Music Recommendation System with Docker, MLflow, Airflow and Streamlit

Submitted By

Full Name	Enrollment No.
Aman Saini	M24CSE003
Neha Sharma	M24CSE014
Anand Saxena	D24CSA001

GitHub Repository Link
Streamlit App Link

Department of Computer Science and Engineering

Contents

Music Recommendation System with Docker, MLflow, Airflow and Streamlit

Abstract

This report provides an in-depth overview of the design, development, and implementation of a state-of-the-art Music Recommendation System. The system leverages advanced machine learning techniques to generate personalized music recommendations based on individual user preferences, improving user experience in the music streaming domain. The recommendation process is powered by a combination of collaborative filtering, genre-based similarity metrics, and historical data analysis. By analyzing user behavior, listening patterns, and other attributes, the system tailors its suggestions to align with the tastes of each user, offering a highly dynamic and personalized music discovery process. The system is designed to scale, process large datasets, and continually adapt to evolving user needs.

The architecture of the system incorporates several cutting-edge tools and technologies to streamline the development and deployment phases. MLflow is utilized for experiment tracking and model management, ensuring reproducibility and consistency across machine learning experiments. Apache Airflow orchestrates data pipelines, automating workflows for data extraction, transformation, and loading (ETL). Streamlit serves as the user interface for presenting the recommendations, providing an interactive and seamless experience for users. Docker is employed to containerize the application, ensuring consistent performance and facilitating easy deployment across different environments. The entire project is version-controlled and managed via GitHub, ensuring collaboration and efficient tracking of changes. This report details the technical implementation, the integration of various technologies, and the potential for future improvements, demonstrating how such a system can significantly enhance music discovery and user engagement in the rapidly growing music streaming industry.

1 Introduction

The Music Recommendation System is a sophisticated solution engineered to revolutionize the user experience in the music streaming domain by delivering highly personalized music suggestions. This system integrates a multi-faceted approach encompassing data preprocessing, advanced model training, and real-time recommendation delivery. The primary objective is to cater to the escalating demand for individualized content in the music streaming industry, a sector that has seen exponential growth with the advent of digital platforms. By leveraging a robust infrastructure for development and deployment, the system aims to provide seamless scalability and adaptability to evolving user preferences and industry trends. This initiative not only enhances user engagement but also sets a benchmark for leveraging big data and machine learning in entertainment technology.

1. **Objective**: Develop a scalable and efficient recommendation engine for personalized music streaming.
2. **Scope**: Integration of historical user data, real-time processing, and deployment via containerized environments.
3. **Technology Stack**: Utilizes Python, Docker, Apache Airflow, and MLflow for a comprehensive development lifecycle.
4. **Industry Relevance**: Addresses the growing need for personalized content in a competitive music streaming market.

2 Dataset Details

The Music Recommendation System relies on an extensive and well-structured dataset housed within the `data/datasets` directory, which serves as the foundational element for generating accurate and personalized recommendations. This dataset is composed of a rich array of features, including `uris`, `names`, `artist_names`, `artist_uris`, `artist_pop`, `artist_genres`, `albums`, `track_pop`, `danceability`, `energy`, `keys`, `loudness`, `modes`, `speechiness`, `acousticness`, `instrumentalness`, `liveness`, `valences`, `tempos`, `types`, `ids`, `track_hrefs`, `analysis_urls`, `durations_ms`, and `time_signatures`, alongside the `playlist_name` for contextual relevance. These attributes provide a holistic view of musical content, capturing both technical audio characteristics and metadata such as artist popularity and genre classifications. The data is meticulously preprocessed using scripts within the `dags` directory to ensure high quality, consistency, and compatibility with the recommendation algorithms. Regular updates and retraining schedules are meticulously managed through the `scheduler` and `2025-04-09` directories, ensuring the system remains current with the latest user interactions and music trends.

1. **Data Volume**: Comprises large set of records, enabling robust statistical analysis and model training.
2. **Feature Richness**: Includes audio features (e.g., `danceability`, `energy`) and metadata (e.g., `artist_genres`, `track_pop`) for comprehensive profiling.
3. **Preprocessing**: Utilizes `check_files.py` to validate data integrity and handle missing values or inconsistencies.
4. **Update Frequency**: Scheduled retraining aligns with data updates, managed via Airflow workflows.

3 Methodology

The methodology for developing the Music Recommendation System is a structured, multi-stage process designed to ensure the creation of a robust, scalable, and accurate recommendation engine.

3.1 Data Collection

- Raw data is sourced and stored in the `data` directory.
- Primary dataset file is `tracks.csv`, containing features such as user listening histories, song metadata (e.g., `names`, `artist_names`, `uris`), and audio characteristics (e.g., `danceability`, `energy`, `track_pop`).
- `check_files.py` validates `tracks.csv` availability within `/opt/airflow/datasets/` by checking file existence and directory contents.
- Leverages Docker-mounted volume `./datasets:/opt/airflow/datasets` from `docker-compose.yml` for consistency across the containerized environment.

3.2 Data Preprocessing

- Transforms raw data into a suitable format for model training.
- `check_files.py` initializes by verifying `tracks.csv` integrity, listing contents of `/opt/airflow` and `/opt/airflow/datasets`.
- `music_recommendation_retraining.py` and `train_and_deploy.py` load `tracks.csv` using `pd.read_csv`.
- `music_recommendation_retraining.py` pushes preprocessed data to Airflow's XCom via `load_and_preprocess_data` task.
- `parse_genres` function extracts genres from `artist_genres` using regular expressions.
- Ensures data quality and compatibility with recommendation algorithms.

3.3 Feature Engineering

- Enhances dataset by deriving new features to improve model performance.
- `music_recommendation_retraining.py`'s `create_user_genre_matrix_task` generates a user-genre interaction matrix using `create_user_genre_matrix`.
- Randomly samples `track_pop` for each user-genre pair, pivoting data into a matrix filled with zeros where no interactions exist.
- `train_and_deploy.py` performs the same step independently.
- Leverages rich feature set (e.g., `tempos`, `loudness`, `artist_genres`) to capture nuanced user preferences.
- Integrated into Airflow DAG or standalone execution pipeline.

3.4 Model Training

- Conducted using a collaborative filtering approach via `CollaborativeFilteringModel` class in `music_recommendation_retraining.py` and `train_and_deploy.py`.
- Preprocessed user-genre matrix split into train (80%) and test (20%) sets using `train_test_split`.
- Model initialized with `n_similar_users=5`, using cosine similarity for predictions.
- `music_recommendation_retraining.py`'s `train_and_evaluate_model` task trains, evaluates with MSE, logs to MLflow at `http://mlflow:5000`, and saves to `/opt/airflow/models/collaborative_filtering_model.pkl`.
- `train_and_deploy.py` performs standalone training, saving to `models/collaborative_filtering_model.pkl`.
- `mlruns` directory tracks experiments, with `automated-ml-training.iml` potentially automating this phase.

3.5 Workflow Orchestration

- Orchestrated using Apache Airflow, configured in `dags` directory via `docker-compose.yml`'s `airflow-webserver` and `airflow-scheduler` services.
- `dag_processor_manager` manages DAGs, with tasks (e.g., `load_and_preprocess_data`, `train_and_evaluate_model`) defined in `music_recommendation_retraining.py`.
- Schedules in `2025-04-08`, and `2025-04-09` log retraining details, executed daily per DAG's `schedule_interval`.
- `scheduler` service depends on PostgreSQL (`postgres`) and MLflow (`mlflow`).
- `idea` directory's `automated-ml-training.iml` enhances automation within the IDE.

3.6 Evaluation

- Assesses model performance using MSE, calculated in `music_recommendation_retraining.py` and `train_and_deploy.py`.
- `train_and_evaluate_model` task and standalone script compare predictions against test set.
- Logs MSE to MLflow, with `mlruns` storing metrics and hyperparameters (e.g., `n_similar_users`).
- Enables comparison across experiments.
- Evaluates `collaborative_filtering_model.pkl` against `tracks.csv`-derived test data.

3.7 Deployment

- Prepares trained model for production use.
- `app.py`, running in `streamlit-app` service from `docker-compose.yml`, serves API via Streamlit on port `8501`.

- Loads `collaborative_filtering_model.pkl` from `models/` using `dill`, with TF-IDF fallback if model fails.
- `train_and_deploy.py` facilitates model saving.
- `docker-compose.yml` mounts `./models` to `/opt/airflow/models` and `./data` to `/app`.
- Dependencies in `requirements.txt` and virtual environment in `venv` support deployment.

3.8 Maintenance and Retraining

- Involves periodic retraining to adapt to new data, managed by Airflow DAG in `music_recommendation_retraining.py`.
- `train_and_evaluate_model` task, triggered by scheduler, updates model.
- Logs stored in `logs`, with retraining details in `2025-04-08` and `2025-04-09`.
- `idea` directory's `profiles_settings.xml` and `misc.xml` store IDE-specific maintenance settings.
- `pycache_` optimizes performance.
- `train-model` service in `docker-compose.yml` supports manual retraining.

4 Collaborative Filtering for Music Recommendation

4.1 What is Collaborative Filtering?

Collaborative Filtering is a widely-used technique in recommendation systems that relies on the past behavior and preferences of users to suggest items. The key idea is that users who have agreed in the past tend to agree again in the future. Instead of relying on content-based features (such as metadata of songs), collaborative filtering makes recommendations based on interactions between users and items (e.g., user-song ratings or preferences).

4.2 Types of Collaborative Filtering

There are mainly two types of collaborative filtering:

- **User-Based Collaborative Filtering**: Recommends items to a user by finding similar users based on their preferences and recommending items those similar users liked.
- **Item-Based Collaborative Filtering**: Recommends items that are similar to items the user has liked before. Similarity is computed between items, not users.

There is also a more advanced variant known as **Model-Based Collaborative Filtering**, which uses machine learning models (like matrix factorization, SVD, or deep learning) to learn latent user-item interactions.

4.3 Our Approach: User-Based Collaborative Filtering

In this project, we adopted **User-Based Collaborative Filtering**. The rationale behind this choice was to simulate how users with similar taste in music (based on genre interactions) can help each other discover new tracks. We created a *user-genre interaction matrix*, where each row represented a user, and each column corresponded to a genre. The interaction value represented how much a user engaged with that genre, using the `track_pop` (track popularity) value as a proxy.

We then used this matrix to calculate similarities between users and generated recommendations for a user based on the average preferences of the top-N most similar users.

4.4 Similarity Measures Used

To determine user similarity, we explored and applied two commonly used similarity metrics: **Cosine Similarity** and **Pearson Correlation Coefficient**.

4.4.1 Cosine Similarity

Why we use it: Cosine similarity is ideal when we care more about the orientation of user vectors rather than their magnitude. It captures how similar the preference patterns are, regardless of the intensity.

Mathematical Formula:

$$\text{Cosine Similarity}(A, B) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

In our project: We used the `cosine_similarity` function from `sklearn.metrics.pairwise` to compute similarity between a target user and all other users in the user-genre interaction matrix.

4.4.2 Pearson Correlation

Why we use it: Pearson correlation considers the *mean-centered behavior* of users, i.e., how their preferences deviate from their own average. It's especially useful when users have different rating scales.

Mathematical Formula:

$$\text{Pearson}(A, B) = \frac{\sum_{i=1}^{n}(A_i - \bar{A})(B_i - \bar{B})}{\sqrt{\sum_{i=1}^{n}(A_i - \bar{A})^2}\sqrt{\sum_{i=1}^{n}(B_i - \bar{B})^2}}$$

In our project: While our final implementation used cosine similarity, Pearson correlation can be implemented using `numpy.corrcoef` or similar functions. It can be useful in future extensions of the project when real user ratings (rather than synthetic popularity scores) are available.

4.5 How We Used Cosine Similarity in Our Code

In the `CollaborativeFilteringModel` class, the following code snippet demonstrates the use of cosine similarity:

```
similarity = cosine_similarity(
    user_profile.values.reshape(1, -1),
    self.user_genre_matrix
)[0]
```

- Here, `user_profile` is the interaction vector of the target user.
- We reshape it into a row vector and calculate cosine similarity against every row (user) in the `user_genre_matrix`.
- The top-N most similar users are selected using `argsort()`.
- Their average preferences are computed and used to form the recommendation vector.

This method ensures that each user receives recommendations tailored to what similar users liked in terms of genre popularity.

5 Implementation Details

5.1 Model Development

The recommendation system was developed using Python and its machine learning libraries. We used **Pandas** to load and preprocess a dataset of song tracks (e.g., `tracks.csv`), extracting features such as artist genres with a custom parsing function. **NumPy** facilitated numerical operations, including random sampling for user interactions. A user-genre matrix was created, where rows represent users, columns represent genres, and values indicate interaction strength based on track popularity. The recommendation model, a custom **CollaborativeFiltering-Model**, was implemented using **Scikit-learn**'s cosine similarity to identify similar users and generate recommendations by averaging their preferences. The dataset was split into training and test sets using **Scikit-learn**, and model performance was evaluated with mean squared error. All code was versioned using Git to ensure collaboration and reproducibility.

5.2 Docker

Docker is a platform that automates the deployment of applications inside lightweight, portable containers. These containers package an application with its dependencies, ensuring consistency across different environments. In our project, Docker was utilized to containerize key services, such as MLflow for experiment tracking, Apache Airflow for task orchestration, and PostgreSQL for metadata storage.

The main advantages of using Docker in this project were:

- **Environment Consistency**: Ensured consistent performance across various environments.
- **Isolation**: Prevented conflicts by isolating services in separate containers.
- **Scalability**: Simplified scaling by allowing replication of services across environments.
- **Simplified Deployment**: Made deployment more portable by packaging all components into isolated containers.

In our setup, Docker helped containerize MLflow, Apache Airflow, and PostgreSQL, streamlining the deployment and management of these services. Docker Compose was used to define the multi-container environment, specifying service configurations and dependencies to ensure seamless communication.

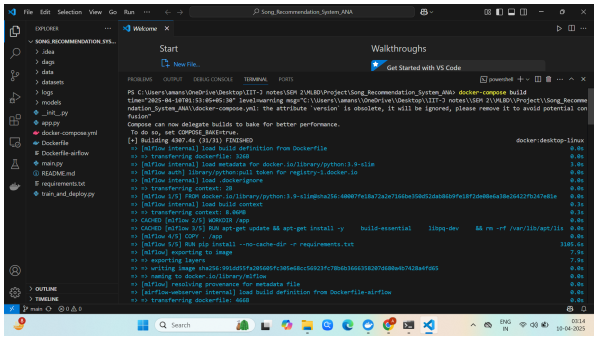
5.2.1 Docker Compose and Configuration

We used a `docker-compose.yml` file to configure and orchestrate the multi-container services, specifying environment variables, port mappings, and dependencies for each service.

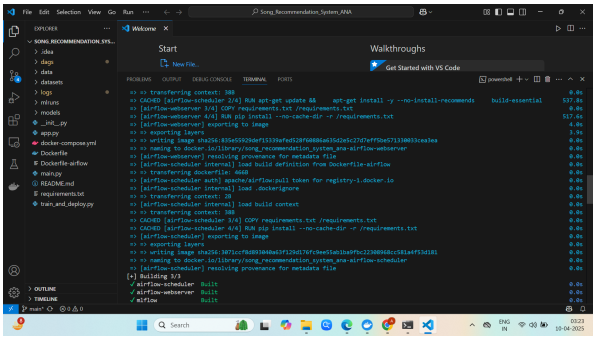
5.2.2 Data Persistence and Volumes

To ensure data persistence, Docker volumes were used for storing experiment tracking data from MLflow and metadata from PostgreSQL, so that no data was lost during container restarts.

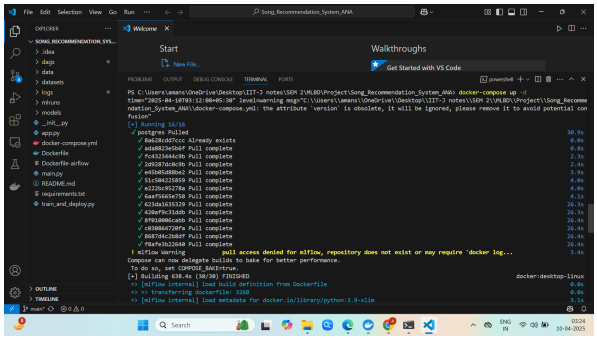
Overall, Docker played a crucial role in ensuring a consistent and scalable environment, facilitating the deployment and management of services across the entire project.



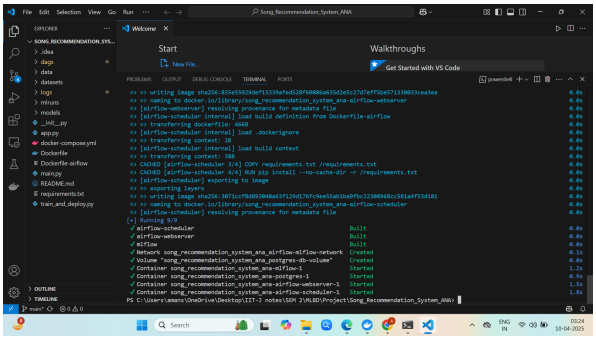
(a) Docker Build



(b) Docker Build Confirmation



(c) Starting Docker Up



(d) Starting Docker Confirmation

5.3 MLflow

MLflow is an open-source platform designed for managing the end-to-end machine learning lifecycle. In our project, MLflow was integrated to streamline and manage various components of the recommendation system pipeline, ensuring better tracking, reproducibility, and model management.

5.3.1 What is MLflow?

MLflow is a machine learning lifecycle management tool that helps in tracking experiments, packaging code into reproducible runs, and sharing and deploying models. It provides a robust environment for managing machine learning models, experiments, and their associated metadata. It offers four main components:

- **Tracking**: Records and queries experiments.
- **Projects**: Packages code in a reusable and reproducible form.
- **Models**: Manages and deploys machine learning models in various formats.
- **Registry**: Centralized model store for managing the lifecycle of models, including versioning and deployment.

5.3.2 Why Use MLflow in Our Project?

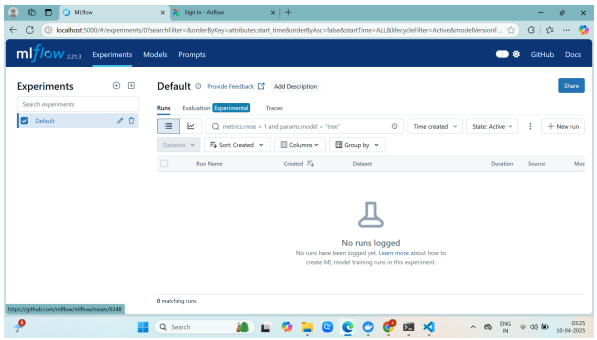
In our project, MLflow was utilized to manage the experiment tracking and model lifecycle. MLflow's ability to log and visualize parameters, metrics, and artifacts provides an efficient way to ensure that each experiment is reproducible and transparent. The key benefits of using MLflow in this project include:

- **Experiment Tracking**: By logging each model training run, its parameters, metrics (such as Mean Squared Error), and other artifacts, we ensured that all experiments were trackable. This enabled easy comparison of models and improved model iteration.
- **Model Versioning**: MLflow allowed us to save and version the models, making it possible to track the progress of the recommendation system over time.
- **Scalability and Reproducibility**: As MLflow centralizes experiment data, it became easier to scale the project, collaborate on model experimentation, and reproduce any model run on different machines.
- **Ease of Integration**: MLflow integrates seamlessly with existing Python code and libraries like `scikit-learn` used in our project, providing flexibility and reducing setup complexity.

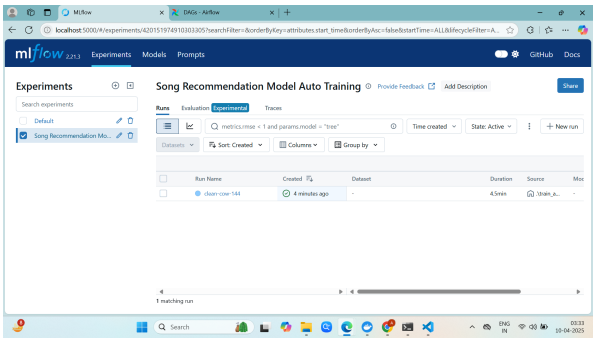
5.3.3 MLflow in the DAG Pipeline

In our Airflow DAG, MLflow was integrated to manage the machine learning lifecycle from data preprocessing to model training and evaluation:

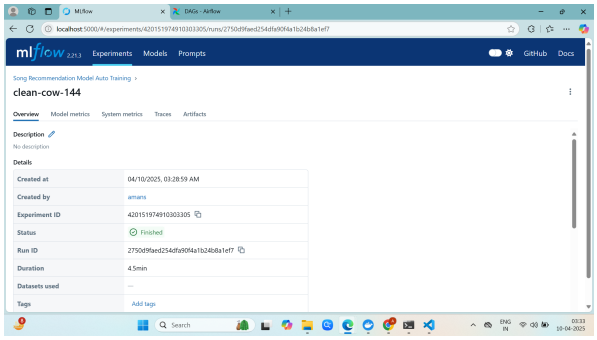
- **Experiment Creation**: The DAG included a task to ensure that the MLflow experiment for the music recommendation system was created if it did not already exist. This helped organize all subsequent experiment runs under the same experiment name.
- **Logging Parameters and Metrics**: During the model training phase, MLflow was used to log important parameters (e.g., number of similar users) and performance metrics (e.g., Mean Squared Error). This ensured transparency and accountability for each experiment.
- **Model Logging**: The trained recommendation model was serialized and logged as an artifact in MLflow, allowing us to track and version it. This made it easier to deploy the best-performing model at any point in the future.
- **Tracking URI Setup**: The MLflow server URI was set to the server running the tracking UI, ensuring that all metrics and models were logged to the centralized server.



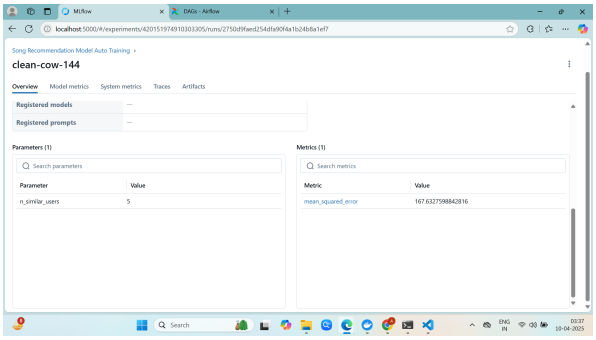
(a) ML Flow Interface



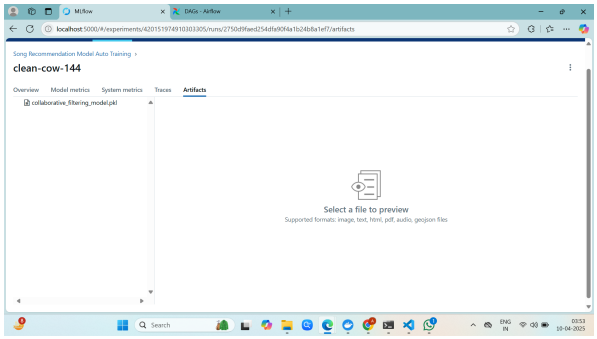
(b) ML Flow Deploy Model



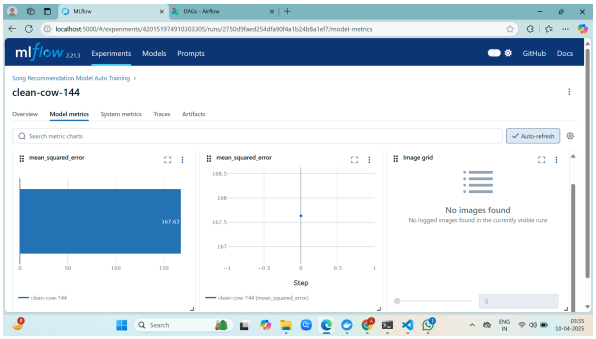
(c) ML Flow Experiment Section



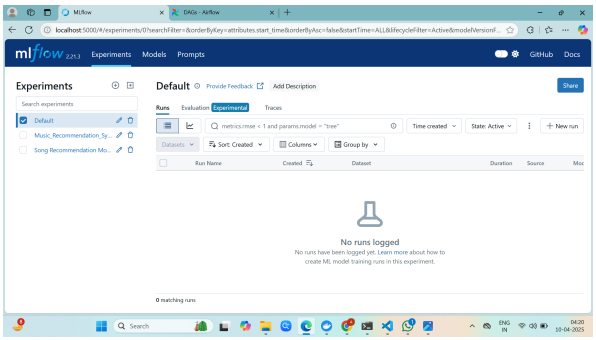
(d) ML Flow Mean Square Error



(e) ML Flow Model



(f) ML Flow Model Metric



(g) ML Flow music recommendation

5.4 Airflow

Airflow is an open-source platform used to programmatically author, schedule, and monitor workflows. It allows the creation of Directed Acyclic Graphs (DAGs) to define workflows as a series of tasks executed in a specific order. Airflow is widely used in data engineering and machine learning projects for automating and orchestrating data workflows.

5.4.1 What is Airflow?

Airflow enables the definition, scheduling, and monitoring of workflows. Workflows are represented as DAGs, where each task is executed according to defined dependencies. Key features of Airflow include:

- **Task Scheduling and Orchestration**: Automates task execution at specified times or intervals.

- **Dynamic Workflow Definition**: Workflows are defined using Python code.
- **Task Dependencies and Execution Order**: Ensures tasks run in the correct sequence based on dependencies.
- **Error Handling and Retries**: Provides retry mechanisms for failed tasks.

5.4.2 Why Use Airflow in Our Project?

Airflow was integrated into our project to automate and manage the machine learning pipeline for the music recommendation system. It helped in:

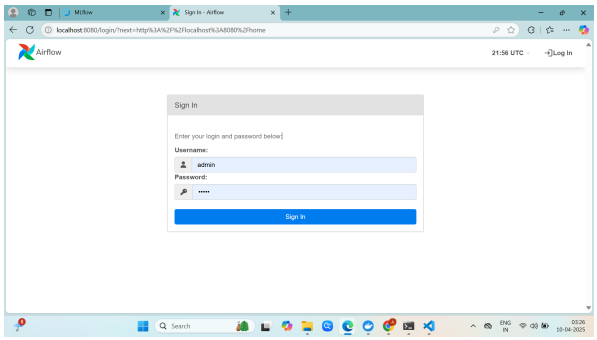
- Automating tasks like data preprocessing, model training, and evaluation.
- Scheduling periodic tasks for model retraining.
- Monitoring task execution and handling errors.

5.4.3 How Airflow Was Used in Our Project

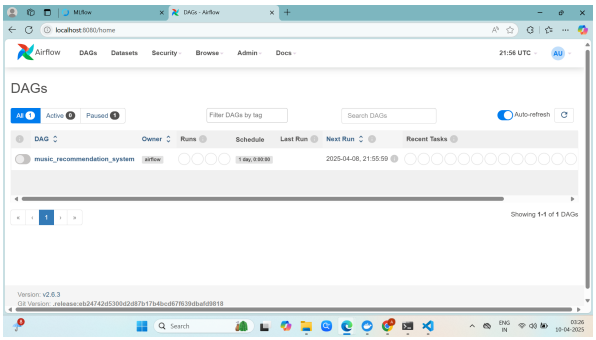
In our project, we used Airflow to orchestrate the entire machine learning pipeline. The pipeline was defined as a DAG with tasks such as:

- **Data Loading and Preprocessing**: Preprocessing the dataset before training.
- **Model Training**: Training the collaborative filtering model.
- **Model Evaluation and Logging**: Evaluating the model and logging results to MLflow.

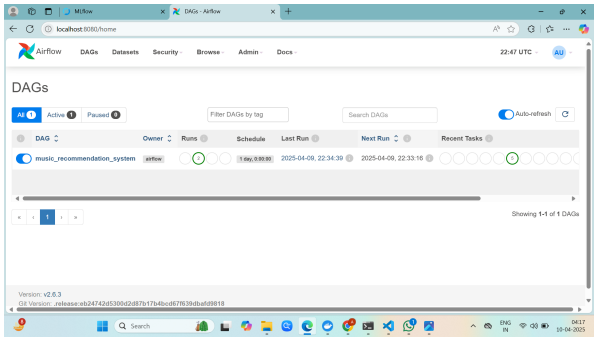
Each task was defined as a PythonOperator, and dependencies ensured that tasks executed in the correct order.



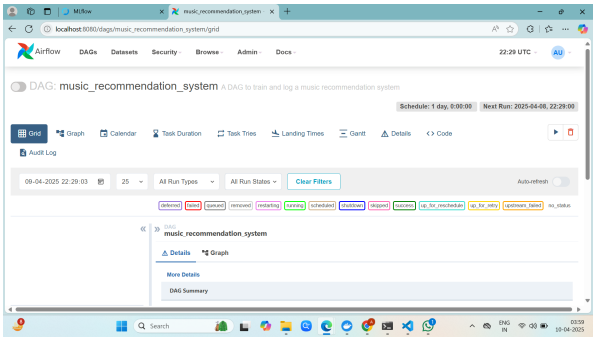
(a) Airflow Admin Page



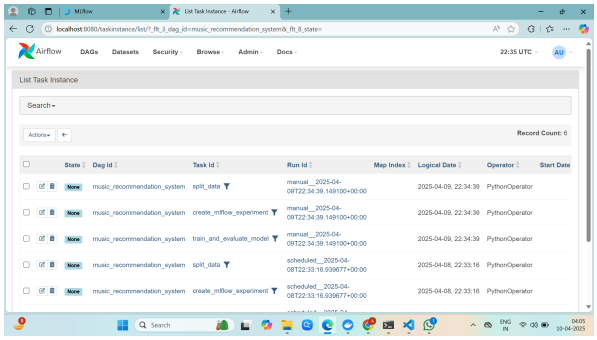
(b) Airflow Admin Login



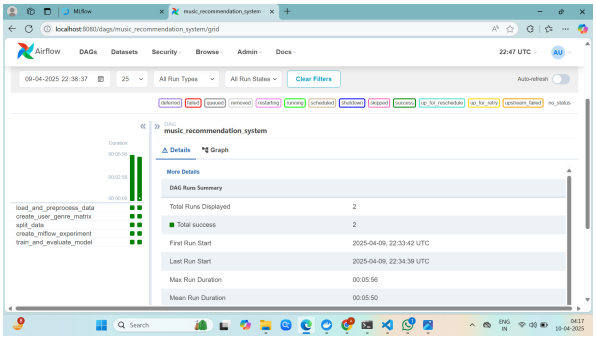
(c) Airflow Music Recommendation System



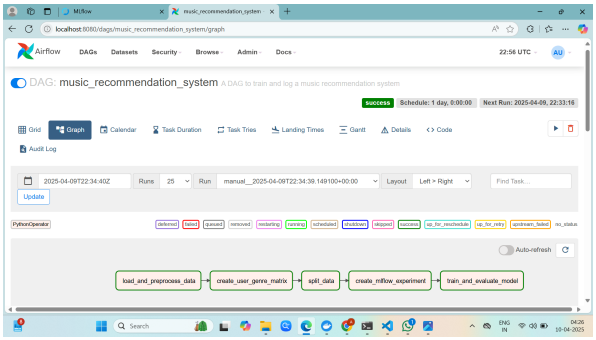
(d) Airflow



(e) Airflow Task



(f) Airflow Task Executed



(g) Airflow Graph

5.5 Streamlit

Streamlit is an open-source app framework that enables the rapid creation of interactive web applications for data science and machine learning projects. It allows users to easily visualize and interact with machine learning models and datasets without the need for extensive web development knowledge.

5.5.1 What is Streamlit?

Streamlit is a Python library that enables the creation of web applications for data science workflows. It simplifies the process of building and deploying apps, allowing data scientists to focus on the functionality rather than the web development details. Key features of Streamlit include:

- **Interactive Widgets**: Easily create interactive UI elements like buttons, sliders, and dropdowns.
- **Real-time Visualization**: Supports real-time updates for visualizations and outputs.
- **Easy Integration with Python**: Works seamlessly with Python libraries like Pandas, NumPy, and Matplotlib.
- **Instant Feedback**: Changes to the app are immediately reflected in the interface.

5.5.2 Why Use Streamlit in Our Project?

In our project, we utilized Streamlit to build an interactive web interface for the music recommendation system. It allowed users to:

- Select preferred music genres and receive personalized song recommendations.

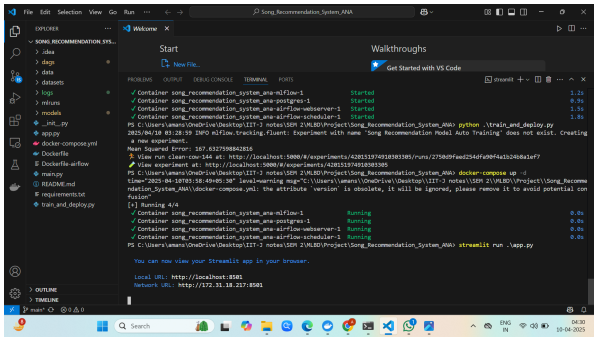
- Interact with the model outputs via a simple and intuitive interface.
- Visualize the top recommended songs and listen to them directly from the platform.

5.5.3 How Streamlit Was Used in Our Project

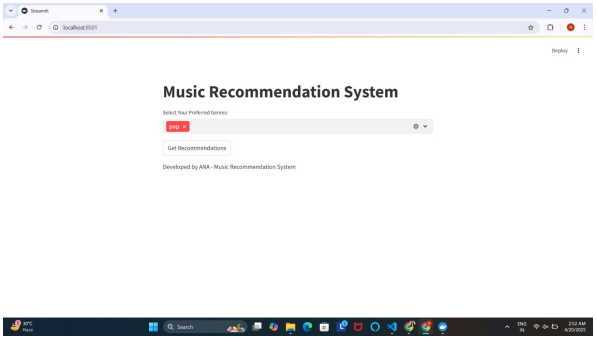
Streamlit was used to create a user-friendly interface that interacts with the underlying machine learning model. The application consists of the following key components:

- **Genre Selection**: Users can select their preferred music genres from a list.
- **Recommendation Generation**: The system uses either a collaborative filtering model or a fallback TF-IDF-based recommender to generate song recommendations based on the selected genres.
- **Song Display**: The top 5 recommended songs are displayed along with their artist names and links to listen to the tracks.

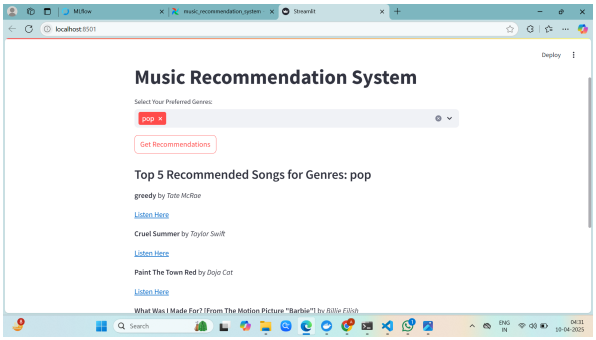
The application also provides real-time feedback and allows users to experiment with different genres to see how recommendations change.



(a) Streamlit Run



(b) WebApp Interface



(c) Recommendation Songs

6 Future Work

The future enhancements for the Music Recommendation System aim to expand its capabilities and improve user experience by incorporating advanced features and alternative recommendation strategies. These improvements will leverage emerging technologies and refined data handling techniques to meet evolving industry demands.

- **Multiple User Data Storage**: Develop a database schema to store data for multiple users, utilizing PostgreSQL or a similar relational database to manage user profiles and listening histories.
- **Real-Time Feedback Integration**: Incorporate real-time user feedback mechanisms, such as likes or skips, to dynamically adjust recommendations and improve model adaptability using online learning techniques.
- **Multimodal Data Analysis**: Extend the dataset to include multimodal inputs, such as audio waveforms or lyrics, analyzed using deep learning models (e.g., CNNs or NLP techniques) to enrich recommendation features.
- **Personalized Playlists Generation**: Develop an algorithm to automatically generate personalized playlists based on user listening habits, mood detection, or contextual data (e.g., time of day), enhancing user engagement.

7 Conclusion

In this project, we designed and implemented a modular and reproducible end-to-end machine learning pipeline for a Music Recommendation System. The pipeline integrates multiple modern tools and technologies to ensure seamless data flow, experiment tracking, orchestration, and model deployment.

We utilized **MLflow** for experiment tracking and model management, allowing us to log parameters, metrics, and artifacts with ease. **Apache Airflow** was employed for orchestrating workflows and automating different stages of the ML lifecycle, such as data preprocessing, model training, and evaluation. **Docker** was used to containerize all services, ensuring consistent environments across development and deployment. **Streamlit** enabled us to build a lightweight and interactive user interface for song recommendations based on genre preferences, using both collaborative filtering and a fallback TF-IDF-based model.

We also incorporated **GitHub** for version control, collaboration, and tracking of code changes. This helped maintain a clean codebase and streamlined development among team members.

By integrating these technologies, the project not only highlights the practical aspects of building a recommendation system but also emphasizes best practices in machine learning engineering — reproducibility, modularity, automation, and user interaction.

References

[1] MLflow Documentation. MLflow is an open-source platform for managing the end-to-end machine learning lifecycle. Available online: https://mlflow.org/docs/latest/index.html.

[2] Apache Airflow Documentation. Apache Airflow is an open-source tool to help track and schedule workflows. Available online: https://airflow.apache.org/docs/apache-airflow/stable/index.html.

[3] Docker Documentation. Docker is a platform designed to help developers build, ship, and run applications in containers. Available online: https://docs.docker.com/.

[4] Streamlit Documentation. Streamlit is an open-source app framework for Machine Learning and Data Science projects. Available online: https://docs.streamlit.io/.

[5] GitHub Documentation. GitHub is a platform for version control and collaboration, allowing multiple people to work on projects simultaneously. Available online: https://docs.github.com/en.

[6] Machine Learning with Python - Full Course by freeCodeCamp. A free YouTube course offering an in-depth guide to machine learning using Python. Available online: https://www.youtube.com.