VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
INTERNATIONAL UNIVERSITY

PROJECT REPORT

SONGS RECOMMENDATION SYSTEM USING PYSPARK

BIG DATA TECHONOLOGY

Course by

Dr. Van Ho Long

Student in charge:

Full name	Student ID	Contribution
Nguyen Xuan Tram Anh	ITDSIU22177	33%
Dang Phuong Mai	ITDSIU22172	33%
Tram Phuong Nghi	ITDSIU22145	33%

Sumário

1 Overview

In this project, we present a music recommendation system that leverages big data technology to enhance user experience in playlist creation. Utilizing PySpark for data processing and analysis, the system is designed to handle large-scale music datasets efficiently. The core functionality includes a web-based interface that allows users to search for songs, create personalized playlists, and receive recommendations for additional tracks based on their selections.

The recommendation engine utilizes advanced machine learning algorithms implemented in PySpark to analyze and identify patterns in song preferences. The system generates playlists tailored to individual tastes. The scalability of PySpark enables the processing of extensive music datasets, ensuring timely and accurate recommendations even as the number of users and data grows. This integration of big data processing with personalized recommendations showcases the system's potential to enhance user engagement and satisfaction in the music streaming experience.

2 Methodology

2.1 Development Process

2.1.1 Front End

This report outlines the development process of the front-end interface for the Music Recommendation System. The objective of this system is to allow users to search for songs, create personalized playlists, and receive recommendations based on their selected tracks. The front-end implementation focuses on user-friendly navigation, seamless interactions, and visually appealing design to enhance the user experience.

Development Goals:

- Create an intuitive and responsive UI that is accessible across devices.
- Enable dynamic interaction with back-end APIs for functionalities like searching songs and fetching recommendations.
- Provide a visually appealing layout with animations to engage users.

1. Framework and technology selection

- HTML: Used to structure the webpage with semantic tags for better readability

and maintainability.

- CSS: Styled the interface for an engaging and modern look, including animations, transitions, and custom scrollbars.

- JavaScript: Implemented dynamic functionality for the user interface, enabling seamless interaction between the user and the system.

2. Wireframe design

The system's layout was designed with simplicity and functionality in mind:

- A header section for system branding.

- Two primary sections:

+ Left Panel: Search bar, search results, and playlist management.

+ Right Panel: Song recommendations with dynamic update functionality.

- Wireframes were reviewed iteratively to refine user experience.

3. HTML Implementation

The application's HTML was structured to separate concerns and ensure maintainability. Key sections include:

- Header section:

+ Title: "Music Recommendation System."

+ Brief description: A single-line overview of the application's functionality.

- Content section:

+ Left panel: Contains the search bar, a dynamic list for search results, and a scrollable playlist area.

+ Right panel: Displays the recommended songs and a button to fetch updated recommendations.

- Accessibility:

+ Semantic HTML tags were used to improve navigation for screen readers.

+ ARIA roles and labels were added to key elements like buttons and search inputs.

4. CSS Styling

The CSS implementation focused on responsiveness, aesthetics, and interactivity:

- Global styles:

+ Fonts: Integrated Google Fonts, using "Montserrat" and "Poppins" for a modern and clean look.

+ Colors: Used a minimalist palette with contrasting colors to emphasize important elements (e.g., primary buttons).

- Layout:

+ Flexbox: Employed flexbox for responsive alignment and consistent spacing between sections.

+ Scrollable areas: Applied custom-styled scrollbars for the playlist and recommendation sections.

Animations:

+ Fade-in effects for the header and panels upon page load.

+ Button hover effects with smooth transitions.

+ Scroll animations for dynamic updates in the playlist and recommendation panels.

5. Javascript implementation

JavaScript powered the dynamic features of the application, including API integration, DOM manipulation, and user interactions. Key functionalities include:

- Dynamic content rendering:

+ Search results are fetched and rendered dynamically based on user input.

+ Playlists and recommendations are updated in real-time without reloading the page.

- Functions developed:

+ fetchInitialRecommendations: Retrieves and displays the top 10 trending songs when the page loads.

+ searchSongs: Listens for user input in the search bar and queries the back-end to fetch relevant results.

+ addToPlaylist: Allows users to add a song to their playlist with a single click, dynamically updating the playlist area.

+ fetchRecommendations: Generates and displays personalized recommendations based on the current playlist.

- Event listeners:

+ Buttons: Click listeners for search and refresh actions.

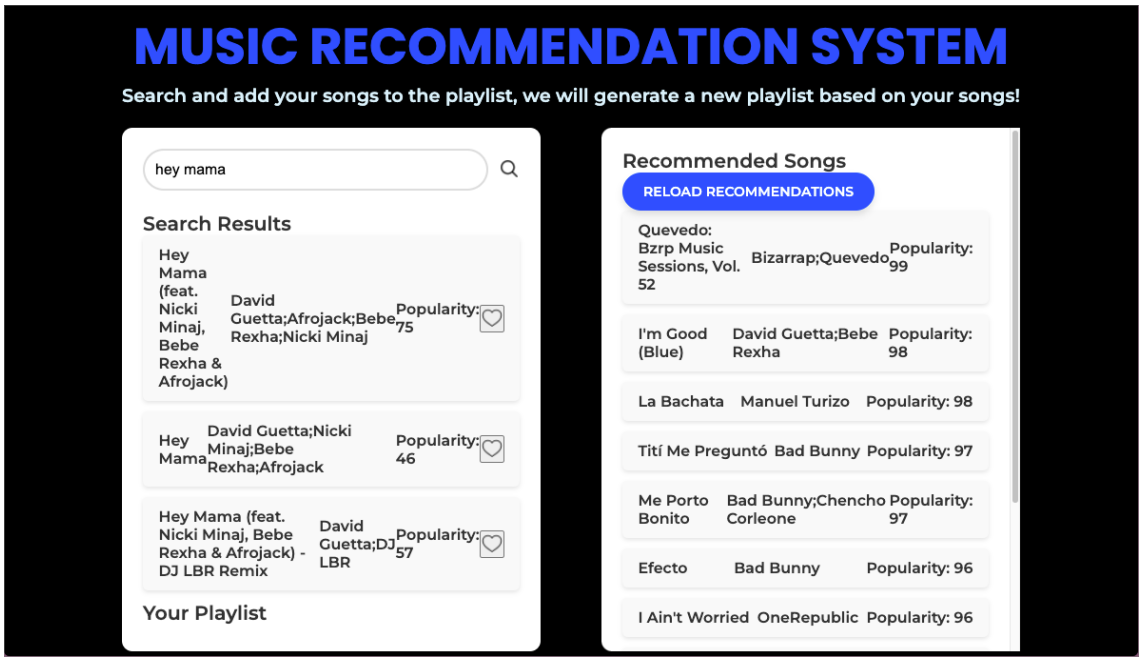
+ Dynamic elements: Listeners attached to dynamically created elements, like adding songs to the playlist.

- Error handling:

+ Displayed user-friendly error messages (e.g., "No results found" or "Unable to

fetch recommendations").

+ Implemented fallback mechanisms for failed API calls, including retry attempts.



2.1.2 Back End

The backend of this website serves as the core engine for a music recommendation system. Built using Flask, it integrates data processing with PySpark for handling song datasets and performs complex recommendation logic. It employs a multi-threaded architecture to enhance responsiveness and integrates RESTful APIs for interaction with the frontend.

2.1.3 Backend Features and Implementation

1. Data Management

+ Flask: Acts as the primary web framework for serving HTML templates, handling API requests, and managing the application flow.

+ PySpark: Processes and manipulates large-scale music datasets efficiently.

+ Flask-Limiter: Implements rate-limiting to secure the API from abuse.

+ ThreadPoolExecutor: Handles background tasks asynchronously to improve responsiveness.

2. Framework and Libraries

+ Dataset Loading: A preprocessed dataset of songs in Parquet format is loaded and

cached into memory during server startup to optimize repeated query performance.

+ Data Cleaning: The system cleans song data by trimming whitespace and handling case-insensitivity during searches.

3. APIs for Recommendations

Initial Recommendations:

+ Fetches the top 10 most popular songs based on their popularity scores.

Search Songs:

+ Accepts a user-provided query and performs a case-insensitive search on the dataset.

Recommendations Based on Playlist:

+ Matches songs from the user-selected playlist with the dataset, computes similarity metrics using features such as cosine similarity, and recommends songs not already in the playlist.

Custom Song Recommendation:

+ Handles user-provided song names and performs recommendation logic in the background.

4. Advanced Recommendation Logic

Cosine Similarity:

A user-defined function cosine similarity udf which is calculates the similarity between features of a given playlist and the dataset.

Popularity and Feature-Based Sorting:

Recommendations are sorted by similarity and popularity to ensure the most relevant results are displayed.

5. Error Handling

Implements error catching at the API level to return informative messages and maintain robustness during unexpected scenarios, such as missing data or query issues.

6. Frontend Integration

Returns JSON responses to be dynamically consumed by the frontend, enabling interactive user features like live search and playlist-based recommendations.

7. Security and Performance

+ Utilizes Flask-Limiter to control API usage rates and prevent abuse.

+ Caches frequently accessed datasets to reduce processing time and improve throughput.

8. Scalability

+ By leveraging PySpark, the backend can scale efficiently with large datasets.

+ Multi-threaded execution supports parallel processing for long-running tasks, ensuring the system remains responsive.

2.1.4 Deployment and Execution

The application runs locally for development (debug=True) but is scalable for production environments with proper optimizations. Port configurations and Spark session settings ensure compatibility across different deployment setups.

2.2 Modeling Process

2.2.1 Preprocessing

The goal of preprocessing step is to ensure data validity and integrity before ingesting into the recommendation model. The processing steps involve handling missing and duplicate values, checking for data types and handling any inconsistency in data.

1) Create SparkSession and load dataset

```
# Start a Spark session
spark = SparkSession.builder \
    .appName("PreprocessSongData") \
    .getOrCreate()

# Preprocessing
tracks = spark.read.csv("dataset.csv", header=True, inferSchema=True)
```

- Dataset structure:

```
root
 |-- _c0: integer (nullable = true)
 |-- track_id: string (nullable = true)
 |-- artists: string (nullable = true)
 |-- album_name: string (nullable = true)
 |-- track_name: string (nullable = true)
 |-- popularity: float (nullable = true)
 |-- duration_ms: float (nullable = true)
 |-- explicit: string (nullable = true)
 |-- danceability: float (nullable = true)
 |-- energy: float (nullable = true)
 |-- key: float (nullable = true)
 |-- loudness: float (nullable = true)
 |-- mode: float (nullable = true)
 |-- speechiness: float (nullable = true)
 |-- acousticness: float (nullable = true)
 |-- instrumentalness: float (nullable = true)
 |-- liveness: float (nullable = true)
 |-- valence: float (nullable = true)
 |-- tempo: float (nullable = true)
 |-- time_signature: float (nullable = true)
 |-- track_genre: string (nullable = true)
```

2) Data Cleaning

- Identify missing values in the dataset. As the count of missing values is minimal (only one in three columns), remove rows with missing values.

```
tracks = spark.read.csv("dataset.csv", header=True, inferSchema=True)
tracks = tracks.dropna()
```

- Check for duplicate values in track names and remove all rows with duplicates

```
tracks = tracks.dropDuplicates(["track_name"])
```

- Verify numerical columns for non-numeric values, then remove those rows.

```
columns_to_check = ['popularity', 'duration_ms', 'danceability', 'energy', 'key', 'loudness', 'mode', 'speechiness', 'acousticness', 'liveness', 'valenc

# Filter out rows with non-numeric values
for col_name in columns_to_check:
    tracks = tracks.filter(F.col(col_name).rlike('^[+-]?\\d*(\\.\\d+)?([eE][+-]?\\d+)?$'))
```

3) Transformation

- Since machine learning algorithms cannot directly process text when working with text data, the process of tokenizing text features and converting them into numerical vectors is a key. The goal is to encode the genre information into tokens and then transform them into vectors.

```
tokenizer = Tokenizer(inputCol="track_genre", outputCol="genres_token")
tracks = tokenizer.transform(tracks)

cv = CountVectorizer(inputCol="genres_token", outputCol="genres_vector")
cv_model = cv.fit(tracks)
tracks = cv_model.transform(tracks)
```

```
numerical_cols = [col for col, dtype in tracks.dtypes if dtype in ("double", "float")]
assembler = VectorAssembler(inputCols=numerical_cols, outputCol="numerical_features")
tracks = assembler.transform(tracks)
```

```
assembler = VectorAssembler(
    inputCols=["genres_vector", "numerical_features"],
    outputCol="features"
)
tracks = assembler.transform(tracks)
```

- Cosine similarity calculation: The function is used to calculate the similarity between the input song and the feature vectors (derived from numerical columns and genre vectors)

```
def vector_to_numeric_list(vector):
    if isinstance(vector, SparseVector) or isinstance(vector, DenseVector):
        return [float(x) for x in vector.toArray()]
    return vector

vector_to_list_udf = udf(vector_to_numeric_list, ArrayType(DoubleType()))
tracks = tracks.withColumn("features_list", vector_to_list_udf(col("features")))
```

```
# Define cosine similarity function
def cosine_similarity_list(vec1, vec2):
    vec1 = np.array(vec1, dtype=float)
    vec2 = np.array(vec2, dtype=float)
    dot_product = float(np.dot(vec1, vec2))
    norm1 = float(np.linalg.norm(vec1))
    norm2 = float(np.linalg.norm(vec2))
    return dot_product / (norm1 * norm2)


cosine_similarity_udf = udf(cosine_similarity_list, DoubleType())
```

2.2.2 Recommend function

.

```
def recommend_songs(song_name):
    global tracks
    input_song = tracks.filter(col("track_name") == song_name).select("features_list", "artists").collect()

    if not input_song:
        print("This song is either not so popular or you entered an invalid name.\nSome songs you may like:")
        tracks.select("track_name").orderBy(col("popularity").desc()).show(5)
        return

    input_features = input_song[0]["features_list"]
    input_artist = input_song[0]["artists"]

    artist_tracks = tracks.filter(col("artists") == input_artist)


    if artist_tracks.count() == 0:
        print(f"No other songs by {input_artist} found, recommending based on features.")
        artist_tracks = tracks


    tracks_with_similarity = tracks.withColumn(
        "similarity", cosine_similarity_udf(lit(input_features), col("features_list"))
    )


    recommendations = tracks_with_similarity.filter(
        col("track_name") != song_name
    ).orderBy(
        (col("artists") == input_artist).desc(),
        col("similarity").desc(),
        col("popularity").desc()
    )


    recommendations.select("track_name", "artists").show(10)
```

The recommendation algorithm is developed using the content-based recommendation system. It provides personalized suggestions by analyzing the features of an input song, such as its genre, acoustic properties, and other numerical attributes. This is ideal for users seeking songs similar to a favorite track.

- Functionality:

+ Retrieve the name of the input song from the users.

+ Calculate the similarity between the input song's feature and the feature vectors.

+ Sort recommendations based on whether they are the same artist first, then high similarity score and population score.

2.2.3 Recommend function based on popularity

```
def recommend_songs_by_popularity():
    popular_tracks = tracks.orderBy(col("popularity").desc())

    popular_tracks.select("track_name", "artists").show(10)
```

This function recommends songs based solely on their popularity, independent of the user's preferences or input.

- Functionality:

+ The function sorts all songs in the dataset by their popularity in descending order.
+ The top 10 most popular songs are displayed.

3 Summary

The report details the development of a Songs Recommendation System using PySpark to enhance playlist creation through personalized song suggestions. The system features a web-based interface for searching songs, managing playlists, and receiving recommendations based on user preferences.

The front-end was built with HTML, CSS, and JavaScript, emphasizing user-friendliness, responsiveness, and dynamic interactions. The back-end utilizes Flask and PySpark for data processing, implementing APIs for personalized and popularity-based recommendations. Advanced recommendation logic includes cosine similarity and feature-based sorting.

Preprocessing steps addressed data cleaning, handling missing and duplicate values, and transforming text into numerical vectors. The system supports both content-based and popularity-based recommendations, ensuring scalability and responsiveness through multi-threading and parallel processing.

The project demonstrates the effective use of big data and machine learning to deliver scalable, efficient, and personalized music recommendations.

4 References

[1] GeeksforGeeks. (2022, November 1). Music Recommendation System Using Machine Learning. GeeksforGeeks. https://www.geeksforgeeks.org/music-recommendation-system-using-machine-learning/

[2] Tony Teaches Tech. (2023, March 9). How to Make a Website with Python (Flask app tutorial) [Video]. YouTube. https://www.youtube.com/watch?v=qaBo$_I iE4Gc$