AudioAlchemy

Musical transitions in a DJ set are optimal when the song transitioned from and the song to have technical similarities. Our goal in this project is to use a heuristic search algorithm called Beam Search and personal rankings of transitions to build an optimal playlist with an optimal set of transitions. We will be connecting to the Spotify API to get pre-built playlists and reorganizing them using our own Beam Search algorithm with customized transition score calculations.

Authors: Dexter Renick, John Curnaj, & Manav Sharma

Collecting song information

Spotipy is a package that allows developers to interface with the Spotify API through Python code. It is useful to manage the playlists that we are looking to reorder.

In [1]:
```
!pip install spotipy
```

```
Requirement already satisfied: spotipy in /opt/anaconda3/lib/python3.8/site
-packages (2.23.0)
Requirement already satisfied: six>=1.15.0 in /opt/anaconda3/lib/python3.8/
site-packages (from spotipy) (1.15.0)
Requirement already satisfied: requests>=2.25.0 in /opt/anaconda3/lib/pytho
n3.8/site-packages (from spotipy) (2.28.1)
Requirement already satisfied: redis>=3.5.3 in /opt/anaconda3/lib/python3.
8/site-packages (from spotipy) (4.5.4)
Requirement already satisfied: urllib3>=1.26.0 in /opt/anaconda3/lib/python
3.8/site-packages (from spotipy) (1.26.4)
Requirement already satisfied: async-timeout>=4.0.2 in /opt/anaconda3/lib/p
ython3.8/site-packages (from redis>=3.5.3->spotipy) (4.0.2)
Requirement already satisfied: certifi>=2017.4.17 in /opt/anaconda3/lib/pyt
hon3.8/site-packages (from requests>=2.25.0->spotipy) (2022.12.7)
Requirement already satisfied: charset-normalizer<3,>=2 in /opt/anaconda3/l
ib/python3.8/site-packages (from requests>=2.25.0->spotipy) (2.1.0)
Requirement already satisfied: idna<4,>=2.5 in /opt/anaconda3/lib/python3.
8/site-packages (from requests>=2.25.0->spotipy) (2.10)
```

In [2]:
```
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
```

In [3]:
```
cid = "3519692942004325a2c7160c90717ca5"
secret = "9243be1df96e48bb829c4b07254bd82c"
```

In [4]:
```
# This cell is used to clear any previous user data from the cache of the nc
import os
cache_path = ".cache"
```

```
if os.path.exists(cache_path):
    os.remove(cache_path)
```

In [5]:
```
# This cell sets the Spotify API object sp that we can interface with later
client_credentials_manager = SpotifyClientCredentials(client_id=cid, client_
sp = spotipy.Spotify(client_credentials_manager = client_credentials_manager
```

In [6]:
```
# The playlist stores its unique ID in the URL so the following code will ex
playlist_link = "https://open.spotify.com/playlist/0whpp60V3f8y5G1e6HV59N?si
uri = playlist_link.split("/")[-1].split("?")[0]
track_uris = [x["track"]["uri"] for x in sp.playlist_tracks(uri)["items"]]
```

Spotify's API has some amazing metrics to be able to understand the characteristics of a song before you ever even hear it.

The following descriptions come directly from Spotify's documentation:
https://developer.spotify.com/documentation/web-api/reference/get-audio-features

Danceabilitity

"Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable."

Energy

"Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy."

Key

"The key the track is in. Integers map to pitches using standard Pitch Class notation. E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on."

Liveness

"Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live."

Loudness

"The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typically range between -60 and 0 db."

Mode

"Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0."

Speechiness

"Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks."

Tempo

"The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration."

Valence

"A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry)."

Acoustic

"A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic."

In [7]:
```
print(sp.audio_features(track_uris)[0])
```

```
{'danceability': 0.653, 'energy': 0.826, 'key': 6, 'loudness': -2.93, 'mod
e': 0, 'speechiness': 0.354, 'acousticness': 0.0153, 'instrumentalness': 0.
00146, 'liveness': 0.131, 'valence': 0.416, 'tempo': 108.52, 'type': 'audio
_features', 'id': '2I9foKseoFQh07p6sD2voE', 'uri': 'spotify:track:2I9foKseo
FQh07p6sD2voE', 'track_href': 'https://api.spotify.com/v1/tracks/2I9foKseoF
Qh07p6sD2voE', 'analysis_url': 'https://api.spotify.com/v1/audio-analysis/2
I9foKseoFQh07p6sD2voE', 'duration_ms': 256227, 'time_signature': 5}
```

We will now extract those metrics from the playlist that we input.

In [8]:
```python
import pandas as pd

# Initialize empty lists for columns
song_names = []
danceability = []
energy = []
key = []
loudness = []
mode = []
speechiness = []
acousticness = []
instrumentalness = []
liveness = []
valence = []
tempo = []

# Loop through the tracks and append the data to the respective lists
for track in sp.playlist_tracks(playlist_link)["items"]:
    track_name = track["track"]["name"]
    song_names.append(track_name)

    audio_features = sp.audio_features(track["track"]["uri"])[0]
    danceability.append(audio_features["danceability"])
    energy.append(audio_features["energy"])
    key.append(audio_features["key"])
    loudness.append(audio_features["loudness"])
    mode.append(audio_features["mode"])
    speechiness.append(audio_features["speechiness"])
    acousticness.append(audio_features["acousticness"])
    instrumentalness.append(audio_features["instrumentalness"])
    liveness.append(audio_features["liveness"])
    valence.append(audio_features["valence"])
    tempo.append(audio_features["tempo"])

# Create a DataFrame
data = {
    "Song Name": song_names,
    "Danceability": danceability,
    "Energy": energy,
    "Key": key,
    "Loudness": loudness,
    "Mode": mode,
    "Speechiness": speechiness,
    "Acousticness": acousticness,
```

```
    "Instrumentalness": instrumentalness,
    "Liveness": liveness,
    "Valence": valence,
    "Tempo": tempo
}

df = pd.DataFrame(data)
df
```

Out[8]:

	Song Name	Danceability	Energy	Key	Loudness	Mode	Speechiness	Acousticness	Instr
0	Many Men (Wish Death)	0.653	0.826	6	-2.930	0	0.3540	0.01530	
1	Can't Tell Me Nothing	0.596	0.620	5	-6.133	0	0.0390	0.01220	
2	Poppin' Them Thangs	0.769	0.818	6	-1.385	0	0.1920	0.26300	
3	Ambitionz Az A Ridah	0.801	0.910	2	-6.566	1	0.2250	0.00239	
4	How We Do	0.862	0.648	4	-7.401	0	0.2510	0.04550	
5	Locked Up	0.833	0.568	7	-4.312	1	0.0847	0.02620	
6	Nuthin' But A "G" Thang	0.669	0.821	11	-4.370	0	0.2900	0.00182	
7	Hit 'Em Up - Single Version	0.916	0.844	7	-3.967	1	0.2360	0.03940	
8	Got It On Me	0.688	0.647	2	-7.258	1	0.1900	0.00815	
9	The Enemy (feat. Fat Joe)	0.606	0.649	11	-8.372	1	0.3920	0.11500	

Score

We need a weighting system that identifies the importance of each feature.

In [9]:

```
def get_relative_key(key, mode):
    if mode == 1:  # Major key
        return (key + 9) % 12  # Relative minor key
```

```
    else:  # Minor key
        return (key + 3) % 12  # Relative major key

def key_compatibility(key1, mode1, key2, mode2):
    if key1 == key2:
        return True
    if mode1 != mode2 and (key1 == get_relative_key(key2, mode2) or key2 ==
        return True
    return False

def genre_similarity(genres1, genres2):
    if not genres1 or not genres2:
        return 0
    shared_genres = len(set(genres1).intersection(genres2))
    # print("shared_genres: ", shared_genres)
    # print("shared_genres / max(len(genres1), len(genres2)): ", shared_genr

    # max of 5 shared genres
    return min(shared_genres, 5)

# The closer the score to 0, the better the transition
def evaluate_transition(song1, song2):
    score = 0
    #total genres for a given song
    max_genres = 10

    # Weights for different attributes
    weights = {
        'danceability': 7,
        'energy': 5,
        'loudness': 1,
        'tempo': 100,
        'valence': 5,
        'genre': 4
    }

    # Check key compatibility
    if not key_compatibility(song1['key'], song1['mode'], song2['key'], song
        score += 6  # Return a large score if keys are not compatible

    # Calculate the differences in attributes
    diff = {}
    for attribute in weights:
        if attribute == 'genre':
            # if they have 5 in common, it is going to add 0
            diff[attribute] = 5 - genre_similarity(song1[attribute], song2[a
        else:
            diff[attribute] = abs(song1[attribute] - song2[attribute])

    # Normalize loudness differences
    diff['loudness'] /= 60  # Assume max difference is 60 dB

    # Calculate tempo difference considering double/half time mixing
    tempo_diff = min(
        abs(song1['tempo'] - song2['tempo']),
        abs(song1['tempo'] * 2 - song2['tempo']),
```

```
        abs(song1['tempo'] / 2 - song2['tempo'])
    )
    diff['tempo'] = tempo_diff / 200  # Normalize tempo difference

    # Calculate the transition score
    for attribute in weights:
        # print("attribute: ", attribute, "   score: ", weights[attribute] *
        score += weights[attribute] * diff[attribute]

    return score
```

Transition Score Testing

In [10]:
```
def print_song_data(song):
    print("Song Name:", song["track_name"])
    print("Danceability:", song["danceability"])
    print("Energy:", song["energy"])
    print("Key:", song["key"])
    print("Loudness:", song["loudness"])
    print("Mode:", song["mode"])
    print("Valence:", song["valence"])
    print("Tempo:", song["tempo"])
    print("Genre:", ', '.join(song["genre"]))
    print()

def get_related_artist_genres(artist_id):
    related_artists = sp.artist_related_artists(artist_id)
    genres = []
    for artist in related_artists['artists']:
        genres.extend(artist['genres'])
    return list(set(genres))

def get_song_data(track_id):
    track = sp.track(track_id)
    audio_features = sp.audio_features(track_id)[0]
    artist_id = track['artists'][0]['id']
    genres = get_related_artist_genres(artist_id)

    song_data = audio_features.copy()
    song_data['genre'] = genres
    song_data['track_name'] = track['name']
    return song_data
```

Same Song score

In [11]:
```
song1_id = "0hURIUSiPFIv7dzlejdf3N"
song2_id = "0hURIUSiPFIv7dzlejdf3N"

song1 = get_song_data(song1_id)
song2 = get_song_data(song2_id)

print("Song 1 Info:")
print_song_data(song1)
```

```
print("Song 2 Info:")
print_song_data(song2)

transition_score = evaluate_transition(song1, song2)
print(f"Transition score between the two songs: {transition_score:.2f}")
```

```
Song 1 Info:
Song Name: The Sweetest Taboo - Quiroga Remix
Danceability: 0.802
Energy: 0.898
Key: 6
Loudness: -9.163
Mode: 1
Valence: 0.628
Tempo: 128.021
Genre: classic indo pop, indonesian city pop, trio batak, lagu maluku, indo
nesian worship, batak

Song 2 Info:
Song Name: The Sweetest Taboo - Quiroga Remix
Danceability: 0.802
Energy: 0.898
Key: 6
Loudness: -9.163
Mode: 1
Valence: 0.628
Tempo: 128.021
Genre: classic indo pop, indonesian city pop, trio batak, lagu maluku, indo
nesian worship, batak

Transition score between the two songs: 0.00
```

Good Song score

In [12]:

```
song1_id = "0yLdNVWF3Srea0uzk55zFn"
song2_id = "3RfNQMIeuL2QC9l4VxOMoj"

song1 = get_song_data(song1_id)
song2 = get_song_data(song2_id)

print("Song 1 Info:")
print_song_data(song1)

print("Song 2 Info:")
print_song_data(song2)

transition_score = evaluate_transition(song1, song2)
print(f"Transition score between the two songs: {transition_score:.2f}")
```

```
Song 1 Info:
Song Name: Flowers
Danceability: 0.707
Energy: 0.681
Key: 0
Loudness: -4.325
Mode: 1
Valence: 0.646
Tempo: 117.999
Genre: alt z, boy band, art pop, girl group, pop, post-teen pop, talent sho
w, electropop, transpop, candy pop, swedish electropop, swedish synthpop, u
k pop, swedish pop, metropopolis, dance pop

Song 2 Info:
Song Name: Dream
Danceability: 0.684
Energy: 0.675
Key: 0
Loudness: -5.886
Mode: 1
Valence: 0.846
Tempo: 117.97
Genre: alt z, boy band, art pop, girl group, pop, post-teen pop, talent sho
w, electropop, transpop, candy pop, swedish electropop, swedish synthpop, u
k pop, swedish pop, metropopolis, dance pop

Transition score between the two songs: 1.23
```

Medium Song score

In [13]:

```
song1_id = "1j6kDJttn6wbVyMaM42Nxm"
song2_id = "2I9foKseoFQh07p6sD2voE"

song1 = get_song_data(song1_id)
song2 = get_song_data(song2_id)

print("Song 1 Info:")
print_song_data(song1)

print("Song 2 Info:")
print_song_data(song2)

transition_score = evaluate_transition(song1, song2)
print(f"Transition score between the two songs: {transition_score:.2f}")
```

```
Song 1 Info:
Song Name: Lord Pretty Flacko Jodye 2 (LPFJ2)
Danceability: 0.485
Energy: 0.72
Key: 6
Loudness: -5.991
Mode: 1
Valence: 0.0471
Tempo: 207.982
Genre: drill, miami hip hop, chicago drill, chicago rap, rap, atl hip hop,
conscious hip hop, memphis hip hop, new orleans rap, dark trap, chicago bo
p, dirty south rap, crunk, psychedelic hip hop, alternative hip hop, hip ho
p, underground hip hop, pop rap, tennessee hip hop, trap, boom bap, gangste
r rap, experimental hip hop, virginia hip hop, detroit hip hop, southern hi
p hop, escape room, indiana hip hop

Song 2 Info:
Song Name: Many Men (Wish Death)
Danceability: 0.653
Energy: 0.826
Key: 6
Loudness: -2.93
Mode: 0
Valence: 0.416
Tempo: 108.52
Genre: bronx hip hop, old school atlanta hip hop, g funk, east coast hip ho
p, atl hip hop, rap, singeli, hardcore hip hop, dirty south rap, crunk, dan
ce pop, urban contemporary, nyc rap, hip pop, hip hop, pop rap, trap, queen
s hip hop, gangster rap, detroit hip hop, west coast rap, r&b, southern hip
hop

Transition score between the two songs: 5.87
```

Bad Song score

In [14]:

```python
song1_id = "1j6kDJttn6wbVyMaM42Nxm"
song2_id = "0hURIUSiPFIv7dzlejdf3N"

song1 = get_song_data(song1_id)
song2 = get_song_data(song2_id)

print("Song 1 Info:")
print_song_data(song1)

print("Song 2 Info:")
print_song_data(song2)

transition_score = evaluate_transition(song1, song2)
print(f"Transition score between the two songs: {transition_score:.2f}")
```

```
Song 1 Info:
Song Name: Lord Pretty Flacko Jodye 2 (LPFJ2)
Danceability: 0.485
Energy: 0.72
Key: 6
Loudness: -5.991
Mode: 1
Valence: 0.0471
Tempo: 207.982
Genre: drill, miami hip hop, chicago drill, chicago rap, rap, atl hip hop,
conscious hip hop, memphis hip hop, new orleans rap, dark trap, chicago bo
p, dirty south rap, crunk, psychedelic hip hop, alternative hip hop, hip ho
p, underground hip hop, pop rap, tennessee hip hop, trap, boom bap, gangste
r rap, experimental hip hop, virginia hip hop, detroit hip hop, southern hi
p hop, escape room, indiana hip hop

Song 2 Info:
Song Name: The Sweetest Taboo - Quiroga Remix
Danceability: 0.802
Energy: 0.898
Key: 6
Loudness: -9.163
Mode: 1
Valence: 0.628
Tempo: 128.021
Genre: classic indo pop, indonesian city pop, trio batak, lagu maluku, indo
nesian worship, batak

Transition score between the two songs: 38.08
```

Beam Search Description

Beam Search is a heuristic algorithm that explores a graph by expanding only on the most promising of nodes. To summarize prior, a heuristic algorithm is an algorithm that prioritizes a near optimal solution based on a variety of specialized techniques. As a result, heuristic algorithms like Beam Search are greedy and prioritize a locally optimal solution trading accuracy for speed.

Beam Search is a breadth search algorithm that generates all the current successors. Here is the general outline of our algorithm in terms of musical transitions: First, we will begin with a a singular node, containing a transition score of 0, current songs (which is empty to begin with), and a list of available songs.

Second, for each available song, we will evaluate a transition score from the last song of the current playlist to the available song selected. The way we evaluated transition scores for each song is done by scoring musical likeness between songs. We will then remove the available song from the available songs list and add it to the current songs list. We will add this transition score to the total score and store this iteration of current songs, transition score, available songs as a new node in our heap of nodes to be explored. Once all available songs have been visited we will then remove from the heap

of newly created nodes with some of the highest transition scores. We will now consider only a constant set of nodes (which are of the lowest scores). We will repeat these steps until all available songs have been considered and a full playlist has been created. We will select the remaining node with the highest transition score, as this node contains the playlist with the heuristically determined optimal solution.

In [15]:
```
import heapq

def beam_search(songs, beam_width=3):
    # Initialize the search space with the initial state
    search_space = [([], songs, 0)]

    while search_space:
        # Keep track of the next states with their corresponding costs
        next_states = []

        for state in search_space:
            playlist, remaining_songs, cost = state

            # If there are no remaining songs, we have a complete playlist
            if not remaining_songs:
                return playlist, cost

            # Generate possible next states by adding one of the remaining s
            for song in remaining_songs:
                if song not in playlist:
                    new_remaining_songs = remaining_songs.copy()
                    new_remaining_songs.remove(song)
                    new_playlist = playlist + [song]

                    if len(new_playlist) > 1:
                        last_song = get_song_data(playlist[-1])
                        current_song = get_song_data(song)
                        transition_cost = evaluate_transition(last_song, cur
                    else:
                        transition_cost = 0

                    new_cost = cost + transition_cost
                    next_states.append((new_playlist, new_remaining_songs, r

        # Keep only the best 'beam_width' states for the next iteration
        # You can experiment with beam width lengths.
        search_space = heapq.nsmallest(beam_width, next_states, key=lambda x

    return None, float('inf')
```

Now, let's see it in action

In [16]:
```
sample_playlist_link = "https://open.spotify.com/playlist/0whpp60V3f8y5G1e6H
sample_uri = sample_playlist_link.split("/")[-1].split("?")[0]
sample_playlist = [x["track"]["uri"] for x in sp.playlist_tracks(sample_uri)

optimal_playlist, cost = beam_search(sample_playlist, beam_width=3)
```

```
print("Optimal playlist order:")
for i, song in enumerate(optimal_playlist):
    print(f"{i + 1}. {get_song_data(song)['track_name']}")

print(f"\nTransition cost: {cost:.2f}")
```

```
Optimal playlist order:
1. Poppin' Them Thangs
2. Ambitionz Az A Ridah
3. Got It On Me
4. Nuthin' But A "G" Thang
5. Hit 'Em Up - Single Version
6. How We Do
7. Locked Up
8. The Enemy (feat. Fat Joe)
9. Many Men (Wish Death)
10. Can't Tell Me Nothing

Transition cost: 93.57
```

Conclusion

Analysis

During the course of this project, we aimed to find an optimal sorting mechanism for reordering the Spotify playlist tracks. To achieve this goal, we developed an evaluation score function, which we tweaked iteratively using a combination of tests, intuition, and analysis. The following is an analysis of the process we went through to arrive at our final evaluation score function:

Initial function

We started with a basic evaluation score function that considered a few track features. However, we quickly realized that this initial function did not produce satisfactory results in terms of reordering the playlist tracks.

Feature analysis

To improve our evaluation score function, we first analyzed the available track features in the dataset. We focused on those features that could have a significant impact on the listening experience, such as tempo, danceability, energy, and valence.

Feature selection

Based on our analysis, we selected a set of track features that we believed would contribute to a better listening experience. We then experimented with different combinations of these features to understand their relative importance in the evaluation score function.

Tweaking weights

Once we had chosen the relevant features, we assigned weights to them in our evaluation score function. We tweaked these weights iteratively, running tests on different playlists to see how they affected the resulting track order. We adjusted the weights based on the test results and our intuition about the importance of each feature in the listening experience.

Testing

To ensure the effectiveness of our evaluation score function, we ran multiple tests using different playlists with diverse characteristics. We compared the original track order to the reordered tracklist generated by our function, looking for improvements in the listening experience.

Intuition and feedback

Throughout the process, we also relied on our intuition and gathered feedback from users to understand how the reordered playlists aligned with their preferences. This information helped us further refine the weights and the overall evaluation score function.

Final function

After numerous iterations, tests, and refinements, we arrived at an optimal evaluation score function that produced satisfactory results in terms of reordering the playlist tracks. This function was able to effectively balance the selected features, creating a more enjoyable listening experience for the users.

In conclusion, our iterative approach, involving feature analysis, feature selection, tweaking of weights, testing, and the incorporation of intuition and user feedback, helped us develop an optimal evaluation score function for reordering Spotify playlists. This function is capable of enhancing the listening experience and can be further refined or expanded upon as needed.

Next Steps

Potential next steps could be:

Implement additional sorting methods to allow users to experiment with different listening experiences. Some examples include sorting by tempo, danceability, valence (mood), or track length.

Analyze the playlists of multiple users to understand the common characteristics and trends in playlist creation, which could lead to insights into user behavior and

preferences using a deep learning algorithm and or reinforcement learning.

Create a user-friendly interface (e.g., a web or desktop application) that allows users to interact with the project easily, reorder their playlists, and visualize the results.

Consider using machine learning techniques to create personalized playlist recommendations based on the user's listening history and preferences.

Experiment with creating playlists that cater to specific moods, activities, or situations (e.g., workout playlists, relaxation playlists, or party playlists) by analyzing the audio features of the tracks and generating playlists that meet the desired criteria.

In [ ]: