

# Music Recommender System

---

## A Simple, Interactive System

By Haiman Karim Hamzah, 20267940

*Where words fail, Music speaks*

*Hans Christian Andersen*

# Table of Contents

## Contents

1.0	Introduction .....	3
2.0	Chatbot Architecture .....	3
1.0	Introduction .....	2
2.0	Questions .....	3
2.1	Design Implementation w/ Justification .....	3
2.1.1	NLP .....	4
2.1.2	Agents .....	5
2.1.3	Agent Utilities .....	8
2.1.4	Cosine Similarity .....	9
2.1.5	Collaborative Filtering & Matrix Factorisation .....	9
2.1.6	Authentication .....	10
2.1.7	Main File .....	11
3.0	Technologies, Environment and Challenges .....	12
3.1	Technologies .....	13
3.2	Environments .....	13
3.3	Challenges .....	13
4.0	Experiments and Results .....	14
5.0	Discussion on Research, Responsible and Innovation Guidelines in Music .....	15
6.0	Conclusion .....	16
7.0	References .....	16

## 1.0 Introduction

Digital music streaming lets users browse millions of tracks from numerous artists, genres, and eras. Given the plethora of musical content, finding new songs and singers that suit one's

tastes may be difficult. Music recommendation algorithms are changing how consumers find and engage with music.

A variety of algorithms are used to recommend songs, artists, and playlists to users based on their listening history, tastes, and behaviour. These complicated systems combine collaborative filtering, content-based filtering, and matrix factorization to provide tailored and relevant music selections for each user.

A music recommendation system optimizes the user's experience by helping them find new music that matches their tastes and explore a wide range of musical material.

The dataset, approach, and assessment criteria utilized to build a music recommendation system are examined in this paper. This study uses user listening histories, song metadata, and user-song interactions to train and evaluate recommendation systems.

More specifically, we will demonstrate how Spotify Web API, spotipy, numpy, pandas, fuzzy matching, and other Python modules were integrated into this project.

Additionally, this research examines the challenges and considerations of building a music recommendation system. The material addresses data sparsity, cold-start, and scalability issues. It also proposes solutions and development areas.

In summary, this study analyses music recommendation system construction and evaluation. Advanced algorithms and user data improve the user experience, ease music exploration, and make personalized suggestions. This work contributes to music recommendation system research and development.

## **2.0 Questions**

The questions as to which I prioritise solving are

- How well the agents classify user preference on songs with contribution to genre
- How well collaborative filtering (matrix factorisation) coincides with the user-item recommendations
- How well content analysis (Tf-idf) focuses on features to recommend music to users
- How autonomous models compare (in my case fuzzy matching)
- Is the user data relevant?

Aside from that, I increased the amount of questions to answer getting similar artists based on the user likings (content analysis), how does user interactivity play a role in music recommendation systems and how well the agents explore to search according to the user's preference.

## **2.1 Design Implementation w/ Justification**

In terms of design from a general overview, I did an agent based of the environment Spotify Web utilizing auth and tokens to access a spotify account to gain access to the developer API's. In terms of agents, I build an agent that does a search

through a specific playlist in coordination with the user's preference of song, aside from that I also build an agent utilizing cosine similarity to analyse the music features and return an artist similar to the user's taste. Finally, I also created an agent that scans through a dataset, pre-processes and cleans the dataset using pandas and recommend a "user – user" similarity taste – Matrix factorization

### 2.1.1 NLP

In the tokenisation file, I go through a function that provides two different features for my user-based music system. In the 'process' function, it is used to detect the user's name using Named-entity-recognition (pos tags) by getting the tags and referring the name to a specific tag. I tokenize the user input,

```
def process(user_input):  
    #Tokenizes the sentence  
    post_tokens = nltk.word_tokenize(user_input)
```

which divides the string into a list of substrings – to determine which substring is considered as the user's name.

```
#does pos tag on it for named entity recognition  
post_word = nltk.pos_tag(post_tokens)
```

As nltk pos\_tag can be inconsistent in detecting the name of the user due to the ambiguity of tagging the strings – incorrect tagging, I utilized an if statement that would return if the user decided to input just their name or a sentence consisting their name, e.g. Q: *What is your name?* A: *Haiman / My Name is Haiman*

```
if len(post_word) == 1:  
    for i in post_word:  
        return i[0]  
else:  
    #detects the name in the sentences  
    detected_names_nltk = [name.title() for name, pos in post_word if pos == 'NNP' or pos == 'JJ' or pos == 'ADJ']  
    return detected_names_nltk
```

This 'if statement' would iterate through the length of the 'post\_word' variable (consisting of the word and the tags) and returning if the length is 1, as this would assume the user inputted their name as a sole entity. If the length doesn't equate to 1, it would iterate through the sentence to see which string is of either Proper Noun Singular (NNP), Adjective 'big' (JJ), or Adjective (ADJ). The 'title()' method was also used to capitalize each word in the list of substrings.

### Justification

For the function handling named entity recognition (NER), I built it in this format to handle multiple different inputs aside from just typing in your name. e.g. Haiman. I wanted to build it in a more flexible manner due to uncertainty on how different users introduce themselves.

This function is solely used for the purpose of recognising names and saving them to a variable to be used whenever the user requests for it e.g. “What is my name?”

As for the if statement, I built it that way because if I were to return a name based on their pos\_tag – it would assign the name to ‘NN’ (which isn’t correct) hence why it would return if the length of the string equates to 1, assuming the string is a name. The else statement iterates through a bunch of strings to detect which word is considered a name (NNP or ADJ). e.g. My name is Haiman.

### 2.1.2 Agents

For the agents that I’ve implemented, I’ve created three agents, two using Spotify’s Environment and one through data analysing and autonomous calculations on a dataset.

```
fields = 'items(track(id,name,artists,album(id,name)))'  
playlist_data = spotify.playlist_tracks(playlist_id, fields=fields)  
#print(json.dumps(playlist_data, sort_keys=True, indent=4))
```

My first agent involves using spotipy API and scanning through a specified playlist to match a track given by the user.

```
scaler = MinMaxScaler()  
music_features = music_df[['Danceability', 'Energy', 'Key',  
                           'Loudness', 'Mode', 'Speechiness', 'Acousticness',  
                           'Instrumentalness', 'Liveness', 'Valence', 'Tempo']].values  
music_features_scaled = scaler.fit_transform(music_features)
```

Based on a scaled outlook on the dataframe, it is then processed through cosine similarity to get the similarity scores of each track in the playlist based on their music features. Then, the similarity scores are then sorted to give a descending frame to fit the pursue of our next step – multiplying by popularity.

```
playlist_recommendations = pd.concat([content_based_recommendations, input_song_df], ignore_index=True)  
playlist_recommendations = playlist_recommendations.sort_values(by='Popularity', ascending=False)  
playlist_recommendations = playlist_recommendations[playlist_recommendations['Track Name'] != song_name]
```

The similarity scores are then multiplied by a weighted popularity score that was calculated using spotipy’s parameters, to recommend a track that not only is trending, but fits the criteria of what the user wants. E.g. The Weeknd & Arianna Grande.

The second agent involves a simple search algorithm through the music communities open-source listening history – to get related artists (in terms of similarities) based on the user’s preference. This can be done as spotify themselves collected data based off of their communities – with consent, to be analyse and improve products as well as marketing, B2B services that improves the user experience as a whole.

```

sim_artist = []
for i in artist_all:
    artists_name = i['name']
    genres = i['genres']
    artist_id = i['id']
    popularity = i['popularity']

    similar_artist_data = {
        "Artist": artists_name,
        "Genres": genres,
        "Artist ID": artist_id,
        "Popularity": popularity
    }

    sim_artist.append(similar_artist_data)

```

The agent, if prompted to, would return a series up to 5 (can be edited) artists that are similar to an artist that the user has given. One of the explanations in terms of the algorithm also involve the communities listening pattern

```

for track in self.tracks:
    self.play_counts[track] = self.play_counts.get(track, 0) + 1

```

That allows the algorithm to track what content can be analysed and given to the requester based on user interactions.

For my final agent and arguably the most advanced, involves a dataset separate from the Spotify Web API, but instead uses the [millionsongdataset](#) that governs a community project – to analyse the data through pre-processing, data cleaning, grouping and categorizing, matrix factorizing and finally fuzzy matching.

```

#read million song subset based on user interaction
initial_song = pd.read_csv('10000.txt', sep='\t', header=None)
initial_song.columns = ['user_id', 'song_id', 'listen_count']

#song metadata
metadata_song = pd.read_csv('song_data.csv')
metadata_song.drop_duplicates(['song_id'], inplace=True)

#merge together
initial_metadata = pd.merge(initial_song, metadata_song, on="song_id", how="left")

#convert to csv file
initial_metadata.to_csv('song_dataset.csv', index=False)

# Load and preprocess data
data = pd.read_csv('song_dataset.csv')
print(data.head())

```

As the Million Song Dataset provided consists of hundreds and thousands of users and their listening count to a specific song/artist, I initially turned it into a CSV file in order to output it as a dataframe using pandas.

From there, the preprocessing of the dataset such as removing missing values, scaling the dataset, dropping duplicates and obtaining improved datasets in the end to perform matrix factorization on.

```
recommended_tracks = data_update3.pivot(index="title", columns="user_id", values="listen_count").fillna(0)
recommended_tracks_binary = recommended_tracks.apply(np.sign)
matrix_recommended_tracks_binary = csr_matrix(recommended_tracks_binary.values)
```

As recommended\_tracks consists of our final cleaned dataframe, we use numpy's sign to apply a binary representation of the datasets rather than strings of labels – From there, a matrix can be formed on the rows and columns that is needed – in this case, title, users and amount of times song was listened too.

```
# Calculate the cosine similarity between the user's songs and all songs
user_songs_indices = recommended_tracks.index.isin(user_songs)
user_songs_matrix = matrix_recommended_tracks_binary[user_songs_indices]
cosine_sim = 1 - pairwise_distances(user_songs_matrix, matrix_recommended_tracks_binary, metric="cosine")

# Get the top similar songs based on cosine similarity
similar_songs_indices = cosine_sim.argsort()[0][:-num_recommendations:][::-1]
similar_songs = recommended_tracks.index[similar_songs_indices].tolist()
```

From there, we analyse through one user the songs they listened too and apply cosine similarity between the user's song and all songs to first get a similarity score. From there, it is then sorted based on their ranks of similarity.

After gaining this data, fuzzy matching using python's fuzzywuzzy is performed to find the closest matching songs to recommend to the user's listening history.

```
match_ratio = max(fuzz.ratio(song.lower(), user_song.lower()) for user_song in user_songs)
if match_ratio >= 15: # Adjust the threshold as needed
    matched_songs.append(song)
```

The threshold is set to 15 because the average listens to a song is 269 per song, however in the dataset the max amount of listens to a song is 8227, aside from also only taking the top 30% of the dataset, as to avoid efficiency and redundant noise when searching through the dataset. Hence, the threshold would have to be low based due to the scaled dataset – in improving performance.

## Justification

For the first agent, the reason why I implemented it that way from a general perspective is because playlists can be large, and to scroll through listening to each music would be tedious. Hence, creating a platform where users can find the most similar songs based on what they like in the playlist not only reduces time, but increases a chain effect where you go can scan a song that u got recommended to, in another playlist

From a technical perspective, I wanted it to be as simple as possible and the best (arguable) method that was referenced online was the analyse the metadata of the tracks and return a number of tracks based on their velocity, tempo, danceability etc..

Not only that, python gives us a way to access this similarity in terms of value – and numbers never lie.

For the second agent, the abundance of artists is growing – especially independent artists – by about 27% every year, this goes to show that the numerous amount of search required can be shortened through a search algorithm that can scan the artists based on their genre and other users listening history, to pinpoint more artists for the requester to follow and listen too.

With the access to spotify's consensual data collection, performing a search on the grouped artists can return substantial recommendations.

As for my final agent, I wanted a hands-on application with a dataset to implement an autonomous intelligent agent so I could further improve the state of recommendation with comparing which agent performs the best when it comes to analysing the user data and giving a recommendation. In this case, scanning with fuzzy logic through a dataset to get a truth output between 0 and 1.

Based on multiple articles online, it outlines the importance of utilizing fuzzy matching to make recommendations for users based on specific tracks or artists.

$$\text{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \text{lev}_{a,b}(i-1,j) + 1 \\ \text{lev}_{a,b}(i,j-1) + 1 \\ \text{lev}_{a,b}(i-1,j-1) + 1_{(a \neq b)} \end{cases} & \text{otherwise.} \end{cases}$$

Furthermore, it uses a Levenshtein distance – python, where it checks the difference between the user's preference of songs and another.

### 2.1.3 Agent Utilities

In terms of utilities based on the spotify environment, the first two agents utilised it the most to do parameter-tuning and return instances of the desired columns and their values into a dataframe

```
tracks_list.append({
    'Track Name': track_name,
    'Artists': artist_names,
    'Album Name': album_name,
    'Album ID': album_identifier,
    'Track ID': track_identifier,
    'Popularity': track_popularity,
    'Release Date': album_release_date,
    'Duration (ms)': audio_features.get('duration_ms', None) if audio_features else None,
    'Explicit': track_details.get('explicit', None),
    'Spotify URL': track_details.get('external_urls', {}).get('spotify', None),
    'Danceability': audio_features.get('danceability', None) if audio_features else None,
    'Energy': audio_features.get('energy', None) if audio_features else None,
    'Key': audio_features.get('key', None) if audio_features else None,
    'Loudness': audio_features.get('loudness', None) if audio_features else None,
    'Mode': audio_features.get('mode', None) if audio_features else None,
    'Speechiness': audio_features.get('speechiness', None) if audio_features else None,
    'Acousticness': audio_features.get('acousticness', None) if audio_features else None,
    'Instrumentalness': audio_features.get('instrumentalness', None) if audio_features else None,
    'Liveness': audio_features.get('liveness', None) if audio_features else None,
    'Valence': audio_features.get('valence', None) if audio_features else None,
    'Tempo': audio_features.get('tempo', None) if audio_features else None,
})
```

Through the audio\_features method, the desired metadata for a track can be extracted and analysed.



## Justification

The reason as to why I did this is to create a content-based analysis. To recommend a user a certain song based on the characteristics that are like by the user. Hence through this dataframe, a vectorizer can be performed to compare the similarities and match the user with their designated songs.

Another method that I hope to apply if ever the chance given would be a more interactive method where it would require deep learning and approval from the user – In a sense that when given a track, the user could like or dislike to further train. This is because music can be very scarce as it follows what you're into at the moment, so just because I like a fast tempo, doesn't mean I hate a slow tempo.

### 2.1.4 Cosine Similarity

In simple terms, cosine similarity measures the numerical distance in terms of features between one data and another. It is a metric of cosine as it as it goes through a binary non-zero vector in a multi-dimensional space. – Also known as calculating the angle of cosine between them.

```
user_songs_indices = recommended_tracks.index.isin(user_songs)
user_songs_matrix = matrix_recommended_tracks_binary[user_songs_indices]
cosine_sim = 1 - pairwise_distances(user_songs_matrix, matrix_recommended_tracks_binary, metric="cosine")
```

## Justification

It has been of significant importance to utilize cosine similarity especially in search algorithms and recommendation systems because it provides a very robust numerical (otherwise accurate) way to represent similarities between datasets. Furthermore, no matter how large the data are, they could still potentially be similar if the angle vector between them are small – This avoids the cause of false negative if implemented with TfIDF.

### 2.1.5 Collaborative Filtering & Matrix Factorisation

Based on our collaborative filtering, I implemented a matrix factorisation of getting an item based on what other user's used to interact with. User A and User B has similar taste, hence recommend User A about User B's likes and vice versa.

```
recommended_tracks = data_update3.pivot(index="title", columns="user_id", values="listen_count").fillna(0)
recommended_tracks_binary = recommended_tracks.apply(np.sign)
matrix_recommended_tracks_binary = csr_matrix(recommended_tracks_binary.values)

def recommend_songs(user_id, num_recommendations=10):
    # Get the user's listened songs
    user_songs = data_update3[data_update3['user_id'] == user_id]['title'].tolist()

    # Calculate the cosine similarity between the user's songs and all songs
    user_songs_indices = recommended_tracks.index.isin(user_songs)
    user_songs_matrix = matrix_recommended_tracks_binary[user_songs_indices]
    cosine_sim = 1 - pairwise_distances(user_songs_matrix, matrix_recommended_tracks_binary, metric="cosine")
```

Through the data frame, it helps adjust what users in the column like by transforming it with least squared (Alternate least squared in our case) into a matrix – Where the rows are the users and columns are the items (labels).

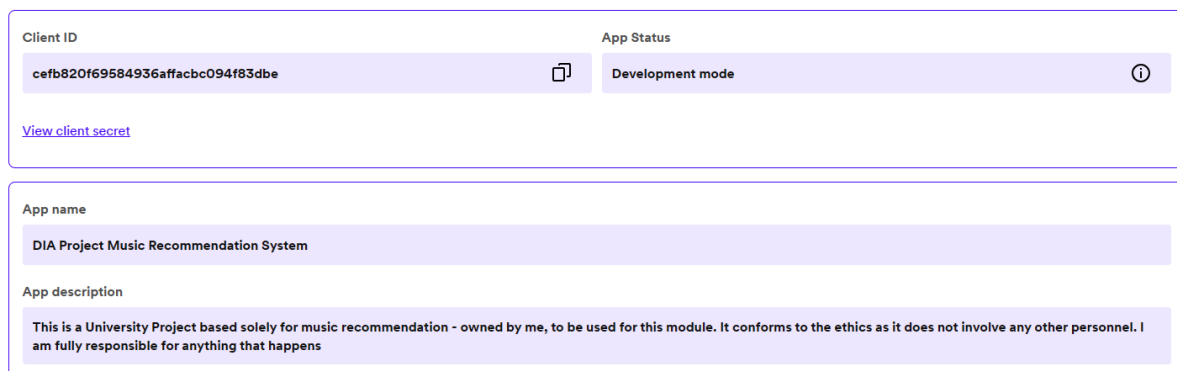
## Justification

Based on a multitude of references – even Google – It was more of a comparison to showcase the big difference between recommending using user's own history and recommending using similar taste among users. This gives us an argument to debate the accuracy of each model in recommending music.

As you can see with the content-based filtering, it is strict to it's own statistics not counting external factors that may play a huge role in recommendations (e.g. emotions and time of day).

### 2.1.6 Authentication

This access token was used to connect to Spotify Developer's environment using a 'CLIENT\_ID' and a 'CLIENT\_SECRET'.



The screenshot displays the Spotify Developer Dashboard for a specific application. It is divided into two main sections. The top section, titled 'Client ID' and 'App Status', shows the Client ID as 'cefb820f69584936affacbc094f83dbe' with a copy icon, and the App Status as 'Development mode' with an information icon. Below this is a link to 'View client secret'. The bottom section, titled 'App name' and 'App description', shows the App name as 'DIA Project Music Recommendation System' and the App description as 'This is a University Project based solely for music recommendation - owned by me, to be used for this module. It conforms to the ethics as it does not involve any other personnel. I am fully responsible for anything that happens'.

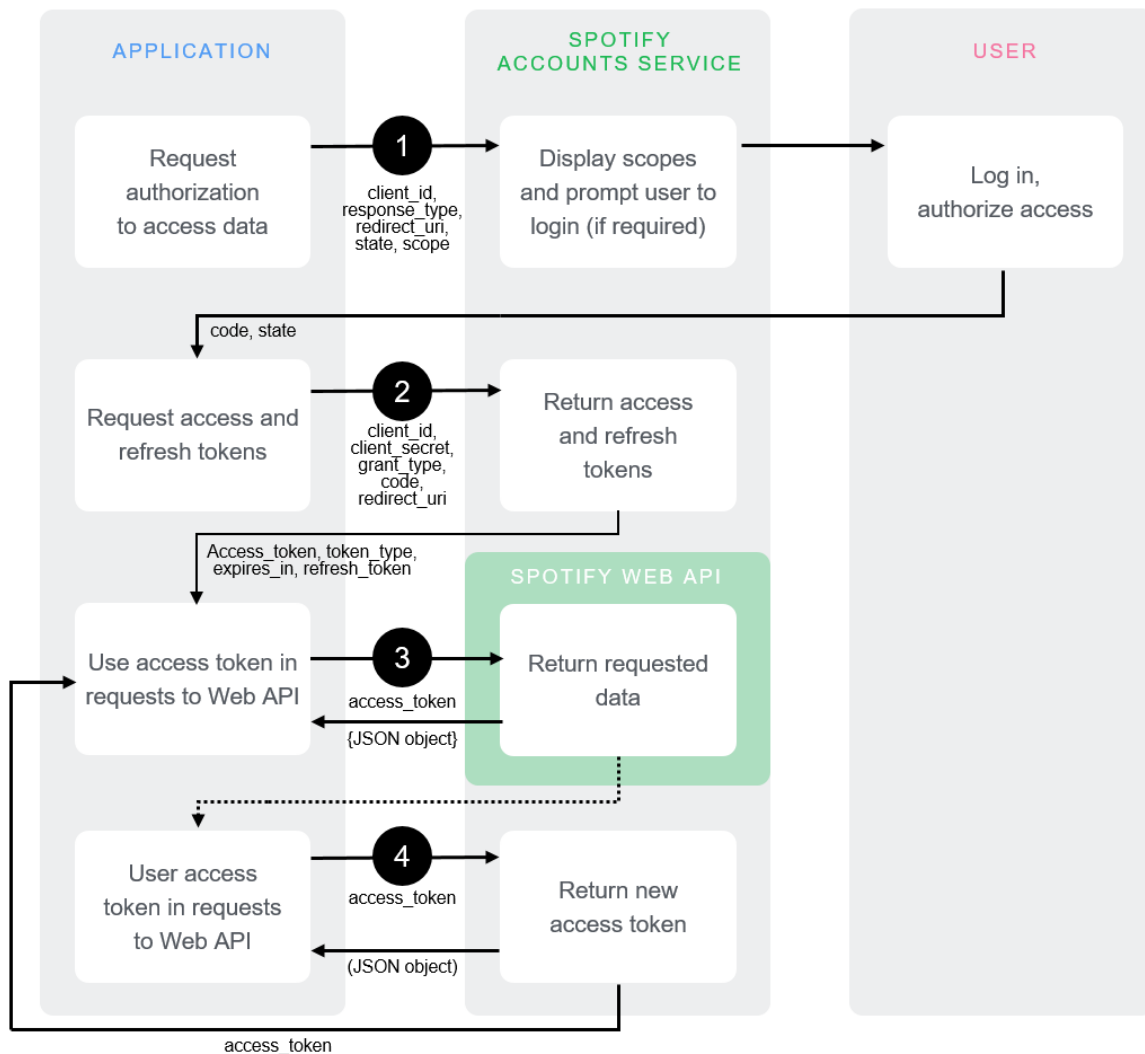
Client ID	App Status
cefb820f69584936affacbc094f83dbe	Development mode

[View client secret](#)

App name
DIA Project Music Recommendation System

App description
This is a University Project based solely for music recommendation - owned by me, to be used for this module. It conforms to the ethics as it does not involve any other personnel. I am fully responsible for anything that happens

This showcases the start of creating an app to gain authentication with Spotify's environment. I conform to the ethics and responsibilities as everything seen here is owned by me. The account used to run the spotify environment is owned by me as a test for this module.



This showcases the method in gaining access to the spotify web api for developers. As you can see, it allows if the access token can be obtained successfully in order to utilize the spotify API's

### Justification

In order to use the spotify web api environment, this was needed to give access to the developer.

#### 2.1.7 Main File

First and foremost, In order to run this program, a few python libraries need to be installed beforehand.

List of python libraries to be installed through pip

- Spotipy
- Scikit-learn
- Numpy
- Pandas
- fuzzywuzzy
- Python -levenshtein

- Scipy
- nltk

List of files that need to be run onced

```
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('universal_tagset')
```

In the main file, this is where it calls all the respective agents and their environments in a simple interactive system through terminal

```
Access token obtained successfully.
Hi there, What is your name?
You: hai

-----
Hi there, Nice to meet you hail!, Welcome to the Music Recommender System
-----
Choose among the available options
-----
0 - Get the tracks recommendation based on a song name in a playlist based on content
1 - Get the artists most similar to your liking based on spotify community
2 - Scan through a dataset to process Matrix Factorisation based on user likings
DISCLAIMER: 2 TAKES QUITE A WHILE TO LOAD
-----
Type 'exit' to quit
```

## Justification

Originally, if you can see in the codebase, I did multiple different variations and ideas based on how I would tackle this music recommendation system. First was by using Flask, but due to my inexperience, I opted for a simple terminal interaction that does what is needed when given an input.

It is a simple method that not only decreases load on the computer, but it also showcases the users what I is printed straight – albeit lacking in user experience.

## 3.0 Technologies, Environment and Challenges

There are several technologies that were used (some discarded) in the process of creating this music recommendation system. Through these technologies, I will explain the environment on which these agents act. – in return changing the environment around them.

From these multiple implementations, several (understatement) challenges arose which caused multiple changes to be made throughout the coursework. I will explain it all and I will justify the usage of why I changed. – Including what difference would it have made if I stuck to the original plan.

### 3.1 Technologies

The technologies I used in implementing this music recommendation system consists of mainly:

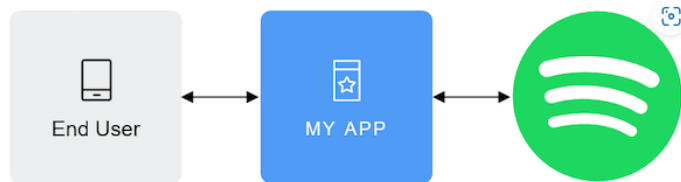
- Python, Spotipy, Scipy, Numpy, Pandas, Scikit-learn, Python -levenshtein, nltk,

Technologies that I originally wanted to implement

- Flask, tensorflow, clusters and machine learning

### 3.2 Environments

As for the environment, I utilized creating an authentication in Spotify Web API to connect my end user through an app created in spotify for developers and a server which is authorised via OAuth 2.0



Where:

- *End User* corresponds to the Spotify user. The *End User* grants access to the protected resources (e.g. playlists, personal information, etc.)
- *My App* is the client that requests access to the protected resources (e.g. a mobile or web app).
- *Server* which hosts the protected resources and provides authentication and authorization via OAuth 2.0.

In terms of environment flow, I utilized the authorization code flow so it would constantly access the token every time the program is called.

Through this virtual connection, my agents are able to utilise the “Scopes” – access to information, after authorization is granted.

### 3.3 Challenges

On to the challenges, the biggest challenge I faced was definitely time constraints and tutorial hell. For an example, the usage of machine learning in an extremely large dataset could be implemented, but without thorough data pre-processing, feature engineering and even model selection – would cause the system to perform worse than intended, sacrificing on efficiency.

As this is quite a new topic with several articles explaining different methods in implementing a music recommendation system, it goes to show that there is no definite method, it’s just what you want to gain by answering your questions and in this case. My implementation gave a sufficient amount of data to be utilized as an answer – which I will explain in the Experiments and Results section

## 4.0 Experiments and Results

There were a few experiments that I wanted to solve by solving the said questions. One of the first experiments was the showcasing of how the users fare with collaborative filtering – based on their own likes

	user_id	song_id	listen_count	title	artist_name	year	total_listen_count	total_artist_listen_count
8	b08344d6c3b5ccb3212f76538f3d9e43d57dc9e	S0FRQTD12A81C233C0	1	Sehr kosmisch	Harmonia	0	8277	8277
88	e006b1a48f466bf59feefed32bec6494495a4436	S0ALWY12A81C206F1	2	Undo	Björk	2001	7032	8519
89	e006b1a48f466bf59feefed32bec6494495a4436	S0AXG3H12A81C3F8A1	2	Dog Days Are Over (Radio Edit)	Florence + The Machine	0	6949	18112
91	e006b1a48f466bf59feefed32bec6494495a4436	S0B0NKR12A58A7A7E0	2	You're The One	Dwight Yoakam	1990	6729	6412
97	e006b1a48f466bf59feefed32bec6494495a4436	S0FRQTD12A81C233C0	3	Sehr kosmisch	Harmonia	0	8277	8277

From this diagram you can see that the similarity and popularity in terms of similar taste resonate with the two users.

So it goes on to the question, how **WELL** was the collaborative filtering method compared to the content based filtering method?

To answer that question, based on a reference data, take a look at the methods on which they stem from.

```
# Get the user's listened songs
user_songs = data_update3[data_update3['user_id'] == user_id]['title'].tolist()
```

Collaborative filtering just relies on the user's past interaction and listening history to deduce a result. Using this theory, you can derive that – it is in itself, a learning algorithm as well. As it learns from the past to be brought into the future results.

	title	total_listen_count
0	#!*@ You Tonight [Featuring R. Kelly] (Explici...	78
1	#40	338
2	& Down	373
3	' Cello Song	103
4	'97 Bonnie & Clyde	93

Such as the total listen count of each song and categorised among users to do the matrix factorization for user and item(label)

As for content filtering, in order for the algorithm to run efficiently, you would need multiple features and metadata about the track to deduce a suggestion

```
content_based_recommendations = music_df.loc[similar_song_indices][['Track Name', 'Artists', 'Album Name', 'Release Date', 'Popularity']]
```

So through this content based analysis, it can showcase you quite an accurate representation as you can see in the table

Track Name	Artists	Album Name	Release Date	Popularity
One Last Time	Ariana Grande	My Everything (Deluxe)	2014-08-22	82
Congratulations	Post Malone, Quavo	Stoney (Deluxe)	2016-12-09	81
Better Now	Post Malone	beerbongs & bentleys	2018-04-27	80
Mo Bamba	Sheck Wes	MUDBOY	2018-10-05	77
In The Night	The Weeknd	Beauty Behind The Madness	2015-08-28	74

However, it does come with cons in terms of external factors outside of music features. For an example, if I like a high tempo song, with sad lyrics – the chances of me being recommended a hip-hop rap song would be quite high, when all I listen to can be, say

metal. So in order for this method to work, the more metadata of audio features, the better.

Onto my final question, is the user data based on their history worthwhile? With the basic research that I've done, through the implementation of the fuzzy matching you can see that there is definitely room for improvement in terms of matching a user's listening history as it should also conform to consent in personal data usage

```
Recommended songs for user b80344d063b5ccb3212f76538f3d9e43d87dca9e:  
Dog Days Are Over (Radio Edit)  
Revelry  
You're The One  
You: exit  
Thank you for using out Music Recommender System! Bye!
```

## 5.0 Discussion on Research, Responsible and Innovation Guidelines in Music

### Anticipate

In anticipating the impact of the music recommendation systems, I have ensured that data privacy is an absolute must by allowing the user a choice to provide us with such details if it happens to arise. The transparency of how the system works is easily laid out for developers to decipher the algorithms used. Based on several citations, there is definitely several ways such as using reinforcement learning in cooperation with neural networks to implement a Music Recommender System.

### Reflect

The number one thought process when creating this system was user experience, how the user can feel satisfied after using our app. Would there be emotional bias regarding song recommendations? Would data be leaked among these users that could potentially harm them? the purpose of this all was to ensure the user would have an open-minded perspective on music data. There are however multiple different instances where this system may be overlook such as inclusivity of the dataset (differing genre likings), as this recommendation system is catered to everyone simplistically, there may also be a concern of not being able to cater to anyone due to users finding the system to be very ambiguous.

### Engage

Unanimously, an RRI-compliant based Recommendation UI must have a consistent plan explaining the details what the system does. In my system, I ensured that all the features are accessible by the users and can conform to transparency and fairness within the users. From an ethics perspective, the system is to respect and guide the user through to ensure maximum ease of access by utilizing only information that is open source or that is mine.

## Act

As for now, the ideal plan moving onwards is to improve the user empowerment within the app. Creating a more interactive and sophisticated recommender system where more than two algorithms are used. Enhancing the data collection would directly translate to improving the data privacy and security as well. Aside from that, data collection as feedback would be a priority as well to ensure fairness and understanding on how users use our recommendation system

## 6.0 Conclusion

In Conclusion, I am quite satisfied with my music recommendation however I do wish I could implement more such as a deep learning experience to fully gain feedback from the user. With this I would need multiple personals to test my system, but if that were to happen – I would definitely gain more understanding and touch a sensitive subject such as emotion, more carefully.

My agents in question, could also classify the music type by maybe even categorising the specific details based on personality, which was an idea I came up with but never came to fruition due to lack of time.

If given more time, I would definitely try to understand a user's perspective when wanting to gain more tracks and artists to follow.

**3763 Words excl. References**

## 7.0 References

- 1) Dey, V. (2021). *Collaborative Filtering Vs Content-Based Filtering for Recommender Systems*. [online] Analytics India Magazine. Available at: <https://analyticsindiamag.com/collaborative-filtering-vs-content-based-filtering-for-recommender-systems/>.
- 2) Glauber, R. and Loula, A. (n.d.). *Collaborative Filtering vs. Content-Based Filtering: differences and similarities*. [online] Available at: <https://arxiv.org/pdf/1912.08932v1> [Accessed 13 May 2024].
- 3) Jeremy (2016). *A Closer Look at the Million Song Dataset*. [online] Modeling Music. Available at: <https://medium.com/modeling-music/the-intersection-between-music-and-computation-or-commonly-referred-to-as-computational-music-49d3311a95e2>.
- 4) cgallay.github.io. (n.d.). *Million Song Project*. [online] Available at: <https://cgallay.github.io/Ada/>.
- 5) www.researchgate.net. (n.d.). *(PDF) The Million Song Dataset*. [online] Available at:



[https://www.researchgate.net/publication/220723656\\_The\\_Million\\_Song\\_Datas](https://www.researchgate.net/publication/220723656_The_Million_Song_Datas)  
et.

6) GeeksforGeeks. (2020). *Cosine Similarity*. [online] Available at:

<https://www.geeksforgeeks.org/cosine-similarity/>.

7) Google Developers. (n.d.). *Matrix Factorization / Recommendation Systems*.

[online] Available at: <https://developers.google.com/machine-learning/recommendation/collaborative/matrix>.

8) Adiyansjah, Gunawan, A.A.S. and Suhartono, D. (2019). Music Recommender System Based on Genre using Convolutional Recurrent Neural Networks. *Procedia Computer Science*, 157, pp.99–109.

doi:<https://doi.org/10.1016/j.procs.2019.08.146>.

9) www.youtube.com. (n.d.). 7 - LEARN SPOTIPY - A Lightweight Python Spotify Library. [online] Available at:

[https://www.youtube.com/watch?v=HxYByMv\\_DO4&list=PLqgOPibB\\_QnzzcaOFYmY2cQjs35y0is9N&index=8](https://www.youtube.com/watch?v=HxYByMv_DO4&list=PLqgOPibB_QnzzcaOFYmY2cQjs35y0is9N&index=8) [Accessed 13 May 2024].

10) spotipy.readthedocs.io. (n.d.). *Welcome to Spotipy! — spotipy 2.0*

*documentation*. [online] Available at: <https://spotipy.readthedocs.io/en/2.22.1/#>.

11) developer.spotify.com. (n.d.). *Scopes / Spotify for Developers*. [online]

Available at: <https://developer.spotify.com/documentation/web-api/concepts/scopes>.