far the predicted play_couont is from the actual play_count (it's a give or take metric). Just looking at the playcounts in the dataset, this seems somewhat high

Precision: 0.396

- This indicates that out of all recommended songs, 39.6% are relevant to users. This, again is not a good outcome.

Recall: 0.692

- Of the songs that are relevant to the listener, 69.2% are recommended. We would like this to be higher.

F1 score: 0.504

- This indicates that a little bit over half of the recommended songs are relevant and were recommended to the user.

This model's performance isn't terrible, but we likely can do much better. We can see it's performance on a specific case below.

In [ ]:
```
# Predicting play_count for a sample user with a listened song
# Use any user id  and song_id
sim_user_user.predict(47786, 9351, r_ui = 2, verbose = True)
```

```
user: 47786      item: 9351       r_ui = 2.00   est = 1.68   {'actual_k': 40, 'was_impossible': False}
```

Out[ ]:
```
Prediction(uid=47786, iid=9351, r_ui=2, est=1.6826124775052502, details={'actual_k': 40, 'was_impossible': False})
```

In [ ]:
```
# Predicting play_count for a sample user with a song not-listened by the user
 #predict play_count for any sample user
sim_user_user.predict(47786, 512, verbose = True)
```

```
user: 47786      item: 512        r_ui = None   est = 1.81   {'actual_k': 40, 'was_impossible': False}
```

Out[ ]:
```
Prediction(uid=47786, iid=512, r_ui=None, est=1.809303404882815, details={'actual_k': 40, 'was_impossible': False})
```

Observations and Insights:_ Just by using examples we can see with a glance at the data, we can see that for user 47786 and song 9351 (which they listened to twice), the model actually predicts 1.68 listens, which isn't that far off from the actual playcount, slightly underestimating. We see that the model predicts 1.81 listens for song 512. These both used an "actual_k" of 40, the value of K in the KNN that is used while training.

We can use GridSearchCV tuning to try and improve the model.

Now, let's try to tune the model and see if we can improve the model performance.

In [ ]:
```
# Setting up parameter grid to tune the hyperparameters
param_grid = {'k': [10, 20, 30], 'min_k': [3, 6, 9],
              'sim_options': {'name': ["cosine", 'pearson', "pearson_baseline"],
                              'user_based': [True], "min_support": [2, 4]}}

# Performing 3-fold cross-validation to tune the hyperparameters
gs = GridSearchCV(KNNBasic, param_grid, measures = ['rmse'], cv = 3, n_jobs = -1)

# Fitting the data
 # Use entire data for GridSearch
gs.fit(data)

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

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

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-37-3c558cc7bc89> in <cell line: 11>()
      9 # Fitting the data
     10  # Use entire data for GridSearch
---> 11 gs.fit(trainset)
     12
     13 # Best RMSE score

/usr/local/lib/python3.10/dist-packages/surprise/model_selection/search.py in fit(sel
f, data)
     98                 self.return_train_measures,
     99             )
--> 100             for params, (trainset, testset) in product(
    101                 self.param_combinations, cv.split(data)
    102             )

/usr/local/lib/python3.10/dist-packages/surprise/model_selection/split.py in split(se
lf, data)
     90         """
     91
---> 92         if self.n_splits > len(data.raw_ratings) or self.n_splits < 2:
     93             raise ValueError(
     94                 "Incorrect value for n_splits={}. "

AttributeError: 'Trainset' object has no attribute 'raw_ratings'
```

In [ ]:

```
# Train the best model found in above gridsearch
sim_options = {'name': 'pearson_baseline',
               'user_based': True}

# Create another instance of KNNBasic with optimal hyperparameter values based on abov
sim_user_user_optimized = KNNBasic(sim_options = sim_options, k = 30, min_k = 9, rando

#Train the algorithm and compute new results
sim_user_user_optimized.fit(trainset)
precision_recall_at_k(sim_user_user_optimized)
```

```
RMSE: 1.0521
Precision:  0.413
Recall:  0.721
F_1 score:  0.525
```

Observations and Insights:_

RMSE: 1.0521

Precision: 0.413

Recall: 0.721

F1 score: 0.525

With the new parameters, we can see that we have slight improvements across the board, and the model has improved with hyperparameter tuning.

In [ ]:
```
# Predict the play count for a user who has listened to the song. Take user_id 6958, s
sim_user_user_optimized.predict(6958, 1671, r_ui = 2, verbose = True)
```

```
user: 6958       item: 1671       r_ui = 2.00   est = 1.96   {'actual_k': 24, 'was_im
possible': False}
```

Out[ ]:
```
Prediction(uid=6958, iid=1671, r_ui=2, est=1.962926073914969, details={'actual_k': 2
4, 'was_impossible': False})
```

In [ ]:
```
# Predict the play count for a song that is not listened to by the user (with user_id
sim_user_user_optimized.predict(6958, 512, verbose = True)
```

```
user: 6958       item: 512        r_ui = None   est = 1.02   {'actual_k': 30, 'was_im
possible': False}
```

Out[ ]:
```
Prediction(uid=6958, iid=512, r_ui=None, est=1.0205736461605335, details={'actual_k':
30, 'was_impossible': False})
```

Observations and Insights:__ We can see that the prediction for user 6958 and song 1671 is very close to the actual value of 2 (with a value of 1.96).

Think About It: Along with making predictions on listened and unknown songs can we get 5 nearest neighbors (most similar) to a certain song?

In [ ]:
```
# Use inner id 0
sim_user_user_optimized.get_neighbors(0, 5)
```

Out[ ]:
```
[42, 1131, 17, 186, 249]
```

Below we will be implementing a function where the input parameters are:

- data: A **song** dataset
- user_id: A user-id **against which we want the recommendations**
- top_n: The **number of songs we want to recommend**
- algo: The algorithm we want to use **for predicting the play_count**
- The output of the function is a **set of top_n items** recommended for the given user_id based on the given algorithm

In [ ]:
```
def get_recommendations(data, user_id, top_n, algo):

    # Creating an empty list to store the recommended product ids
    recommendations = []

    # Creating an user item interactions matrix
    user_item_interactions_matrix = data.pivot_table(index = 'user_id', columns = 'sor

    # Extracting those business ids which the user_id has not visited yet
    non_interacted = user_item_interactions_matrix.loc[user_id][user_item_interactions

    # Looping through each of the business ids which user_id has not interacted yet
    for item_id in non_interacted:

        # Predicting the ratings for those non visited restaurant ids by this user
        estimation = algo.predict(user_id, item_id).est

        # Appending the predicted ratings
        recommendations.append((item_id, estimation))
```

```
    # Sorting the predicted ratings in descending order
    recommendations.sort(key = lambda x : x[1], reverse = True)

    # Returing top n highest predicted rating products for this user
    return recommendations[:top_n]
```

In [ ]:
```
# Make top 5 recommendations for user_id 47786 with a similarity-based recommendation
recommendations = get_recommendations(df_final, 47786, 5, sim_user_user)
```

In [ ]:
```
# Building the dataframe for above recommendations with columns "song_id" and "predict
pd.DataFrame(recommendations, columns = ['song_id', 'predicted_ratings'])
```

Out[ ]:

	song_id	predicted_ratings
0	7224	3.341232
1	8324	2.896419
2	6450	2.706945
3	614	2.575000
4	5653	2.447284

Observations and Insights:__ This shows us user 47786's top 5 predicted songs based on our user_user recommendation systemm.

Correcting the play_counts and Ranking the above songs

In [ ]:
```
def ranking_songs(recommendations, final_rating):
  # Sort the songs based on play counts
  ranked_songs = final_rating.loc[[items[0] for items in recommendations]].sort_values

  # Merge with the recommended songs to get predicted play_count
  ranked_songs = ranked_songs.merge(pd.DataFrame(recommendations, columns = ['song_id'

  # Rank the songs based on corrected play_counts
  ranked_songs['corrected_ratings'] = ranked_songs['predicted_ratings'] - 1 / np.sqrt(

  # Sort the songs based on corrected play_counts
  ranked_songs = ranked_songs.sort_values('corrected_ratings', ascending = False)

  return ranked_songs
```

Think About It: In the above function to correct the predicted play_count a quantity 1/np.sqrt(n) is subtracted. What is the intuition behind it? Is it also possible to add this quantity instead of subtracting?

As 1 / sqrt(n) is a regularization term that penalizes items with a higher play frequency, we reduce popularity bias in these recommendations. This will tend to make the recommendation system less biased towards popular songs and more in tune with the user's preferences. In terms of whether we could add it, it really depends on what the goal of the specific usee case is. However, in my own personal preference, I would be a user who would want similar songs that are also popular. If a user wants similar songs to what they listen to that are also popular, in that case it would make sense to add them.

In [ ]:
```
# Applying the ranking_songs function on the final_play data
ranking_songs(recommendations, play_counts_df)
```

Out[ ]:

	song_id	play_freq	predicted_ratings	corrected_ratings
2	7224	107	3.341232	3.244558
4	8324	96	2.896419	2.794357
3	6450	102	2.706945	2.607930
0	614	373	2.575000	2.523222
1	5653	108	2.447284	2.351059

Observations and Insights:__

In this case, it makes sense that we would subtract the regularization factor as we don't want to exceed 5 plays for each song. The corrected ratings we've calculated allows us to also take into account how many users have played the song, and not just how many plays the song has in general. Although in general, songs with higher play counts per person could be considered better, a song that's been played fewer times by significantly more people might be significant to a user, which is why the corrected_ratings column is important.

Item Item Similarity-based collaborative filtering recommendation systems

In [ ]:
```
# Apply the item-item similarity collaborative filtering model with random_state = 1 a
sim_options = {'name': 'cosine',
               'user_based': False}

# Using KNN algorithm is to find similar items
sim_item_item = KNNBasic(sim_options = sim_options, verbose = False, random_state = 1)

# Training the algorithm
sim_item_item.fit(trainset)

# Computing rmse, precision@k, recall@k, and f_1 score
precision_recall_at_k(sim_item_item)
```

```
RMSE: 1.0394
Precision:  0.307
Recall:  0.562
F_1 score:  0.397
```

Observations and Insights:__ At first glance, we're getting significantly worse performance across the board for item/item as opposed to user/user (without optimization) besides the RMSE, which is slightly lower for item/item. Let's check our predictions and try to tune with hyperparameter tuning.

In [ ]:
```
# Predicting play count for a sample user_id 6958 and song (with song_id 1671) heard b
sim_item_item.predict(6958, 1671, r_ui = 2, verbose = True)
df_final
```

```
user: 6958       item: 1671       r_ui = 2.00   est = 1.36   {'actual_k': 20, 'was_im
possible': False}
```

Out[ ]:

	user_id	song_id	play_count	title	release	artist_name	year
200	6958	447	1	Daisy And Prudence	Distillation	Erin McKeown	2000
202	6958	512	1	The Ballad of Michael Valentine	Sawdust	The Killers	2004
203	6958	549	1	I Stand Corrected (Album)	Vampire Weekend	Vampire Weekend	2007
204	6958	703	1	They Might Follow You	Tiny Vipers	Tiny Vipers	2007
205	6958	719	1	Monkey Man	You Know I'm No Good	Amy Winehouse	2007
...	...	...	...	...	...	...	...
1999734	47786	9139	1	Half Of My Heart	Battle Studies	John Mayer	0
1999736	47786	9186	1	Bitter Sweet Symphony	Bitter Sweet Symphony	The Verve	1997
1999745	47786	9351	2	The Police And The Private	Live It Out	Metric	2005
1999755	47786	9543	1	Just Friends	Back To Black	Amy Winehouse	2006
1999765	47786	9847	1	He Can Only Hold Her	Back To Black	Amy Winehouse	2006

117876 rows × 7 columns

In [ ]:
```
# Predict the play count for a user that has not listened to the song (with song_id 16
sim_item_item.predict(47786, 1671, verbose = True)
```

```
user: 47786      item: 1671       r_ui = None   est = 1.94   {'actual_k': 40, 'was_im
possible': False}
```

Out[ ]:
```
Prediction(uid=47786, iid=1671, r_ui=None, est=1.9365719326237714, details={'actual_
k': 40, 'was_impossible': False})
```

Observations and Insights:__ The initial predictions for item/item is also lower for the sample user_id and song_id. Let's try and improve it

In [ ]:
```
# Apply grid search for enhancing model performance

# Setting up parameter grid to tune the hyperparameters
param_grid = {'k': [10, 20, 30], 'min_k': [3, 6, 9],
              'sim_options': {'name': ["cosine", 'pearson', "pearson_baseline"],
                              'user_based': [False], "min_support": [2, 4]}}

# Performing 3-fold cross-validation to tune the hyperparameters
```

```
gs = GridSearchCV(KNNBasic, param_grid, measures = ['rmse'], cv = 3, n_jobs = -1)
# Fitting the data
gs.fit(data)

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

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

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-59-8920feb9897c> in <cell line: 11>()
      9 gs = GridSearchCV(KNNBasic, param_grid, measures = ['rmse'], cv = 3, n_jobs =
-1)
     10 # Fitting the data
---> 11 gs.fit(data)
     12 
     13 # Find the best RMSE score

/usr/local/lib/python3.10/dist-packages/surprise/model_selection/search.py in fit(sel
f, data)
    102             )
    103         )
--> 104         out = Parallel(
    105             n_jobs=self.n_jobs,
    106             pre_dispatch=self.pre_dispatch,

/usr/local/lib/python3.10/dist-packages/joblib/parallel.py in __call__(self, iterabl
e)
   1950         next(output)
   1951 
-> 1952         return output if self.return_generator else list(output)
   1953 
   1954     def __repr__(self):

/usr/local/lib/python3.10/dist-packages/joblib/parallel.py in _get_outputs(self, iter
ator, pre_dispatch)
   1593 
   1594             with self._backend.retrieval_context():
-> 1595                 yield from self._retrieve()
   1596 
   1597         except GeneratorExit:

/usr/local/lib/python3.10/dist-packages/joblib/parallel.py in _retrieve(self)
   1705                 (self._jobs[0].get_status(
   1706                     timeout=self.timeout) == TASK_PENDING)):
-> 1707                 time.sleep(0.01)
   1708                 continue
   1709 

KeyboardInterrupt: 
```

Think About It: How do the parameters affect the performance of the model? Can we improve the performance of the model further? Check the list of hyperparameters here.

In [ ]:

```
# Apply the best model found in the grid search
sim_options = {'name': 'pearson_baseline', 'user_based': False}
```

```
# Creating an instance of KNNBasic with optimal hyperparameter values
sim_item_item_optimized = KNNBasic(sim_options = sim_options, k = 30, min_k = 6, rando

# Training the algorithm on the trainset
sim_item_item_optimized.fit(trainset)

# computing our new results
precision_recall_at_k(sim_item_item_optimized)
```

```
RMSE: 1.0328
Precision:  0.408
Recall:  0.665
F_1 score:  0.506
```

Observations and Insights:__ Again, we got slight improvements across the board, with the lowest RMSE we've seen, but all of the other values aren't quite as good as the optimized user/user system.

In [ ]:
```
# Predict the play_count by a user(user_id 6958) for the song (song_id 1671)
sim_item_item_optimized.predict(6958, 1671, r_ui = 2, verbose = True)
```

```
user: 6958       item: 1671       r_ui = 2.00   est = 1.96   {'actual_k': 10, 'was_im
possible': False}
```

Out[ ]:
```
Prediction(uid=6958, iid=1671, r_ui=2, est=1.9634957386781853, details={'actual_k': 1
0, 'was_impossible': False})
```

In [ ]:
```
# Predicting play count for a sample user_id 6958 with song_id 3232 which is not heard
sim_item_item_optimized.predict(6958, 3232, verbose = True)
```

```
user: 6958       item: 3232       r_ui = None   est = 1.28   {'actual_k': 10, 'was_im
possible': False}
```

Out[ ]:
```
Prediction(uid=6958, iid=3232, r_ui=None, est=1.2759946618244609, details={'actual_
k': 10, 'was_impossible': False})
```

Observations and Insights:__ We actually got the same level of accuracy with this song as we did before (1.96 compared to the actual 2.0), so we got significant improvement with this model.

In [ ]:
```
# Find five most similar items to the item with inner id 0
sim_item_item_optimized.get_neighbors(0, 5)
```

Out[ ]:
```
[124, 523, 173, 205, 65]
```

In [ ]:
```
# Making top 5 recommendations for any user_id  with item_item_similarity-based recomm
recommendations = get_recommendations(df_final, 47786, 5, sim_item_item_optimized)
```

In [ ]:
```
# Building the dataframe for above recommendations with columns "song_id" and "predict
pd.DataFrame(recommendations, columns = ['song_id', 'predicted_ratings'])
```

Out[ ]:

	song_id	predicted_ratings
0	5158	3.565050
1	9447	3.261276
2	6885	3.245754
3	1682	3.024199
4	614	2.921668

In [ ]:
```
# Applying the ranking_songs function
ranking_songs(recommendations, play_counts_df)
```

Out[ ]:

	song_id	play_freq	predicted_ratings	corrected_ratings
3	5158	126	3.565050	3.475963
4	9447	121	3.261276	3.170367
1	6885	164	3.245754	3.167668
2	1682	146	3.024199	2.941438
0	614	373	2.921668	2.869890

Observations and Insights:_ We can see that the corrected ratings and the predicted ratings are pretty similarly in line with what we had from user/user.

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.

In [ ]:
```
# Build baseline model using svd
svd = SVD(random_state = 1)

svd.fit(trainset)

precision_recall_at_k(svd)
```

```
RMSE: 1.0252
Precision:  0.41
Recall:  0.633
F_1 score:  0.498
```

In [ ]:
```
# Making prediction for user (with user_id 6958) to song (with song_id 1671), take r_u
svd.predict(6958, 1671, r_ui = 2, verbose = True)
```

```
user: 6958       item: 1671       r_ui = 2.00   est = 1.27   {'was_impossible': Fals
e}
```

Out[ ]:
```
Prediction(uid=6958, iid=1671, r_ui=2, est=1.267473397214638, details={'was_impossibl
e': False})
```

In [ ]:
```
# Making a prediction for the user who has not listened to the song (song_id 3232)
svd.predict(6958, 3232, verbose = True)
```

```
user: 6958       item: 3232       r_ui = None   est = 1.56   {'was_impossible': Fals
e}
```

Out[ ]:
```
Prediction(uid=6958, iid=3232, r_ui=None, est=1.5561675084403663, details={'was_impos
sible': False})
```

Observations: At a predicted value of 1.27, the model is severely underestimating the rating (worse than the initial values we had for user/user and item/item. However, we can see that at least initially, we have lower RMSE and higher precision, although recall and F1 scores are lower than initial user/user and item/item).

- F1 score is higher than initial item/item

Let's try to improve it

Improving matrix factorization based recommendation system by tuning its hyperparameters

In [ ]:
```
# Set the parameter space to tune
param_grid = {'n_epochs': [10, 20, 30], 'lr_all': [0.001, 0.005, 0.01],
              'reg_all': [0.2, 0.4, 0.6]}

# Performe 3-fold grid-search cross-validation
gs = GridSearchCV(SVD, param_grid, measures = ['rmse'], cv = 3, n_jobs = -1)

# Fitting data
gs.fit(data)

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

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

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-72-7b1571423fed> in <cell line: 9>()
      7 
      8 # Fitting data
----> 9 gs.fit(data)
     10 
     11 # Best RMSE score

/usr/local/lib/python3.10/dist-packages/surprise/model_selection/search.py in fit(sel
f, data)
    102             )
    103         )
--> 104         out = Parallel(
    105             n_jobs=self.n_jobs,
    106             pre_dispatch=self.pre_dispatch,

/usr/local/lib/python3.10/dist-packages/joblib/parallel.py in __call__(self, iterabl
e)
   1950         next(output)
   1951 
-> 1952         return output if self.return_generator else list(output)
   1953 
   1954     def __repr__(self):

/usr/local/lib/python3.10/dist-packages/joblib/parallel.py in _get_outputs(self, iter
ator, pre_dispatch)
   1593 
   1594             with self._backend.retrieval_context():
-> 1595                 yield from self._retrieve()
   1596 
   1597         except GeneratorExit:

/usr/local/lib/python3.10/dist-packages/joblib/parallel.py in _retrieve(self)
   1705                 (self._jobs[0].get_status(
   1706                     timeout=self.timeout) == TASK_PENDING)):
-> 1707                 time.sleep(0.01)
   1708                 continue
   1709 

KeyboardInterrupt:
```

Think About It: How do the parameters affect the performance of the model? Can we improve the performance of the model further? Check the available hyperparameters here.

In [ ]:
```
# Building the optimized SVD model using optimal hyperparameters
svd_algo_optimized = SVD(n_epochs = 30, lr_all = 0.01, reg_all = 0.2, random_state = 1

svd_algo_optimized = svd_algo_optimized.fit(trainset)

precision_recall_at_k(svd_algo_optimized)
```

```
RMSE: 1.0141
Precision:  0.415
Recall:  0.635
F_1 score:  0.502
```

Observations and Insights:_ We can see that everything has improved across the board. We have gotten the highest precision thus far (slightly outstripping optimized user/user) and the lowest RMSE. However, the recall and F1 scores are significantly lower than optimized user/user.

Let's look at our predictions

In [ ]:
```
# Using svd_algo_optimized model to recommend for userId 6958 and song_id 1671
svd_algo_optimized.predict(6958, 1671, r_ui = 2, verbose = True)
```

```
user: 6958       item: 1671       r_ui = 2.00   est = 1.34   {'was_impossible': False}
```

Out[ ]:
```
Prediction(uid=6958, iid=1671, r_ui=2, est=1.3432395286125098, details={'was_impossible': False})
```

In [ ]:
```
# Using svd_algo_optimized model to recommend for userId 6958 and song_id 3232 with un
svd_algo_optimized.predict(6958, 3232, verbose = True)
```

```
user: 6958       item: 3232       r_ui = None   est = 1.44   {'was_impossible': False}
```

Out[ ]:
```
Prediction(uid=6958, iid=3232, r_ui=None, est=1.4425484461176483, details={'was_impossible': False})
```

Observations and Insights:_ Although the play count isn't actually far off (we're looking at 1.34 plays vs something like 1.96 for user/user optimized), the prediction is noticeably worse than the aforementioned models. For the song the user has not listened to, this play count seems in line with the user's history.

In [ ]:
```
# Getting top 5 recommendations for user_id 6958 using "svd_optimized" algorithm
svd_recommendations = get_recommendations(df_final, 47786, 5, svd_algo_optimized)
```

In [ ]:
```
# Ranking songs based on above recommendations
ranking_songs(svd_recommendations, play_counts_df)
```

Out[ ]:

	song_id	play_freq	predicted_ratings	corrected_ratings
3	7224	107	3.187259	3.090586
2	5653	108	2.609927	2.513702
0	1664	388	2.456143	2.405376
4	6450	102	2.501498	2.402483
1	614	373	2.429437	2.377659

Observations and Insights:_ We see in this table the top 5 recommended songs with optimized SVD, again they seem to pretty in line with the predicted ratings as they were in the earlier models.

Cluster Based Recommendation System

In **clustering-based recommendation systems**, we explore the **similarities and differences** in people's tastes in songs based on how they rate different songs. We cluster similar users together and recommend songs to a user based on play_counts from other users in the same cluster.

In [ ]:
```
# Make baseline clustering model
co_clustering_baseline = CoClustering(random_state = 1)

co_clustering_baseline.fit(trainset)

precision_recall_at_k(co_clustering_baseline)
```

```
RMSE: 1.0487
Precision:  0.397
Recall:  0.582
F_1 score:  0.472
```

Observations:

The values here are solid, but a noticeable dropoff from anything we've had previously. Let's try to improve it after looking at our predictions.

In [ ]:
```
# Making prediction for user_id 6958 and song_id 1671
co_clustering_baseline.predict(6958, 1671, r_ui = 2, verbose = True)
```

```
user: 6958       item: 1671       r_ui = 2.00    est = 1.29   {'was_impossible': Fals
e}
```

Out[ ]:
```
Prediction(uid=6958, iid=1671, r_ui=2, est=1.2941824757363074, details={'was_impossib
le': False})
```

In [ ]:
```
# Making prediction for user (userid 6958) for a song(song_id 3232) not heard by the u
co_clustering_baseline.predict(6958, 3232, verbose = True)
```

```
user: 6958       item: 3232       r_ui = None    est = 1.48   {'was_impossible': Fals
e}
```

Out[ ]:
```
Prediction(uid=6958, iid=3232, r_ui=None, est=1.4785259100797417, details={'was_impos
sible': False})
```

The prediction of 1.29 is in line from what we've seen with other non-optimized models.

Improving clustering-based recommendation system by tuning its hyper-parameters

In [ ]:
```
# Set the parameter space to tune
param_grid = {'n_cltr_u': [5, 6, 7, 8], 'n_cltr_i': [5, 6, 7, 8], 'n_epochs': [10, 20,

# Performing 3-fold grid search cross-validation
gs = GridSearchCV(CoClustering, param_grid, measures = ['rmse'], cv = 3, n_jobs = -1)

# Fitting data
gs.fit(data)

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

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

Think About It: How do the parameters affect the performance of the model? Can we improve the performance of the model further? Check the available hyperparameters here.

In [ ]:
```
# Train the tuned Coclustering algorithm
co_clustering_optimized = CoClustering(n_cltr_u = 5, n_cltr_i = 5, n_epochs = 10, ranc

co_clustering_optimized.fit(trainset)

precision_recall_at_k(co_clustering_optimized)
```

```
RMSE: 1.0654
Precision:  0.394
Recall:  0.566
F_1 score:  0.465
```

Observations and Insights:_ We actually see that the model's performance has gotten worse across every metric.

In [ ]:
```
# Using co_clustering_optimized model to recommend for userId 6958 and song_id 1671
co_clustering_optimized.predict(6958, 1671, r_ui = 2, verbose = True)
```

```
user: 6958       item: 1671       r_ui = 2.00   est = 1.91   {'was_impossible': Fals
e}
```

Out[ ]:
```
Prediction(uid=6958, iid=1671, r_ui=2, est=1.9108882530486497, details={'was_impossib
le': False})
```

In [ ]:
```
# Use Co_clustering based optimized model to recommend for userId 6958 and song_id 323
co_clustering_optimized.predict(6958, 3232, verbose = True)
```

```
user: 6958       item: 3232       r_ui = None   est = 1.24   {'was_impossible': Fals
e}
```

Out[ ]:
```
Prediction(uid=6958, iid=3232, r_ui=None, est=1.2366916027865822, details={'was_impos
sible': False})
```

Observations and Insights:_ We see that, at least for our test prediction, the model gets close with a value of 1.91 listens compared to an actual 2. It also estimates a solid number of predicted listens for a new song (1.24)

Implementing the recommendation algorithm based on optimized CoClustering model

In [ ]:
```
# Getting top 5 recommendations for user_id 6958 using "Co-clustering based optimized"
clustering_recommendations = get_recommendations(df_final, 6958, 5, co_clustering_opti
```

Correcting the play_count and Ranking the above songs

In [ ]:
```
# Ranking songs based on the above recommendations
ranking_songs(clustering_recommendations, play_counts_df)
```

Out[ ]:

	song_id	play_freq	predicted_ratings	corrected_ratings
4	7224	107	3.711503	3.614829
3	5653	108	2.903883	2.807658
0	6860	169	2.691043	2.614120
1	657	151	2.606354	2.524975
2	8483	123	2.582807	2.492640

Observations and Insights:_ We see similar corrected_ratings as compared to predicted ratings as all of the previous models in the dataframe with the top 5 recommended songs for user 6958. (although I used user 47786 for previous predictions)

Content Based Recommendation Systems

Think About It: So far we have only used the play_count of songs to find recommendations but we have other information/features on songs as well. Can we take those song features into account?

In [ ]:

```
df_features = df_final
df_features
```

Out[ ]:

	user_id	song_id	play_count	title	release	artist_name	year
200	6958	447	1	Daisy And Prudence	Distillation	Erin McKeown	2000
202	6958	512	1	The Ballad of Michael Valentine	Sawdust	The Killers	2004
203	6958	549	1	I Stand Corrected (Album)	Vampire Weekend	Vampire Weekend	2007
204	6958	703	1	They Might Follow You	Tiny Vipers	Tiny Vipers	2007
205	6958	719	1	Monkey Man	You Know I'm No Good	Amy Winehouse	2007
...	...	...	...	...	...	...	...
1999734	47786	9139	1	Half Of My Heart	Battle Studies	John Mayer	0
1999736	47786	9186	1	Bitter Sweet Symphony	Bitter Sweet Symphony	The Verve	1997
1999745	47786	9351	2	The Police And The Private	Live It Out	Metric	2005
1999755	47786	9543	1	Just Friends	Back To Black	Amy Winehouse	2006
1999765	47786	9847	1	He Can Only Hold Her	Back To Black	Amy Winehouse	2006

117876 rows × 7 columns

In [ ]:

```
# Concatenate the "title", "release", "artist_name" columns to create a different colu
df_features['text'] = df_features['title'] + ' ' + df_features['release'] + ' ' + df_f

df_features
```

Out[ ]:

	user_id	song_id	play_count	title	release	artist_name	year	text
200	6958	447	1	Daisy And Prudence	Distillation	Erin McKeown	2000	Daisy And Prudence Distillation Erin McKeown
202	6958	512	1	The Ballad of Michael Valentine	Sawdust	The Killers	2004	The Ballad of Michael Valentine Sawdust The Ki...
203	6958	549	1	I Stand Corrected (Album)	Vampire Weekend	Vampire Weekend	2007	I Stand Corrected (Album) Vampire Weekend Vamp...
204	6958	703	1	They Might Follow You	Tiny Vipers	Tiny Vipers	2007	They Might Follow You Tiny Vipers Tiny Vipers
205	6958	719	1	Monkey Man	You Know I'm No Good	Amy Winehouse	2007	Monkey Man You Know I'm No Good Amy Winehouse
...	...	...	...	...	...	...	...	...
1999734	47786	9139	1	Half Of My Heart	Battle Studies	John Mayer	0	Half Of My Heart Battle Studies John Mayer
1999736	47786	9186	1	Bitter Sweet Symphony	Bitter Sweet Symphony	The Verve	1997	Bitter Sweet Symphony Bitter Sweet Symphony Th...
1999745	47786	9351	2	The Police And The Private	Live It Out	Metric	2005	The Police And The Private Live It Out Metric
1999755	47786	9543	1	Just Friends	Back To Black	Amy Winehouse	2006	Just Friends Back To Black Amy Winehouse
1999765	47786	9847	1	He Can Only Hold Her	Back To Black	Amy Winehouse	2006	He Can Only Hold Her Back To Black Amy Winehouse

117876 rows × 8 columns

In [ ]:
```
# Select the columns 'user_id', 'song_id', 'play_count', 'title', 'text' from df_small
df_features = df_features[['user_id', 'song_id', 'play_count', 'title', 'text']]

# Drop the duplicates from the title column
df_features = df_features.drop_duplicates(subset = ['title'])

# Set the title column as the index
df_features = df_features.set_index('title')

# See the first 5 records of the df_small dataset
df_features.head()
```

Out[ ]:

title	user_id	song_id	play_count	text
Daisy And Prudence	6958	447	1	Daisy And Prudence Distillation Erin McKeown
The Ballad of Michael Valentine	6958	512	1	The Ballad of Michael Valentine Sawdust The Ki...
I Stand Corrected (Album)	6958	549	1	I Stand Corrected (Album) Vampire Weekend Vamp...
They Might Follow You	6958	703	1	They Might Follow You Tiny Vipers Tiny Vipers
Monkey Man	6958	719	1	Monkey Man You Know I'm No Good Amy Winehouse

In [ ]:
```
# Create the series of indices from the data
indices = pd.Series(df_features.index)
```

In [ ]:
```
# Importing necessary packages to work with text data
import nltk

# Download punkt library
nltk.download("punkt")

# Download stopwords library
nltk.download("stopwords")

# Download wordnet
nltk.download("wordnet")

# Import regular expression
import re

# Import word_tokenizer
from nltk import word_tokenize

# Import WordNetLemmatizer
from nltk.stem import WordNetLemmatizer

# Import stopwords
from nltk.corpus import stopwords
```

```
# Import CountVectorizer and TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
```

We will create a **function to pre-process the text data:**

In [ ]:
```
# Create a function to tokenize the text
def tokenize(text):

    text = re.sub(r"[^a-zA-Z]"," ", text.lower())

    tokens = word_tokenize(text)

    words = [word for word in tokens if word not in stopwords.words('english')]

    text_lems = [WordNetLemmatizer().lemmatize(lem).strip() for lem in words]

    return text_lems
```

In [ ]:
```
# Create tfidf vectorizer
nltk.download('omw-1.4')
tfidf = TfidfVectorizer(tokenizer = tokenize)

# Fit_transfrom the above vectorizer on the text column and then convert the output in
songs_tfidf = tfidf.fit_transform(df_features['text'].values).toarray()
```

```
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
```

In [ ]:
```
# Compute the cosine similarity for the tfidf above output
similar_songs = cosine_similarity(songs_tfidf, songs_tfidf)
similar_songs
```

Out[ ]:
```
array([[1., 0., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 0., 1.]])
```

Finally, let's create a function to find most similar songs to recommend for a given song.

In [ ]:
```
# Function that takes in song title as input and returns the top 10 recommended songs
def recommendations(title, similar_songs):

    recommendations = []

    # Getting the index of the song that matches the title
    index = indices[indices == title].index[0]

    # Creating a Series with the similarity scores in descending order
    score_series = pd.Series(similar_songs[index]).sort_values(ascending = False)

    # Getting the indexes of the 10 most similar songs
```

```
    top_10 = list(score_series.iloc[1 : 11].index)
    print(top_10)

    # Populating the list with the titles of the best 10 matching songs
    for song in top_10:
      recommendations.append(list(df_features.index)[song])

    return recommendations
```

Recommending 10 songs similar to Learn to Fly

In [ ]:
```
# Make the recommendation for the song with title 'Learn To Fly'
recommendations('Learn To Fly', similar_songs)
```

```
[509, 234, 423, 345, 394, 370, 371, 372, 373, 375]
```

Out[ ]:
```
['Everlong',
 'The Pretender',
 'Nothing Better (Album)',
 'From Left To Right',
 'Lifespan Of A Fly',
 'Under The Gun',
 'I Need A Dollar',
 'Feel The Love',
 'All The Pretty Faces',
 'Bones']
```

Observations and Insights:_ We see a list of the top 10 recommended songs above.

Conclusion and Recommendations

1. Comparison of various techniques and their relative performance based on chosen Metric (Measure of success):

- How do different techniques perform? Which one is performing relatively better? Is there scope to improve the performance further?

We used 6 algorithms for recommendation systems:

- **Rank/Popularity Based**
- **User/User Collaborative Filtering**
- **Item/Item Collaborative Filtering**
- **Matrix Factorization/Model Based**
- **Clustering Based**
- **Content Based**

In terms of the models in which we didn't use RMSE, Precision, Recall & F1 score to measure, we can see that for the **content based recommendation system**, most of our recommendations were of similar songs/artists, which indicates that this model is doing well.

Rank/Popularity Based is a good model that can be used for a user who is new to the platforms, and we don't have any previous data with which to use to find their recommendations with any of the other models.

For the other models:

User/User Collaborative Filtering

We had decent performance before our tuning, but after tuning our RMSE significantly decreased, and precision, recall and F1 score increased significantly, with **recall and F1 score of user/user optimized being the highest of all models**.

Item/Item Collaborative Filtering

We didn't have great performance before tuning, but the precision, recall and F1 scores got significantly better after tuning. However, they didn't compare to the user/user optimized model in terms of performance, with the exception of the **RMSE which was noticeably the lowest we had gotten thus far (although it would soon be surpassed by the matrix factorization model)**.

Matrix Factorization/Model Based

This model had better initial numbers than unoptimized user/user and item/item, and improved even further with tuning, having the **lowest RMSE of all the models and the highest precision**. (Recall and F1 scores were noticeably below User/User optimized).

Clustering Based This model didn't have great initial performance, and its performance, in terms of our metrics, actually worsened after tuning across the board. However, its performance in our sample prediction improved after tuning, but it was still the worst overall model.

It is likely that we could experiment with different hyperparameters to further tune the models for better performance. If I had more time, this is the route I would take.

2. Refined insights:

- What are the most meaningful insights from the data relevant to the problem?

User/User collaborative filtering and Model Based Matrix Factorization were the best performing models. Thinking in the context of the problem, this could make sense. If we look at a genre like rap music, it's so broad that using item/item may not be as helpful, as users may like certain artists and not others. User/user make sense as people have specific tastes within genres and that approach highlights those. This coincides with model based matrix factorization, which may be identifying latent veectors for both users and items which, for a complex subject like music taste, may be perfect for predicting user preference.

For a new user, we may want to consider using popularity based recommendations to start out and then begin using one of our aforementioned models after they've begun interacting with music.

3. Proposal for the final solution design:

- What model do you propose to be adopted? Why is this the best solution to adopt?

I'm not sure I would propose any one of these models singularly. I think a **hybrid recommendation system depending on the need of the business and the consumers would be best**. If we could tailor our recommendations based on our situation, this, I believe, would result in optimal performance.

- **Popularity based would be preferred for new users**
- Models like **item/item** and **clustering** may be more useful for, perhaps not completely new users, but newer users who are still trying to find their tastes and are just looking for any new music that is similar to what they've listenedd to before.
- For more experienced users, it's clear that **user/user and model based collaborative filterng are the most accurate for delivering similar songs as recommendations**, so for this use case those would likely be preferred, although all the models would likely need further tuning to improve performance.
- Content based considers additional data, which may be useful in some cases, so I think it's worth considering as well

Because we might prioritize different parameters at different stages in a user's listening "career", it might be important to retrain the models periodically with updated user data.

- Overall, for such a complex and varying problem such as music preference which depends so heavily on the user, I think it's unwise to pick just one model, and would be better to make a hybrid recommendation system that prioritized different parameters based on factors related to the user's listening history.
- It also depends on what data and recommendations the business prefers. As a business, whenever considering an approach, they will likely have done internal testing and will have data on what approach they think will resonate with the users most, so that must also be taken into consideration. This is why, ultimately, I believe a hybrid recommendation system would give the company the most flexibility, which is extremely important for such a broad topic as music preference.

Executive Summary

We used 6 different models on the dataset we were given to see which model's predictions were the "best" given the training data. Given the Taste Profile Subset as part of the Million Song Subset, we used the techniques described below to see what would output the most accurate song recommendations for a user.

For 4 of the models (User-User based collaborative filtering, Item-Item based collaborative filtering, Matrix Factorization, and Cluster Based), we used RMSE, F1 score, Precision & Recall to rate them.

For these models, **User/User based collaborative filtering** and **Model Based Matrix Factorization** were the best performing models. Thinking in the context of the problem, this could make sense. If we look at a genre like rap music, it's so broad that using item/item may not be as helpful, as users may like certain artists and not others, and some users may have broader tastes (in other words, user A might like genre A and not genre B, and user B might like both). User/user makes sense as people have specific tastes within genres and that approach highlights those. This coincides with model based matrix factorization, which may be identifying latent veectors for both users and items which, for a complex subject like music taste, may be perfect for predicting user preference.

For the other two models in which we did not use those metrics, **Rank/Popularity Based** is a good model that can be used for a user who is new to the platforms, and we don't have any previous data with which to use to find their recommendations with any of the other models. This could work wonders with new user retention, which is integral for growth.

Content Based could also be useful, as it would directly recommend songs/artists based on pre-defined attributes of songs/artists the user had already listened to. However, although this could be useful, I would hesitate to go wth this approach. With so much music available to users on a platform like Spotify, tastes have become much more complex as (observing the data in the EDA) people have significantly increased how much music they listen to. It's likely that with such broad genres like rap, pop and rnb, users will like some rap artists and not others, for example. This approach doesn't seem specific enough for the precision that apps like Spotify and Apple Music require to have effective recommendation systems.

I'm not sure I would propose any one of these models singularly. I think a **hybrid recommendation system depending on the need of the business and the consumers would be best**. If we could tailor our recommendations based on our situation, this, I believe, would result in optimal performance. **Popularity based would be preferred for new users** Users who are new to the platform may not have an idea of what music they like. In general, most people like music that other people like, so this would be a good starting point for new users. As they began to develop their tastes, we will then have enough data to create their own curated recommendations.

For more experienced users, it's clear that **user/user and model based collaborative filterng are the most accurate for delivering similar songs as recommendations**, so for this use case those would likely be preferred, although all the models would likely need further tuning to improve performance.

In terms of improvement, it is paramount that we periodically retrain the models periodically with updated user data. User's music preferences drastically change over time, and the models would become borderline inaccurate if they did not use updated data.

Overall, for such a complex and varying problem such as music preference which depends so heavily on the user, I think it's unwise to pick just one model, and would be better to make a hybrid recommendation system that prioritized different parameters based on factors related to the user's listening history.

It also depends on what data and recommendations the business prefers. As a business, whenever considering an approach, they will likely have done internal testing and will have data on what approach they think will resonate with the users most, so that must also be taken into consideration. This is why, ultimately, I believe a hybrid recommendation system would give the company the most flexibility, which is extremely important for such a broad topic as music preference. If we were to use **weighted hybridization** to combine the results of different techniques, and leverage the data the stakeholders have gathered, we could do internal testing with a hybrid of these two techniques and determine how to move forward.

Problem and Solution Summary

As music becomes more accessible, it becomes even more paramount that companies can key on what will keep people interacting with their app and the music on their app more than ever. Not only companies like Spotify and Apple with Apple music, but artists as well rely on people interacting with these streaming platforms to generate revenue. With so many more songs becoming available, it's become quite tedious to continue to find music similar to one's tastes. That makes it **even more important that users have functionality available to them that streamlines the process of them finding music within the app**.

We're doing this in the context of a business; companies need to be able to figure out what content is needed to increase a user's engagement/time spent on their platform, and leveraging the data they have of what content they're consuming is one way to do that. Our goal is to utilize this pre-existing data to determine how we can use data science more effectively to recommend new music to the consumer to improve their experience.

This will increase user satisfaction and engagement by delivering personalized and accurate music recommendations, enhance the experience of a user by helping them navigating and ever increasing library of music.

To that end, our goal is to create an effective music recommendation system that proposes the top 10 songs for a user based on the likelihood of them listening to those songs. This is based on a number of key factors, including **what kinds of songs they've listened to the most, and their tendencies when it comes to listening to music in general**.

It is important to be able to ensure that their platforms do a good job of both making sure that users can interact with the music they enjoy as easily as possible, but also leveraging algorithms to **be able to recommend new content to users to maintain and even increase engagement, both improving the app's quality and ease of use for the listener. The more time people spend on the app, the more revenue they generate and the more artists benefit as well.**

Recommendations for Implementation

Given the above data we've gathered through our use of the different recommendation systems, it's clear a **weighted hybridization approach for the recommendation system depending on the need of the business and the consumers would be best**. If we could tailor our recommendations based on our situation, this, I believe, would result in optimal performance.

Popularity based would be preferred for new users.

Users who are new to the platform may not have an idea of what music they like. In general, most people like music that other people like, so this would be a good starting point for new users. As they began to develop their tastes, we will then have enough data to create their own curated recommendations.

For more experienced users, it's clear that **user/user and model based collaborative filterng are the most accurate for delivering similar songs as recommendations**, so for this use case those would likely be preferred.

However, it is important to note that it is paramount that we periodically retrain the models with updated user data. User's music preferences drastically change over time, and the recommendations would become borderline inaccurate if they did not use updated data.

Overall, for such a complex and varying problem such as music preference which depends so heavily on the user, I think it's unwise to pick just one model, and would be better to make a hybrid recommendation system that prioritized different parameters based on factors related to the user's listening history.

It also depends on what data and recommendations the business prefers. If we're to use a weighted hybridization approach, it is important to do internal testing before deployinig such a recommendation system to tune the model such that the right factors are being weighted correctly. Using this data would give the company the most flexibility, which is extremely important for such a broad topic as music preference. Combining the results of different techniques, and leveraging the data the stakeholders have gathered with internal testing would allow us to determine how to move forward.

Potential costs/risks of this approach:

- It takes time. Doing internal testing on such a complex system may yield conflicting results; in other words, some people may love the model and it may not work as well for others. This would require plenty of iteration. After doing some research, on average it takes a product recommendation app **400 hours to build**.
- It's expensive to build a model like this. However, investment into a system like this will likely be more than worth the cost considering how overwhelming it can be to listen to music and how users crave guidance in finding new music recommendations.
- Cold start is usually an issue in music recommendation systems, but that's why I recommend using **popularity-based recommendation** for new users to get them started and, for the business, to be able to collect data on that user to develop their recommendations. We could also mitigate this issue by using **explicit user profiling** (making the user explicitly state their music interests while signing up).

- It is expensive to hire engineers and data analysts, among others, to build, deploy and test this system, along with analyzing the results, likely in the **tens of millions**

Benefits of this approach:

- With so much data available to us, both from internal testing and external user data, a company will eventually find what model or combination of models work the best for users. Through some light research, by adding personalized recommendations to an app, it **increases the likelihood of a customer returning to the platform by 300+%** and **reduces the unsubscribe rate by over 65%**. It also makes the user **5 times more likely to recommend the app to others**.
- Looking at an app like Spotify, its average revenue per user is 4.40 dollars per year. With over **574 million users, their revenue is $13.70 billion**.
- For an app focusing on such a fickle form of media as music, user retention, engagement and recommendation are extremely important for the growth of the business. It becomes paramount to give users a reason to not only user their app, but continuously increase engaging with it (and spending money on it) and recommend it to others.
- As engagement increases, you can now use your brand equity to offer a free and paid revenue model, incentivizing users to pay for a premium subscription for a better product, something all music platforms already offer.

Further Analysis/Next Steps

It is clear that further analysis is necessary before undertaking a project such as this. First of all, experiment must be done with different hyperparameters on each model to potentially tune them for better performance. If, after this process, there is one outstanding outlier in terms of performance, that would be our main approach (new users notwithstanding).

If not, we would proceed with building our **weighted hybridization** approach with models outlined previously. The weighting of the results would be decided by the results of internal testing. After iterations of this hybrid model, it could be deployed to users in a beta test to gather more data. We would then continue to iterate until we landed on a model that had the best results, considering all the model analysis metrics we've used above, along with user data such as retention, engagement, and more.

Once we've nailed down the model we wish to use, we would deploy it to all users and continue iterating based on potential feedback.