1. Idea and motivation

Spotify smart shuffle is hot ass so I wanted to apply my decision tree classification skills and recreate the smart shuffle feature.

The idea is simple: I want to create a playlist that takes in recent songs that the user has listened to so that the songs generated are based off of the user's more recent preferences rather than songs which the user has liked in the past -- if the user has listening to songs that were released during the pandemic then the "Smarter Playlist" can generate songs that are similar to that of the "pandemic" songs.

1.1 How am I going to do this

I'm going to create a custom decision tree classification model that learns from the last 30 songs that the user encountered. These aren't unique songs that the user likes, but songs they didn't like as well.

I want to first analyze a couple of songs that the users listens to -- ex: if the user doesn't like a song, they will be more likely to skip the song within the first couple of seconds (the decision whether the user likes the song or not will be called **likability**).

The idea is to also leverage the Spotify Playback SDK in order to track when the user clicks on the skip button. A threshold of 5 seconds will be initialized to determine whether the user likes or dislikes a song. An idea that I have is that I can use the 5 second threshold as sort of a measure of how much a listener enjoys the song -- the closer the skip action is to 5 seconds, the more the user dislikes the song. Furthermore, the number generated also determines how the features are related to a more positive 1 value. For example, if the song is marked as "liked" then we can assess that whatever dominating feature is for that song has a strong association with the user's song preference.

Data Y ratio calculation:

If song is not liked and (time Listened in seconds > 2 seconds):

    Data Y = 1 - (time listened in seconds / 5)

Else:

    Data Y = 1

With this information we can populate our table of songs for the user with data Y being our ratio calculations and data X being our feature values.

1.1.1 Decision tree features

Obviously decision tree classification makes predictions based on different features (X data) and coming up with a decision (Y data). The features of a song can be queried using Spotify API endpoint.

Specifically: GET https://api.spotify.com/v1/audio-features/{id}

From this endpoint, the classification features would be **danceability, energy, tempo, valence, acousticness, instrumentalness, mode, time_signature.** I chose these features specifically since I believed that these features closely represent a user's listening preferences.

Once we have enough user-listened data I can create the table that will train the decision tree model. I'll have a button called "create smart playlist" and that will curate a playlist of songs that the user can refer to.

1.1.2 Querying new music

So the way the decision tree works is that it needs to make predictions on new data. The original idea was to generate a totally new set of music, but the scope for that kinda sucks. So instead I decided to use a list of genres and song years listened to. This will be limited to the last 5 genres and years of the songs the user liked. For example:

Genres = ['Pop', 'Rap', 'RnB', 'Kpop', 'Classical']

Years = [2000, 2001, 2003, 2004, 2020]

Furthermore: for the years list, spotify API only takes an interval of years, so I'm thinking that I implement a queue system and when the API is called to make recommendations, I can get the range of the oldest and youngest years.

Making requests based on genre might take a little longer since we have to make multiple separate calls for each genre listened to. So the recommendation logic would look something like this:

```python
import requests

def search_songs_by_genre_and_year(genre, year_range, access_token, limit=10):
    # Format the query with both genre and year range
    url = f"https://api.spotify.com/v1/search?q=genre:{genre}%20year:{year_range}&type=track&limit=
    headers = {"Authorization": f"Bearer {access_token}"}
    response = requests.get(url, headers=headers)
    return response.json()

def recommend_songs_by_genres_and_years(genres, year_range, access_token, limit=10):
    all_recommendations = []

    for genre in genres:
        # Get songs for each genre and year range
        songs = search_songs_by_genre_and_year(genre, year_range, access_token, limit)
        all_recommendations.extend(songs['tracks']['items'])

    return all_recommendations
```

Spotify get random songs list API call:

https://api.spotify.com/v1/search?q=genre:{genre}%20year:{year_range}&type=track&limit={limit}

2 Decision tree design

This is how the application will flow: user logs on using their Spotify credentials → if the user is new to the site, their user information will be stored in Postgres database → Every 10 songs a user listens to, the tree will be built asynchronously so that when the user wanted to generate a playlist, it will be done faster WITHOUT having to build the tree again → user clicks on “Build my playlist” and the random songs list will be queried and every song will be run against the model. → user can choose to keep the playlist or toss the playlist (extended feature).

XGBoost

3 Database Design

The database will use a relational database. This is because there are specific db features that will make ML training much easier -- joins, LRU cache implementation, storing song metadata.

USERS Table:

```sql
CREATE TABLE users (
  id UUID PRIMARY KEY,
  spotify_id TEXT UNIQUE NOT NULL,
  display_name TEXT,
  email TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- Table to store user ID and information. This information is mainly static and used to query song information from users.

SESSIONS table:

```sql
CREATE TABLE sessions (
  user_id UUID REFERENCES users(id),
  access_token TEXT NOT NULL,
  refresh_token TEXT NOT NULL,
  expires_at TIMESTAMP NOT NULL,
  PRIMARY KEY (user_id)
);
```

- This table stores user session keys -- this table will most likely be updated every time the user needs a new refresh_token.

SONGS Table

```sql
CREATE TABLE songs (
  id TEXT PRIMARY KEY,  -- Spotify track ID
  name TEXT,
  artist TEXT,
  album TEXT,
  genre TEXT,  -- If available or inferred
  release_year INT,
  audio_features JSONB  -- Store danceability, energy, etc.
);
```

- The songs table holds the metadata for every unique song that has been listened to by any user. If User A and User B both listen to the same song, that song is added only once to the songs table

LISTENING_EVENTS table

```sql
CREATE TABLE listening_events (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID REFERENCES users(id),
  track_id TEXT REFERENCES songs(id),
  played_at TIMESTAMP,
  duration_seconds INT,
  skipped BOOLEAN,
  liked BOOLEAN,  -- Whether the user liked the track
  context TEXT,   -- Playlist, album, etc (optional)
  UNIQUE(user_id, track_id, played_at)
);
```

- This table holds the data for each user action. Kind of a big thing tbh idk how to scale this one. Maybe caching.

The idea is to have each user store up to 50 songs for their DT classifier creation. To do this, we can use ROW_NUMBER() function:

```sql
WITH ranked AS (
  SELECT id,
         ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY listened_at DESC) AS row_num
  FROM listening_data
  WHERE user_id = $1
)
DELETE FROM listening_data
WHERE id IN (
  SELECT id FROM ranked WHERE row_num > 50
);
```

5.1 Additional memory related notes

I want to cache certain data such as the previous tree if a user clicks on the “build my playlist” button multiple times. (This avoids the unnecessary re-building and re-querying of a tree). To do this **Redis** seemed like a sensible choice to employ caching. The Redis cache would also store the necessary user data.

4 Hosting Database

I’ll use Supabase

https://supabase.com/

5 Frameworks and Stack Brainstorming

Frontend: React.js (or typescript depending on how I feel tbh)

- Next.js for server sider frontend loading

Backend: Python/Pandas to create DT classifier

- Django to create Postgres HTTP requests
- Postgres for DB