Playlist-Based Recommendation System for Spotify:

A Recommendation System utilizing the cosine similarity metric

Eric Burton Martin
Hermela Darebo
Colorado State University

1. INTRODUCTION:

In this research study, we aim to leverage the data mining techniques taught in the CS481a5 course at Colorado State University to gain insights from the Spotify Million Playlist Dataset [1]. This comprehensive dataset comprises 1,000,000 user-generated playlists from the popular music streaming platform, Spotify, and includes playlist and track titles. Despite the wealth of information provided by these playlists, the dataset lacks the requisite details needed to create an effective music recommendation system. To address this limitation, we employ a two-step approach that involves mapping and reducing the dataset, followed by data enrichment through use of the Spotify API. This process allows us to generate a more comprehensive dataset containing valuable information that can be harnessed for making personalized music recommendations. By employing the cosine similarity metric, we are able to identify the ten most similar songs to the average vector representation of each playlist, thereby providing a basis for more accurate and tailored song recommendations. In the subsequent sections of this paper, we will discuss the methods used to preprocess and enrich the dataset, detail the application of data mining techniques, and present our findings on the effectiveness of the proposed recommendation system. Furthermore, we will analyze the limitations of the current approach and explore potential avenues for future research and improvements in the field of music recommendation system.

2. KEYWORDS:

Data Mining, Spotify Million Playlist, Cosine Similarity, Music Recommendation, Music Preference, Trend Analysis

3. RELATED WORK:

There is a plethora of related works that I use the Spotify Million Playlist Dataset as a basis for a recommendation system. Our system is nothing novel but rather than finding recommendations based on songs, it averages the entirety of the playlists genres and audio features to make recommendations based on that.

4. OBJECTIVES:

The primary goal of this research study is to develop an effective music recommendation system that utilizes the Spotify Million Playlist Dataset, by employing data mining techniques and enriching the dataset with relevant information. To achieve this goal, we have set the following specific objectives:

Data preprocessing: Restructure the raw dataset to facilitate further analysis and application of data mining techniques. This is done through use of MapReduce.

Data enrichment: Enhance the dataset by incorporating additional relevant information such as genre and audio features in order to create a more comprehensive and robust data source for generating music recommendations.

Feature extraction: Identify and extract salient features from the enriched dataset that can be used to represent songs and playlists in a multi-dimensional space, which enables the calculation of similarity between items.

Similarity analysis: Apply the cosine similarity metric to measure the similarity between songs and the average vector representation of each playlist, in order to identify the ten most similar songs for recommendation purposes.

Evaluation: Assess the performance and effectiveness of the proposed music recommendation system by comparing its recommendations with existing popular and collaborative filtering-based systems, as well as by gathering qualitative feedback from users.

By accomplishing these objectives, we aim to create a music recommendation system that can provide users with personalized and accurate song suggestions, thereby enhancing the overall user experience on the Spotify platform and promoting the discovery of new music. Additionally, our research findings may offer valuable insights for future improvements in the field of music recommendation systems and contribute to the growing body of knowledge in this domain.

5. DATASET:

The Million Playlist Dataset consists of 32 gigabytes of data consisting of 1,000 slice files each containing 1000 playlists. Each slice file is a JSON dictionary with two fields: *info* and *playlists*.

The *info* field is a dictionary that contains general information about the particular slice:

- *slice* - the range of slices that in in this particular file - such as 0-999
- *version* - the current version of the MPD (which should be v1)
- *description* - a description of the MPD
- *license* - licensing info for the MPD
- *generated_on* - a timestamp indicating when the slice was generated.

The *playlists* field is an array that typically contains 1,000 playlists. Each playlist is a dictionary that contains the following fields:

- *pid* - integer - playlist id - the MPD ID of this playlist. This is an integer between 0 and 999,999.
- *name* - string - the name of the playlist
- *description* - optional string - if present, the description given to the playlist.
- *modified_at* - seconds - timestamp (in seconds since the epoch) when this playlist was last updated.
  num_artists - the total number of unique artists for the tracks in the playlist.

- *num_albums* - the number of unique albums for the tracks in the playlist
- *num_tracks* - the number of tracks in the playlist
- *num_followers* - the number of followers this playlist had at the time the MPD was created. (Note that the follower count does not including the playlist creator)
- *num_edits* - the number of separate editing sessions. Tracks added in a two hour window are considered to be added in a single editing session.
- *duration_ms* - the total duration of all the tracks in the playlist (in milliseconds)
- *collaborative* - boolean - if true, the playlist is a collaborative playlist. Multiple users may contribute tracks to a collaborative playlist.
- *tracks* - an array of information about each track in the playlist. Each element in the array is a dictionary with the following fields:
  - *track_name* - the name of the track
  - *track_uri* - the Spotify URI of the track
  - *album_name* - the name of the track's album
  - *album_uri* - the Spotify URI of the album
  - *artist_name* - the name of the track's primary artist
  - *artist_uri* - the Spotify URI of track's primary artist
  - *duration_ms* - the duration of the track in milliseconds
  - *pos* - the position of the track in the playlist (zero-based)

Here's an example of a typical playlist entry:

```
{
        "name": "musical",
        "collaborative": "false",
        "pid": 5,
        "modified_at": 1493424000,
        "num_albums": 7,
        "num_tracks": 12,
        "num_followers": 1,
        "num_edits": 2,
        "duration_ms": 2657366,
        "num_artists": 6,
        "tracks": [
           {
              "pos": 0,
              "artist_name": "Degiheugi",
              "track_uri": "spotify:track:7vqa3sDmtEaVJ2gcvxtRID",
              "artist_uri": "spotify:artist:3V2paBXEoZIAhfZRJmo2jL",
              "track_name": "Finalement",
              "album_uri": "spotify:album:2KrRMJ9z7Xjoz1Az4O6UML",
              "duration_ms": 166264,
              "album_name": "Dancing Chords and Fireflies"
           },
           {
              "pos": 1,
              "artist_name": "Degiheugi",
              "track_uri": "spotify:track:23EOmJivOZ88WJPUbIPjh6",
              "artist_uri": "spotify:artist:3V2paBXEoZIAhfZRJmo2jL",
              "track_name": "Betty",
              "album_uri": "spotify:album:3lUSlvjUoHNA8IkNTqURqd",
              "duration_ms": 235534,
              "album_name": "Endless Smile"
           }]
      }
```

6. METHODOLOGY:

This section presents a detailed description of the steps and methodologies employed to achieve the project goals.

6.1 Data Preprocessing:

The original dataset was divided into large yet manageable slices for efficient processing. Due to the limited resources available, we opted to use a subset of the dataset, consisting of 9 files (9,000 playlists), instead of the entire collection of 1,000 files.

6.2 Apache Hadoop and Spark:

To process the dataset efficiently, we leveraged a distributed computing environment powered by Apache Hadoop and Spark. Our cluster consisted of one master namenode, a secondary namenode, and five worker nodes, which allowed for parallel processing and fault tolerance.

In order to run the MapReduce using Python, the following steps were performed:

6.2.1 Step 1:

Locate the path to the hadoop-streaming.jar file which is often found in the $HADOOP_HOME/share/hadoop/tools/lib/ directory.

Hadoop streaming is a utility that comes with the Hadoop distribution. The utility allows you to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer. [3]

In the context of Hadoop Streaming, the print statements are used to emit the key-value pairs. In this case, you don't need to write the results to files manually as Hadoop Streaming will handle the file management for you. When using Hadoop Streaming, the mapper and reducer scripts read from stdin (standard input) and write to stdout (standard output). Hadoop Streaming will take care of the data communication between mappers, reducers, and HDFS.

In the mapper and reducer scripts seen in sections 6.3 and 6.4, the print statements are used to emit key-value pairs separated by a tab character. Hadoop Streaming will read these key-value pairs from stdout and handle the shuffling and sorting of the intermediate data before passing it to the reducers.

Here's an example of how the data flows in a Hadoop Streaming job:

1. Hadoop Streaming reads input data from HDFS and passes it to the mapper script through stdin.
2. The mapper script processes the input data and prints key-value pairs to stdout.
3. Hadoop Streaming reads the key-value pairs from the mapper's stdout, shuffles and sorts the data, and then passes it to the reducer script through stdin.
4. The reducer script processes the shuffled and sorted data, printing the final key-value pairs to stdout.
5. Hadoop Streaming reads the key-value pairs from the reducer's stdout and writes the output data to HDFS.

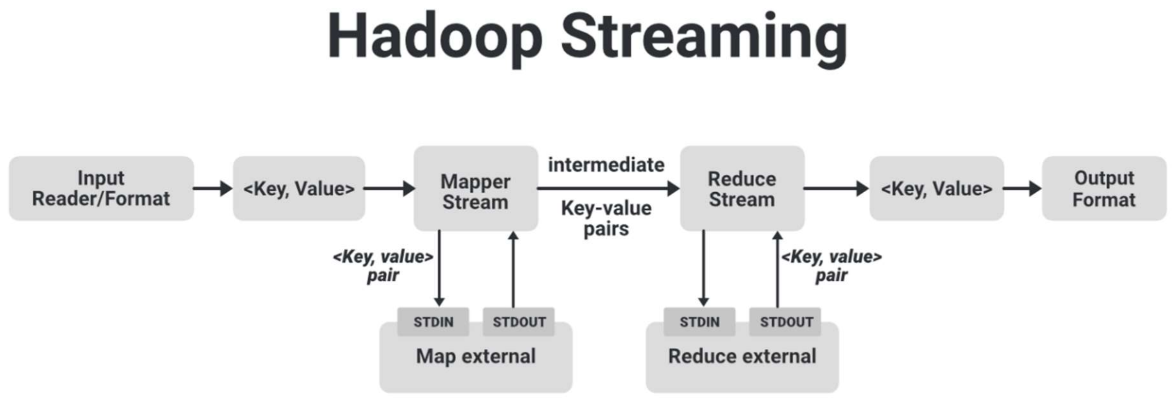

Figure 1: Hadoop Streaming Architecture [4]

6.2.2 Step 2:

Run your MapReduce job with the following commands:

```
hadoop jar /path/to/hadoop-streaming.jar \
    -input /path/to/spotify_million_playlist_dataset \
    -output /path/to/playlist_data \
    -mapper playlist_mapper.py \
    -reducer playlist_reducer.py \
    -file playlist_mapper.py \
    -file playlist_reducer.py

# Step 2: Enrich the data with genre and audio features
hadoop jar /path/to/hadoop-streaming.jar \
    -input /path/to/playlist_data \
    -output /path/to/enriched_data \
    -mapper enrich_mapper.py \
    -reducer enrich_reducer.py \
    -file enrich_mapper.py \
    -file enrich_reducer.py
```

6.3 First MapReduce Phase:

The primary objective of the first MapReduce phase was to remove any unnecessary data and clean the dataset. By employing data cleansing techniques during this stage, we aimed to eliminate unnecessary data to reduce memory requirements and increase performance of our recommendation system. This process ensured that the dataset was in an optimal format for subsequent analysis and processing, laying a solid foundation for the next stages of our methodology. The code snippet provided demonstrates how this process was implemented:

```python
# Read the JSON files and emit (playlist_id, track) pairs where track is a JSON object
def extract_tracks(playlist):
    playlist_id = playlist["pid"]
    tracks = playlist.get("tracks", [])
    for track in tracks:
        # Emit the playlist_id and track dictionary
        track_dict = {
            "track_uri": track["track_uri"],
            "track_name": track["track_name"],
            "artist_name": track["artist_name"]
        }
        yield playlist_id, track_dict

# Read the entire input from stdin and parse it as JSON
input_data = sys.stdin.read()
data = json.loads(input_data)

# Extract the playlists array
playlists = data.get("playlists", [])

for playlist in playlists:
    for playlist_id, track_dict in extract_tracks(playlist):
        with open("data/mapper_output.txt", "a") as f:
            f.write(f"{playlist_id}\t{track_dict}\n")
        print(f"{playlist_id}\t{track_dict}")
```

The output format of the mapper class can be seen in the results section.

6.3 Second MapReduce Phase:

In the second MapReduce phase, our objective was to aggregate tracks by their corresponding playlist IDs, which would facilitate further analysis and processing. The code snippet provided demonstrates how this process was implemented:

```python
# Group tracks by playlist_id and emit the final playlist data
current_playlist_id = None
tracks = []  # store tracks for each playlist_id

for line in sys.stdin:
    # Read the mapper output and group tracks by playlist_id
    playlist_id, track_dict = line.strip().split('\t', 1)

    if current_playlist_id == playlist_id:  # if the playlist_id is the same as the previous one
        tracks.append(track_dict)  # append the track to the list
    else:
        if current_playlist_id:  # if not None or empty (type: str)
            print(f"{current_playlist_id}\t{json.dumps(tracks)}")
        current_playlist_id = playlist_id  # update the current playlist_id
        tracks = [track_dict]  # start a new list

if current_playlist_id:  # this is to print the last playlist_id and its tracks
    print(f"{current_playlist_id}\t{json.dumps(tracks)}")
```

The output format of the reducer class can be seen in

6.4 Enrichment Phase:

In the enrichment phase, our goal was to augment the dataset with additional information, such as genres and audio features, for each track. In order to enrich our data with these features, the Spotify API was utilized through use of the Spotipy library and a Spotify user API token. Due to the rate limiting of the API calls, this phase was not able to be run in the distributed computing environment powered by Apache Hadoop and Spark. With that, this was the slowest phase in the project. Running the enrichment code took hours to finish running on the output from the previous MapReduce phase.

Enrichment features consist of the following data:

`Genres (List of string values):`

```
['pop', 'dance pop', 'post-teen pop', 'neon pop punk', 'canadian pop]}
```

`Audio Features (float value):`

```
{'danceability',
'energy',
'key',
'loudness',
'mode',
'speechiness',
'acousticness',
'instrumentalness',
'liveness',
'valence',
'tempo'}
```

With these features we can create a recommendation based on the audio and genre features of the track.

The code snippet provided demonstrates how this process was implemented:

```python
def batch(iterable, size=50):

    """Split an iterable into batches of the given size."""
    for i in range(0, len(iterable), size):
        yield iterable[i:i + size]

def get_genres_and_audio_features(track_uris, playlist_id, rate_limit_wait=1):
    genres_list = []
    audio_features_list = []

    try:
        for track_uris_batch in batch(track_uris):
            print("Requesting track details...")
            time.sleep(rate_limit_wait)
            print(track_uris_batch)
            tracks = sp.tracks(track_uris_batch)

            print("Requesting artist details...")
            time.sleep(rate_limit_wait)
            artist_uris = [track['artists'][0]['uri'] if track and track['artists'] else '' for track in
tracks['tracks']]
            artists = sp.artists([uri for uri in artist_uris if uri])

            print("Requesting audio features...")
            time.sleep(rate_limit_wait)
            audio_features_batch = sp.audio_features(track_uris_batch)

            genres_list.extend([artist['genres'] for artist in artists['artists']])
            audio_features_list.extend(audio_features_batch)

        return genres_list, audio_features_list
    except Exception as e:
        print("An error occurred:", str(e))
        traceback.print_exc()

        with open("data/timeout_log.txt", "a") as log_file:
            log_file.write(f"Error occurred for playlist_id: {playlist_id}\n")
            log_file.write(f"Track: {track_uris_batch}\n")
            log_file.write(f"Genres List: {genres_list}\n")
            log_file.write(f"Audio Features List: {audio_features_list}\n")

        return None, None


with open("data/mapper_output.txt") as f:
    with open("data/mapper_enricher_output.txt", "a") as file:
        for line in f:
            playlist_id, track_uri_json_list_str = line.strip().split('\t', 1)  # read the playlist_id and
track URI json list string
            print("Playlist ID:", playlist_id)
            print("Track URI JSON list string:", track_uri_json_list_str)
            if int(playlist_id) >= int(0):
                track_uri_json_list = json.loads(track_uri_json_list_str)  # parse the track URI json list
string into a list of json objects
                track_uris = [json.loads(track_uri_json)['track_uri'] for track_uri_json in
track_uri_json_list]  # extract track URIs from the json objects

                genres_list, audio_features_list = get_genres_and_audio_features(track_uris, playlist_id)

                enriched_tracks = []

                for i, (genres, audio_features) in enumerate(zip(genres_list, audio_features_list)):
                    track = {
                        'track_uri': track_uris[i],
                        'genres': genres,
                    }
                    track.update(audio_features)
                    enriched_tracks.append(track)


                # Write the playlist_id and the list of enriched tracks as a JSON string
                file.write(f"{playlist_id}\t{json.dumps(enriched_tracks)}\n")
                print(f"{playlist_id}\t{json.dumps(enriched_tracks)}")
```

The batch() function splits the input iterable into smaller batches of a specified size, which is helpful when making API requests to avoid rate limiting. The get_genres_and_audio_features() function takes track URIs and playlist IDs as input and retrieves genres and audio features for each track using the Spotify API. The function handles rate limiting and exception handling to ensure the process runs smoothly.

In the main part of the code, we read the output from the previous MapReduce phase, which consists of playlist IDs and their corresponding track URIs in JSON format. For each playlist, we extract the track URIs and use the get_genres_and_audio_features() function to obtain genres and audio features for each track. We then create a list of enriched tracks, combining the retrieved information with the original track data.

Finally, the enriched data is written to a new file in the format of playlist IDs paired with their corresponding enriched tracks in JSON format. This enriched dataset, containing valuable information about genres and audio features, forms the foundation for the subsequent analysis and feature extraction processes, enabling us to generate more accurate and personalized music recommendations.

6.5 Recommendation Phase:

For the recommendations we will be utilizing the similarity metric, cosine similarity. Cosine similarity is a widely used metric in the field of information retrieval and machine learning that measures the similarity between two vectors by calculating the cosine of the angle between them. It ranges from -1 (completely dissimilar) to 1 (identical), with 0 indicating no similarity. The advantage of cosine similarity over other distance metrics, such as Euclidean distance, lies in its ability to account for the relative orientation of the vectors rather than their magnitudes. This makes it particularly suitable for comparing high-dimensional, sparse data, as is often encountered in text or music recommendation systems. [2]

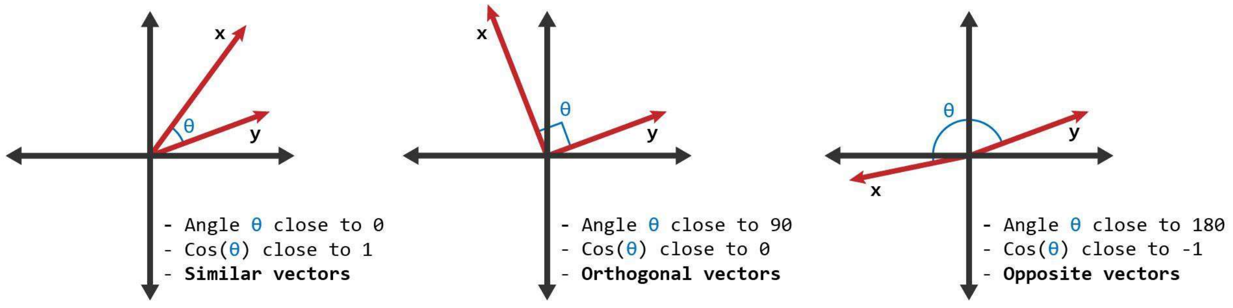

Figure 2: Visualization of cosine similarity [5]

Although our data is not sparse, we have also opted for the use of cosine similarity since it has been widely adopted and proven effective in various recommendation system applications and is quite simple to implement. By employing cosine similarity to measure the similarity between songs and the average vector representation of each playlist, we aim to generate personalized and accurate song recommendations that reflect the unique characteristics and preferences that the target playlist is attempting to embody.

The recommendation system we have created aims to identify the top 10 unique tracks that are most similar to a given target playlist based on a combination of genre similarity and audio feature similarity. In order to prevent multiple recommendations of the same artist, we have ensured the output recommendation set consists of unique artists. The recommendation system includes various functions that calculate genre similarity, normalize audio features, and determine cosine similarity between tracks. Additionally, it computes the average genres and audio features for the target playlist that represents the "overall theme" of the music within the playlist. The get_top_10_unique_matches_for_playlist() function combines these methods to calculate a weighted similarity score for each track in the dataset relative to the target playlist, considering both genre similarity and audio feature similarity. For this project we utilized the scikit cosine_similarity method.

The code snippet provided demonstrates how this process was implemented:

```
def genre_similarity(track1, track2):
    genres1 = set(track1['genres'])
    genres2 = set(track2['genres'])
    common_genres = len(genres1.intersection(genres2))
    total_genres = len(genres1.union(genres2))
    return common_genres / total_genres if total_genres > 0 else 0


def normalize_audio_features(data, target_track):
    scaler = MinMaxScaler()
    audio_features = ['danceability', 'energy', 'key', 'loudness', 'mode', 'speechiness', 'acousticness',
'instrumentalness', 'liveness', 'valence', 'tempo']
    for feature in audio_features:
        feature_values = [track[feature] for playlist in data.values() for track in playlist]
        feature_values.append(target_track[feature])
        scaler.fit(np.array(feature_values).reshape(-1, 1))
        for playlist_id, tracks in data.items():
            for track in tracks:
                track[f"{feature}_norm"] = scaler.transform([[track[feature]]])[0][0]
        target_track[f"{feature}_norm"] = scaler.transform([[target_track[feature]]])[0][0]

def cosine_similarity_tracks(track1, track2):
    audio_features_norm = ['danceability_norm', 'energy_norm', 'key_norm', 'loudness_norm', 'mode_norm',
'speechiness_norm', 'acousticness_norm', 'instrumentalness_norm', 'liveness_norm', 'valence_norm',
'tempo_norm']
    vector1 = np.array([track1[feature] for feature in audio_features_norm]).reshape(1, -1)
    vector2 = np.array([track2[feature] for feature in audio_features_norm]).reshape(1, -1)
    return cosine_similarity(vector1, vector2)[0][0]

# return top ten most common genres by count
def get_average_genres(playlist):
    genre_counter = defaultdict(int)
    for track in playlist:
        for genre in track['genres']:
            genre_counter[genre] += 1

    # Sort genres by count and return the top 10 genres
    sorted_genres = sorted(genre_counter.items(), key=lambda x: x[1], reverse=True)
    top_10_genres = [genre for genre, count in sorted_genres[:10]]

    return {'genres': top_10_genres}

def get_average_audio_features(playlist):
    audio_features = ['danceability', 'energy', 'key', 'loudness', 'mode', 'speechiness', 'acousticness',
'instrumentalness', 'liveness', 'valence', 'tempo']
    total_tracks = len(playlist)
    avg_audio_features = {feature: 0 for feature in audio_features}
    for track in playlist:
        for feature in audio_features:
            avg_audio_features[feature] += track[feature]
    return {feature: value / total_tracks for feature, value in avg_audio_features.items()}

def get_top_10_unique_matches_for_playlist(target_playlist, data, genre_weight=0.3, feature_weight=0.7):
    print("Getting top 10 unique matches for target playlist.")
    target_avg_genres = get_average_genres(target_playlist)
    print("Target playlist average genres: ", target_avg_genres)
    target_avg_audio_features = get_average_audio_features(target_playlist)
    print("Target playlist average audio features: ", target_avg_audio_features)

    target_track = {
        **target_avg_genres,
        **target_avg_audio_features
    }

    print("Normalizing audio features...")
    normalize_audio_features(data, target_track)

    similarity_scores = []
    print("Calculating similarity scores...")
    for tracks in data.values():
        for track in tracks:
            if track in target_playlist:
                continue
            genre_sim = genre_similarity(target_track, track)
            cosine_sim = cosine_similarity_tracks(target_track, track)
            weighted_similarity = genre_weight * genre_sim + feature_weight * cosine_sim
            similarity_scores.append((weighted_similarity, {**track, 'weighted_similarity':
weighted_similarity}))

    print("Sorting similarity scores...")
    similarity_scores.sort(reverse=True, key=lambda x: x[0])

    # Ensure no two matches have the same artist
    print("Ensuring no two matches have the same artist...")
    top_10_unique_matches = []
    for score, track in similarity_scores:
        if len(top_10_unique_matches) == 10:
            break
        artists = [track['artist_name']]
        if not any(artist in reduce(lambda x, y: x + [y], [track['artist_name'] for track in
top_10_unique_matches], []) for artist in artists):
            top_10_unique_matches.append(track)

    print("Top 10 unique matches for target playlist are complete.")
    return top_10_unique_matches


# Read data from file
file_path = "data/mapper_enricher_output.txt"
with open(file_path, "r") as f:
    data = {}
    for line in f:
        playlist_id, tracks_dict = line.strip().split('\t', 1)
        tracks = json.loads(tracks_dict)
        data[playlist_id] = tracks

# Define target_playlist
playlist_id = '0'
target_playlist = data[playlist_id]

# Get top 10 unique matches for the target_playlist
top_10_matches = get_top_10_unique_matches_for_playlist(target_playlist, data)

# Print the top 10 unique matches to file and console
with open("data/top_10_matches.txt", "w") as f:
    f.write(f"Target Playlist: {playlist_id}\n")
    # write the track name and artist
    for i, match in enumerate(top_10_matches, start=1):
        f.write(f"{i}. {match['track_name']} - {match['artist_name']} -Weighted Similarity:
{round(match['weighted_similarity'], 4)}\n")
        print(f"{i}. {match['track_name']} - {match['artist_name']} - Weighted Similarity:
{round(match['weighted_similarity'], 4)}")
```

This recommendation system offers a personalized approach to music recommendations, as it considers both the genre preferences and audio feature patterns within a user's playlist to suggest new tracks that align with their listening habits.

7. RESULTS:

Here the outputs from the mapper, reducer, enricher, and recommendation engine are shown.

7.1 Mapper Output

The input format is seen in the data section. The output after the mapper (section 6.3) was a series of key-value pairs consisting of the playlist ID and a single track in that playlist.

```
<playlist pid>  < str(track JSON object)>

0       "{\"track_uri\": \"spotify:track:5Q0Nhxo0l2bP3pNjpGJwV1\", \"track_name\": \"Party In The U.S.A.\", \"artist_name\": \"Miley Cyrus}"

0       "{\"track_uri\": \"spotify:track:6GIrIt2M39wEGwjCQjGChX\", \"track_name\": \"The Great Escape\", \"artist_name\": \"Boys Like Girls\"}"

0       "{\"track_uri\": \"spotify:track:4E5P1XyAFtrjpiIxkydly4\", \"track_name\": \"Replay\", \"artist_name\": \"Iyaz\"}"

…

…
```

7.2 Reducer Output

The input format is the output above. The output after the reducer (section 6.4) was a series of aggregated key-value pairs consisting of the playlist ID and a list of track JSON objects consisting of all the tracks in the playlist.

<playlist pid> < List of str(track JSON object)>

```
0   ["{\"track_uri\": \"spotify:track:5Q0Nhxo0l2bP3pNjpGJwV1\", \"track_name\": \"Party In The U.S.A.\", \"artist_name\": \"Miley Cyrus}",
    "{\"track_uri\": \"spotify:track:6GIrIt2M39wEGwjCQjGChX\", \"track_name\": \"The Great Escape\", \"artist_name\": \"Boys Like Girls\"}",
    "{\"track_uri\": \"spotify:track:4E5P1XyAFtrjpiIxkydly4\", \"track_name\": \"Replay\", \"artist_name\": \"Iyaz\"}, …"]

1   ["{\"track_uri\": \"spotify:track:39C5FuZ8C8M0QI8CrMsPkR\", \"track_name\": \"Foreplay / Long Time\", \"artist_name\": \"Boston\"}",
    "{\"track_uri\": \"spotify:track:1GqlvSEtMx5xbGptxOTTyk\", \"track_name\": \"Peace of Mind\", \"artist_name\": \"Boston\"}, …"]
```

7.3 Enricher Output

The input format is the output above. The output after the enrichment phase (section 6.5) was the same as the reducer output with the exception of an additional genre dictionary object and a list of audio features added to the track JSON. We did not use the added values "id", "uri", "track_href", "analysis_url", "duration_ms", and "time_signature".

<playlist pid> < List of str(track JSON object)>

```
0   [{"track_uri": "spotify:track:5Q0Nhxo0l2bP3pNjpGJwV1",
    "track_name": "Party In The U.S.A.",
    "artist_name": "Miley Cyrus",
    "genres": ["pop"],
    "danceability": 0.652,
    "energy": 0.698,
    "key": 10,
    "loudness": -4.667,
    "mode": 0,
    "speechiness": 0.042,
    "acousticness": 0.00112,
    "instrumentalness": 0.000115,
    "liveness": 0.0886,
    "valence": 0.47,
    "tempo": 96.021,
    "type":
    "audio_features",
    "id": "5Q0Nhxo0l2bP3pNjpGJwV1",
    "uri": "spotify:track:5Q0Nhxo0l2bP3pNjpGJwV1",
    "track_href": "https://api.spotify.com/v1/tracks/5Q0Nhxo0l2bP3pNjpGJwV1",
    "analysis_url": "https://api.spotify.com/v1/audio-analysis/5Q0Nhxo0l2bP3pNjpGJwV1",
    "duration_ms": 202067,
    "time_signature": 4}, …, …, …, …}]
```

7.4 Recommendation Engine Output

The recommendation engine input is the output of the enrichment phase seen above. The recommendation engine gathers the genres and audio features from each track in the target playlist and generates an average representation for the playlist. The generated average top 10 genres and the average audio features are seen below:

Target playlist average genres:

```
{'genres': ['pop', 'dance pop', 'r&b', 'urban contemporary', 'rap', 'pop punk', 'post-teen pop', 'neon pop punk', 'canadian pop', 'pop rap']}
```

Target playlist average audio features:

```
{'danceability': 0.6640769230769232, 'energy': 0.7810769230769232, 'key': 5.038461538461538, 'loudness': -4.8912115384615396, 'mode': 0.6923076923076923, 'speechiness': 0.10369807692307693, 'acousticness': 0.0836741134615385, 'instrumentalness': 0.0006743819230769231, 'liveness': 0.1870865384615384, 'valence': 0.6427500000000002, 'tempo': 121.1575}
```

From the average representation vectors of the playlist the recommendation engine output the following unique artists.

```
Target Playlist ID: 0

1. Caught Up - Usher -Weighted Similarity: 0.8072

2. So You Can Cry - Ne-Yo -Weighted Similarity: 0.7996

3. Dan Bilzerian - T-Pain -Weighted Similarity: 0.796

4. Let Me Hold You - Bow Wow -Weighted Similarity: 0.7934

5. Overdose - Ciara -Weighted Similarity: 0.7923

6. Blurred Lines - Robin Thicke -Weighted Similarity: 0.7919

7. Long Way 2 Go - Cassie -Weighted Similarity: 0.7881

8. The Best Things In Life Are Free - Janet Jackson -Weighted Similarity: 0.7819

9. Sexy And I Know It - LMFAO -Weighted Similarity: 0.7809

10. I Really Like You - Carly Rae Jepsen -Weighted Similarity: 0.7806
```

Prior to ensuring the recommendations consisted of unique artists, the recommendation engine tended to recommend multiple tracks by the same artist as seen below:

```
Top 10 unique matches for target playlist are complete.

1. Caught Up - Usher - Weighted Similarity: 0.8072

2. So You Can Cry - Ne-Yo - Weighted Similarity: 0.7996

3. Dan Bilzerian - T-Pain - Weighted Similarity: 0.796

4. Best Love Song - T-Pain - Weighted Similarity: 0.7938

5. Let Me Hold You - Bow Wow - Weighted Similarity: 0.7934

6. Overdose - Ciara - Weighted Similarity: 0.7923

7. Blurred Lines - Robin Thicke - Weighted Similarity: 0.7919

8. So Sick - Ne-Yo - Weighted Similarity: 0.7897

9. Outta My System - Bow Wow - Weighted Similarity: 0.7883

10. Long Way 2 Go - Cassie - Weighted Similarity: 0.7881
```

Generating unique artists allows for a more diverse recommendation system.

8. DISCUSSION:

8.1 Results

Upon implementing the recommendation engine and evaluating the outcomes, it can be concluded that the recommendations generally exhibit high quality. The engine exclusively generates unique artists, thereby eliminating repetition and excessive similarity in music selections. By considering the user's interests and preferences, the recommendation engine successfully generated a tailored list of suggested songs. Furthermore, the engine identified similarities between tracks and recommended analogous songs by examining the playlist's content and the tracks' features.

Upon reviewing the songs generated by the aforementioned process, it can be inferred that the songs possess sufficient similarity in terms of genre and audio characteristics. The recommended songs showcase diversity in terms of artists and release dates. From a personal perspective, the suggested songs exhibited consistency in genre, mood, and tempo, making for a satisfactory recommendation. However, the effectiveness of the recommendation engine could be considered subjective, as individual users possess unique musical preferences and playlist organization methods.

8.2 Limitations

The recommendation engine encountered several limitations, such as its reliance on the enriched data, which constituted a mere fraction of the comprehensive million-song dataset (9,000 out of the 1,000,000 playlists). For users with eclectic musical tastes not adequately represented within the limited dataset, this constraint could hinder the scalability of the recommendations.

One potential solution to address this limitation involves analyzing the complete dataset. Another approach is to incorporate user feedback to surmount this restriction and refine the recommendation engine. The optimal strategy for enhancing the recommendation engine, effectively catering to users, and providing personalized and novel song suggestions entails the integration of user feedback.

The technique employed by the recommendation engine, which averages the data's attributes, may not yield the most fitting song selections if the playlist encompasses a broad array of musical styles. This is attributable to the possibility that the resulting recommendations might not align with the playlist's overarching theme or mood. For instance, a playlist may contain both energetic pop music and slow ballads, and the recommendation engine might provide suggestions that straddle these two genres. Consequently, the suggested songs may not accurately reflect the user's intended mood or theme, potentially diminishing the user's listening experience.

8.3 Alternative methods

We contemplated the Scalable Optimal Neighborhood (SON) algorithm, which could have been utilized to identify frequent item sets representing combinations of songs that regularly appear together to predict recommendations for automatic playlist continuation. Nevertheless, we opted for the cosine similarity algorithm due to its simplicity and efficacy in discerning similarities between distinct songs. The SON algorithm is typically employed for detecting frequent itemsets, which may not necessarily correlate with identifying song similarities. Given the constraint of working with a subset of the Spotify dataset, the cosine similarity algorithm proved to be the superior choice for identifying musical similarities among the songs in the playlist, rather than frequent itemsets.

Project Contribution:

Hermela Darebo was responsible for the development of both the mapper and reducer components, while Eric Martin focused on the enricher, recommendation engine, data and cluster management, and code execution. Both worked collaboratively on the preparation of the paper and presentation.

References:

1. *“Spotify Million Playlist Dataset Challenge: Challenges,” AIcrowd. [Online]. Available: https://www.aicrowd.com/challenges/spotify-million-playlist-dataset-challenge. [Accessed: 26-Apr-2023].*

2. *J. Han, M. Kamber, and J. Pei, “Getting to know your data,” Data Mining, pp. 39–82, 2012.*

3. *“Hadoop streaming,” Hadoop 1.2.1 Documentation, 18-May-2022. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/streaming.html. [Accessed: 28-Apr-2023].*

4. *“What is Hadoop streaming?,” GeeksforGeeks, 03-Jan-2021. [Online]. Available: https://www.geeksforgeeks.org/what-is-hadoop-streaming/. [Accessed: 28-Apr-2023].*

5. *Author: Fatih Karabiber Ph.D. in Computer Engineering, Fatih Karabiber Ph.D. in Computer Engineering, E. R. Psychometrician, and E. B. F. of LearnDataSci, “Cosine similarity,” Learn Data Science - Tutorials, Books, Courses, and More. [Online]. Available: https://www.learndatasci.com/glossary/cosine-similarity/. [Accessed: 28-Apr-2023].*