artistify

Low-Level Design Document

Aadit Mahajan and Anirudh Rao

1 Overview

This document details the low-level design for `artistify`, a FastAPI-based application that takes a storyline, segments it into scenes, and assigns suitable soundtrack songs using semantic analysis and artist matching.

2 API Endpoints

2.1 `GET /get_track_data`

Description: Fetches track metadata from Spotify based on the given track name.

Query Parameters:

- `track_name` (string) – Name of the track to search.

Response (200 OK):

```
{
  "track_data": {
    "name": "Track Title",
    "artist": "Artist Name",
    "album": "Album Name",
    ...
  }
}
```

Response (500):

```
{
  "error": "Error message"
}
```

2.2 POST /generate_soundtrack

Description: Generates a soundtrack by assigning songs to segmented scenes from a storyline.

Request Body (JSON):

```
{
  "storyline": "Once upon a time...",
  "artist": "OptionalArtistName"
}
```

Response (200 OK):

```
{
  "Chosen Artist": "Artist Name",
  "Scene 1": {
    "scene_text": "...",
    "assigned_song": "Song Title",
    "similarity_to_song": 0.873
  },
  ...
}
```

Response (500):

```
{
  "error": "Error message"
}
```

2.3 GET /

Description: Root endpoint. Returns a welcome message.

Response:

```
{
  "message": "Welcome to the Artistify API!"
}
```

3 Data Models and I/O

Request Model

`soundtrack_request`:

- `storyline` (str) – required
- `artist` (str) – optional

Internal Data Structures

- `scenes: List[str]` – Scene segments
- `scene_esa_vectors: List[np.ndarray]` – ESA vectors for scenes
- `tracks_esa_vectors: List[np.ndarray]` – ESA vectors for track lyrics
- `assignments: List[Tuple[int, int]]` – Scene-to-track matchings

4 Core Modules

- **split_into_scenes**: Segments input storyline into coherent scenes.
- **generate_esa_vectors**: Generates TF-IDF ESA vectors from text using Wikipedia corpus.
- **ArtistRecommender.predict()**: Recommends a suitable artist based on semantic similarity to the storyline.
- **get_artist_top_tracks**: Fetches top tracks and lyrics of the artist from Genius.
- **assign_songs_to_scenes**: Matches tracks to scenes based on cosine similarity of ESA vectors.

5 Performance Monitoring

5.1 Prometheus

Prometheus metrics are exposed on port `9090`:

- `request_count`: Tracks number of soundtrack generation requests.
- `scene_split_time`: Measures time taken to segment scenes.
- `esa_vector_generation_time`: Time to generate ESA vectors for scenes.

- `story_esa_vector_time`: Time to compute average vector for full story.
- `artist_recommendation_time`: Time taken to find suitable artist.
- `top_tracks_retrieval_time`: Time to fetch tracks and lyrics.
- `tracks_esa_vector_generation_time`: Time to vectorise track lyrics.
- `song_assignment_time`: Time for matching songs to scenes.

5.2 Grafana Dashboard

A Grafana dashboard is exposed on port 3000 for keeping track of the metrics that are tracked by Prometheus.

6 Error Handling and Logging

- All exceptions are caught and logged using `logging` module.
- Logs are stored in `debug.log`.
- Errors are returned as structured JSON with HTTP 500 status.

7 Sequence of Operations

1. Receive storyline and optional artist.
2. Segment the storyline into scenes.
3. Generate ESA vectors for each scene.
4. If artist not provided, predict best match.
5. Fetch top tracks and lyrics.
6. Generate ESA vectors for lyrics.
7. Compute similarity and assign tracks to scenes.
8. Return scene-song mappings and similarity scores.

8 MLOps Implementation

This section outlines the MLOps practices and tools employed in the development and deployment of the `artistify` API, focusing on the machine learning components.

8.1 Experiment Tracking

- **Tool**: MLflow
- **Purpose**:
  - Tracking of scene splitting experiments to determine optimal parameters.
  - Logging of parameters, metrics, and artifacts related to model training and evaluation.
  - Ensuring reproducibility of experiments by recording code versions, configurations, and data splits.
- **Implementation Details**:
  - The `mlflow_process_storyline.py` script utilizes MLflow to log parameters like `similarity_threshold` and `min_scene_length`, and metrics such as the `dissimilarity` score, for each experiment run.
  - MLflow runs are organized under the experiment name "scene_splitting".
  - Each run is named descriptively (e.g., "thr_0.8_minlen_3") for easy identification of the tested configuration.

8.2 Data Versioning and CI/CD

- **Tool**: DVC (Data Version Control)
- **Purpose**:
  - Managing and versioning the datasets used in the machine learning pipeline (e.g., artist lists, scraped lyrics, ESA vectors).
  - Defining and executing the steps of the machine learning pipeline in a reproducible manner.
  - Tracking dependencies between data, code, and models to ensure pipeline integrity.
- **Pipeline Stages (defined in `dvc.yaml`)**:
  1. `scrape_artists`: Scrapes artist names from Billboard charts.
  2. `fetch_artist_lyrics`: Fetches top tracks and lyrics for scraped artists.
  3. `generate_esa_vectors`: Generates ESA vectors for the fetched lyrics.
- **Artifact Tracking**:
  - DVC tracks the output datasets of each stage (e.g., `scraped_artists.txt`, `scraped_artist_data.csv`, `scraped_esa_vectors_all_lyrics.csv`).
  - Dependencies between stages are explicitly defined, ensuring that changes in one stage trigger necessary updates in downstream stages.

8.3 Model Management

- The trained Artist Recommender model is implicitly managed through the data pipeline.
- The model's behavior and performance are evaluated and tracked through MLflow experiments, allowing for selection of the best-performing model configuration.
- While explicit model versioning isn't implemented within this iteration, MLflow's artifact tracking provides a mechanism to retrieve specific model versions based on experiment runs.

8.4 Distributed ESA Vector Generation with Spark

- **Tool**: Apache Spark
- **Purpose**:
  - Distributed processing of large volumes of lyrics data to generate ESA vectors.
  - Parallelization of the computationally intensive `generate_esa_vectors` function across a cluster.
  - Improved processing speed and scalability for handling a growing corpus of song lyrics.
- **Implementation Details**:
  - A Spark session is initialized with the application name "ESA vector generation".
  - The input data, consisting of artist, track, and lyrics, is converted into an RDD (Resilient Distributed Dataset).
  - The RDD is partitioned based on the available CPU cores to maximize parallelism. The number of partitions is determined dynamically using `multiprocessing.cpu_count()`.
  - The `mapPartitions` transformation is used to apply the `get_lyrics_partition` function to each partition of the RDD. This function generates ESA vectors for the lyrics within each partition.
  - The resulting ESA vectors are collected back to the driver node using the `collect()` action.
  - The Spark session is stopped to release resources.
- **Benefits**:
  - Significantly reduces the processing time for ESA vector generation, especially with large datasets.
  - Enables the system to scale to handle a larger number of artists and songs.
  - Improves resource utilization by distributing the workload across multiple nodes or cores.

9 Software Packaging

The software has been packaged into a set of docker containers, setup using docker compose. The ports to the frontend and the backend APIs have been forwarded to the internet for access through https servers. The web application has been currently served on an NGROK free-tier domain, with both the backend server and the frontend flutter page running in through separate NGROK APIs, each bound to their own containers.

The docker container runs a total of 6 services, namely one container/service for each frontend, backend, ngrok-frontend, ngrok-backend, prometheus, grafana. This allows for monitoring the application effortlessly, while packaging the solution into a reproducible package.

FastAPI has been used to expose the script at specified API endpoints and to build the API for the application.

10 UI Implementation

The UI for this application has been implemented in Flutter. Flutter flow was used to make most of the design choices in the UI. The interface has been kept simple in order to reduce chances of erroring.

There are three screens in the UI, the home screen, loading screen and the results screen.

- Home screen has the input text fields for the user's storyline and artist of choice.
- Loading screen - Buffer screen which has backend API calls to the server.
- Results screen - Displays the results of the scene-song mapping.