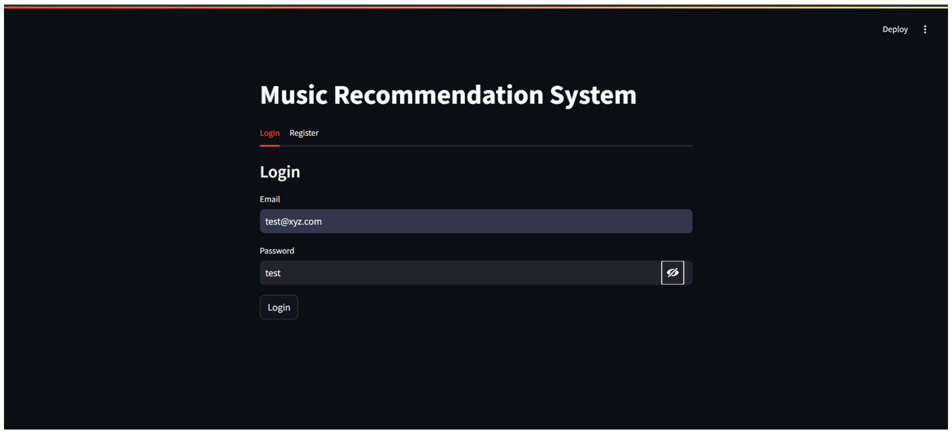
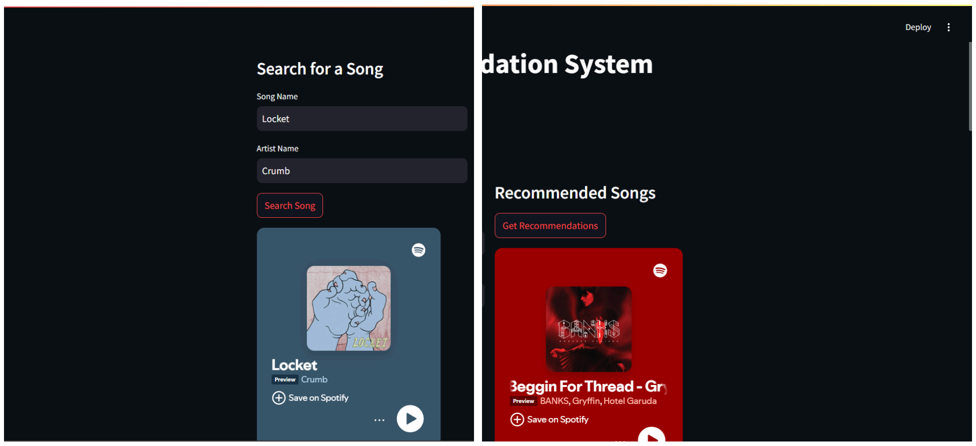
Music Listener Behavior Analysis

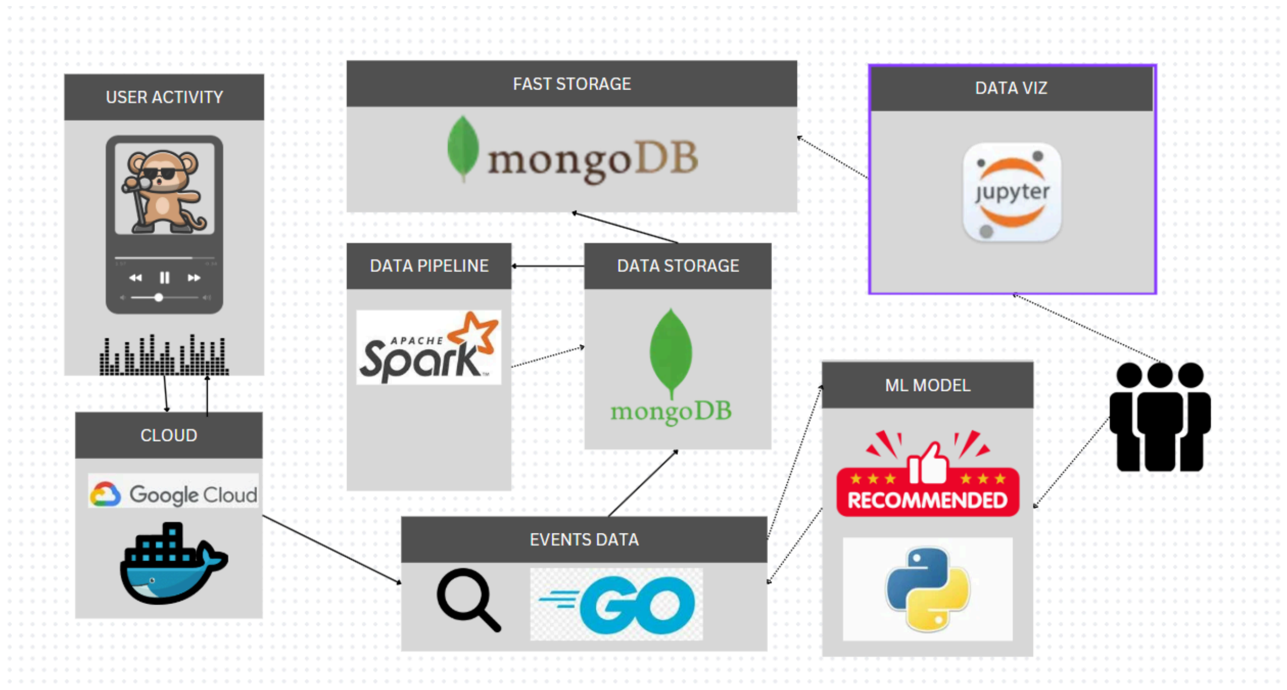
Project Overview & Deliverables:

1. **Github Repo:** https://github.com/legend4137/Statify





Technologies:



Data Engineering

Each layer in the architecture serves a specific function in the pipeline, detailed below:

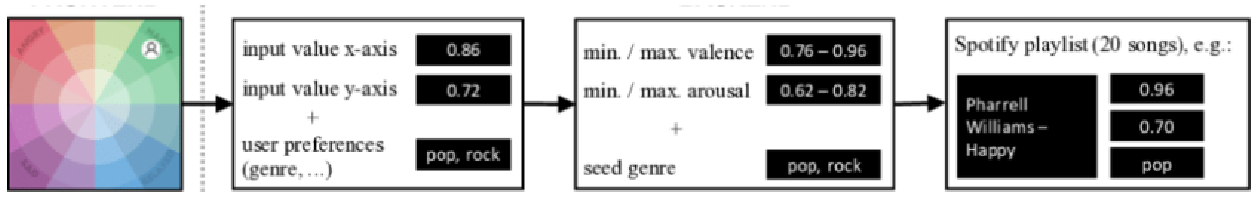
Data Ingestion

- **Sources**:
  - **Spotify API**: Retrieves track attributes (e.g., energy, valence, danceability). [Link]
  - **Kaggle:** Got Spotify Million Dataset for various features. [Link]
  - **Users:** Made 1000 Dummy users with different genres, different tracks listened and very much hybrid choices for better classification.
- **Tools Used**:
  - **Spotify API**: Collects audio features and track metadata.
  - **MongoDB Atlas:** Used MongoDB Atlas for better Data Visualization and Elasticsearch Queries.
- **Docker Integration**:
  - The ingestion scripts are containerized using Docker, ensuring consistent deployments across environments and simplifying replication of the entire pipeline.

Data Processing

- **Data Cleaning**:
    - **Pandas and Python**: Data cleaning functions standardize and cleanse the data by handling null values, removing duplicates, and ensuring reliable feature consistency.
- **Mood Classification**:
    - Assigning mood categories based on Thayer's and Russell's models (e.g., high energy, low valence for 'anger'). [RESEARCH PAPER]

Getting USER_MOOD from the database by applying standard deviation of all tracks' energy and valence values to get his overall mood and use it for RECOMMENDATION.



- **Real-Time Processing with the Speed Layer**:
    - Captures user interactions in real-time, allowing quick adaptation to a user's current mood, and incorporates immediate data into the collaborative filtering model.

Data Storage and Indexing

- **MongoDB**:
    - Stores unstructured and semi-structured data, including lyrics and user interaction logs, providing flexibility for fast updates
- **Elasticsearch**:
    - Indexes track data, user preferences, and mood categories for high-speed retrieval, enabling rapid queries that enhance the responsiveness of recommendations.
- **Apache Spark**:
    - Data
- **Data Versioning**:

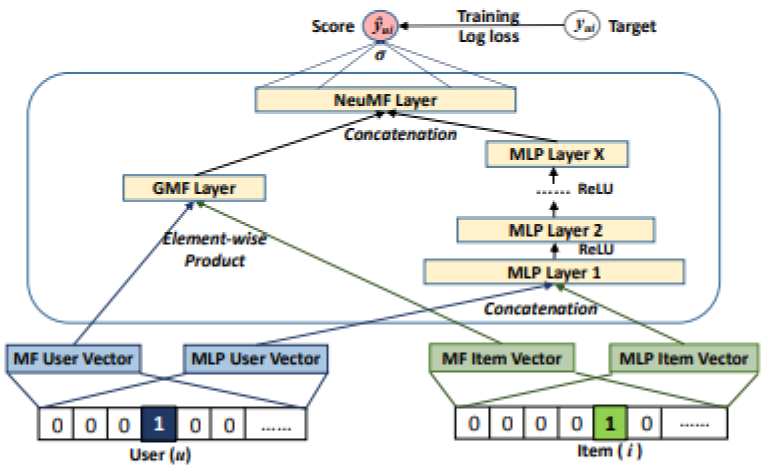
- Data versioning tracks changes in song metadata, user profiles, and collaborative filtering models. Each model iteration or data update is logged to ensure reproducibility across different versions.

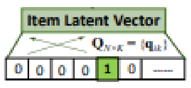
<u>Analyses Results:</u>

MACHINE LEARNING MODELS:

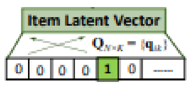
Collaborative Filtering [Hybrid Neural Collaborative Filtering]:

- Processes user-song interactions (e.g., skips, repeats) by generating key-value pairs where the key is the user ID and the value contains interaction metrics for each song.
- Aggregates interaction metrics per user, building clusters of users with similar listening patterns. This facilitates collaborative filtering by identifying songs that users with similar behaviors enjoyed.
- **NCF Approach**: Neural Collaborative Filtering further enhances recommendation accuracy by leveraging these grouped user preferences.
- Used Hybrid Application for the conditions to include more personalized recommendations to users like language, age, gender,
- **Output**: A list of top recommended songs per user cluster, ready for real-time mood-based refinement.

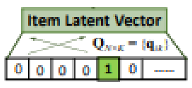




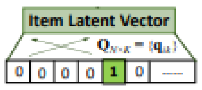
Age



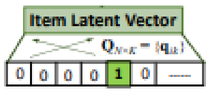
Gender



Language



Genre



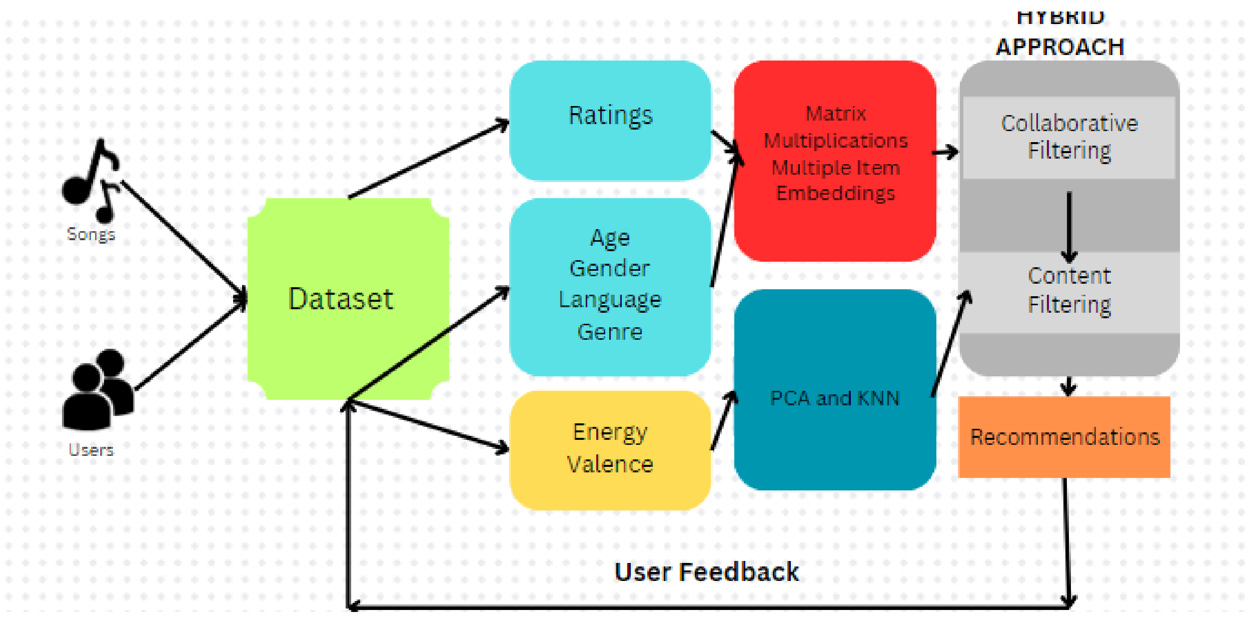
Ratings

Bias in Data:

- **Sampling Bias:**
  - This occurs when certain groups of users or songs are underrepresented or overrepresented in your data. For instance, if your dataset contains more data from users in a specific geographic location, the recommendations may not generalize well to other locations.
- **Implicit Bias:**
  - Implicit bias in recommendation systems comes from relying on implicit feedback, like user clicks or play counts, rather than explicit ratings. Implicit data may reflect more passive preferences rather than active decisions, leading to misinterpretation of users' true preferences.

Recommendation Engine:

- **Recommendation Process**:
  - Processes user-item interactions by pairing each user with their listening history and behavior.
  - Aggregates users with similar listening habits to refine collaborative filtering scores. The Reduce function outputs song clusters tailored to user preferences based on similar listener patterns.
  - **Final Recommendation Generation**:
    - Mood analysis results are combined with collaborative filtering scores, mood classifications, and real-time user interactions. These components enable the system to provide a mood-aligned set of next 5 songs.
- **Views and Materialized Views**:
  - **Views**: Aggregated views of user interaction data (e.g., searches, skipping) provide insights into listening patterns.
  - **Materialized Views**: Frequently queried data, such as mood-based song clusters, is cached to optimize the responsiveness of the recommendation engine.



Fine Tuning:

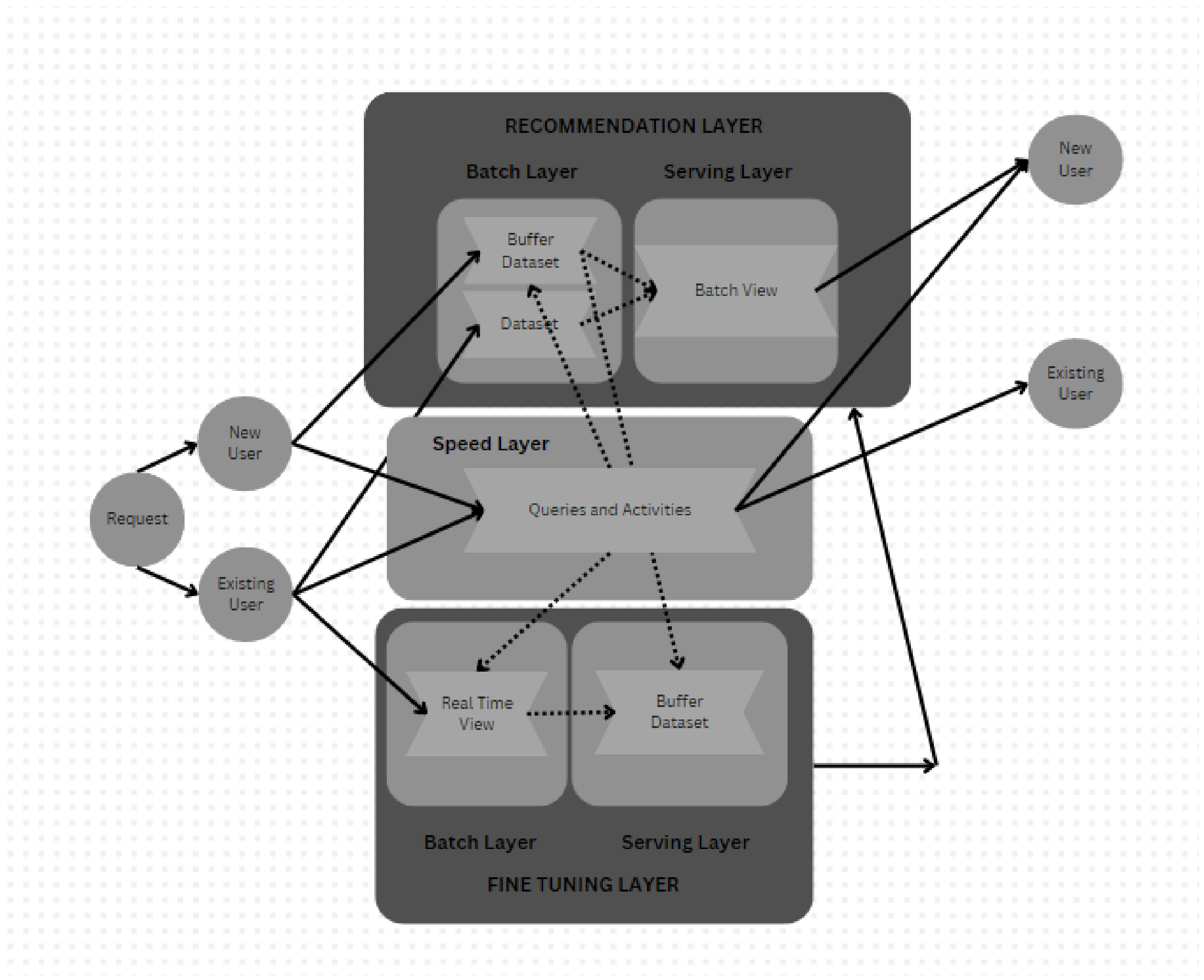
Fine-tuning refers to the process of adjusting the parameters of a machine learning model after its initial training to improve its performance on specific tasks or datasets. In the context of your recommendation system, fine-tuning typically involves:

- **Incorporating New Data:**
    - As new user interactions and song data are added, the model can be fine-tuned by retraining or incrementally updating the model with this fresh data. This helps the model adapt to evolving user preferences and trends.
- **Evaluation Metrics:**
    - During fine-tuning, you would evaluate the model's performance using metrics like precision, recall, or Mean Average Precision (MAP) to see how well the recommendations are aligning with user expectations.
- **Model Adjustments:**
    - If using neural networks (like NCF), you might adjust the architecture by adding layers, changing the activation function, or introducing dropout to prevent overfitting.

    - Additionally, using domain-specific features (like user mood or emotions) during fine-tuning could enhance recommendations.
- **Feedback Loop:**
    - Fine-tuning can also be based on user feedback. For example, if users rate songs or interact more with certain types of content, this feedback can be used to refine the recommendation model and focus on these preferences.

Serving Layer:

- **Pipeline Integration with Batch and Serving Layers**:
    - The batch layer periodically updates collaborative filtering models and sentiment analysis scores. These updates flow into the recommendation pipeline, incorporating new user data and interactions.
    - The serving layer delivers real-time, mood-based recommendations and adapts to changes in user behavior as they interact with the system.
- **Real-Time Recommendations with the Speed Layer**:
    - The speed layer captures instant interactions, updating sentiment and mood states dynamically to refine recommendations that reflect the user’s current listening preferences.



Application of Docker:

- **Containerization of Services**:
  - Each pipeline component is deployed as a Docker container. For example, data ingestion, processing, and the recommendation engine each run in isolated containers.
- **Microservice Management**:
  - Docker enables independent scaling and management of each service. We can deploy the recommendation engine as a separate service, allowing faster updates to model parameters without affecting data ingestion

Application of Elasticsearch:

- **MongoDB Atlas Search Vs Elasticsearch [Blog]:**
  - **Atlas Search** is a Elasticsearch but is built on top of MongoDB by embedding **Apache Lucene Search Index.**
  - Atlas Search is directly integrated into MongoDB, which simplifies managing and maintaining a single data source, reducing the need for separate synchronization processes.
  - Elasticsearch introduces costs related to managing and scaling additional servers and handling potential data transfer between systems.
- **Data Indexing for Fast Retrieval**:
  - Atlas Search indexes song features (e.g., valence, energy) and user preferences, supporting quick lookups and complex querying for recommendations based on real-time user actions.
- **Scalability**:
  - Atlas Search's clustering capabilities allow horizontal scaling, ensuring the system can handle increasing data and traffic loads without compromising performance.

Materialized Views:

- **Materialized Views**:
  - Frequently accessed data, such as mood-based song lists and user recommendation history, are stored in materialized views. These views reduce query times, enhancing the responsiveness of recommendation updates.
- **Justification for Completeness**:
  - The use of views and materialized views allows for a comprehensive representation of both user preferences and song metadata, ensuring the recommendation engine considers both historical data and real-time interactions.

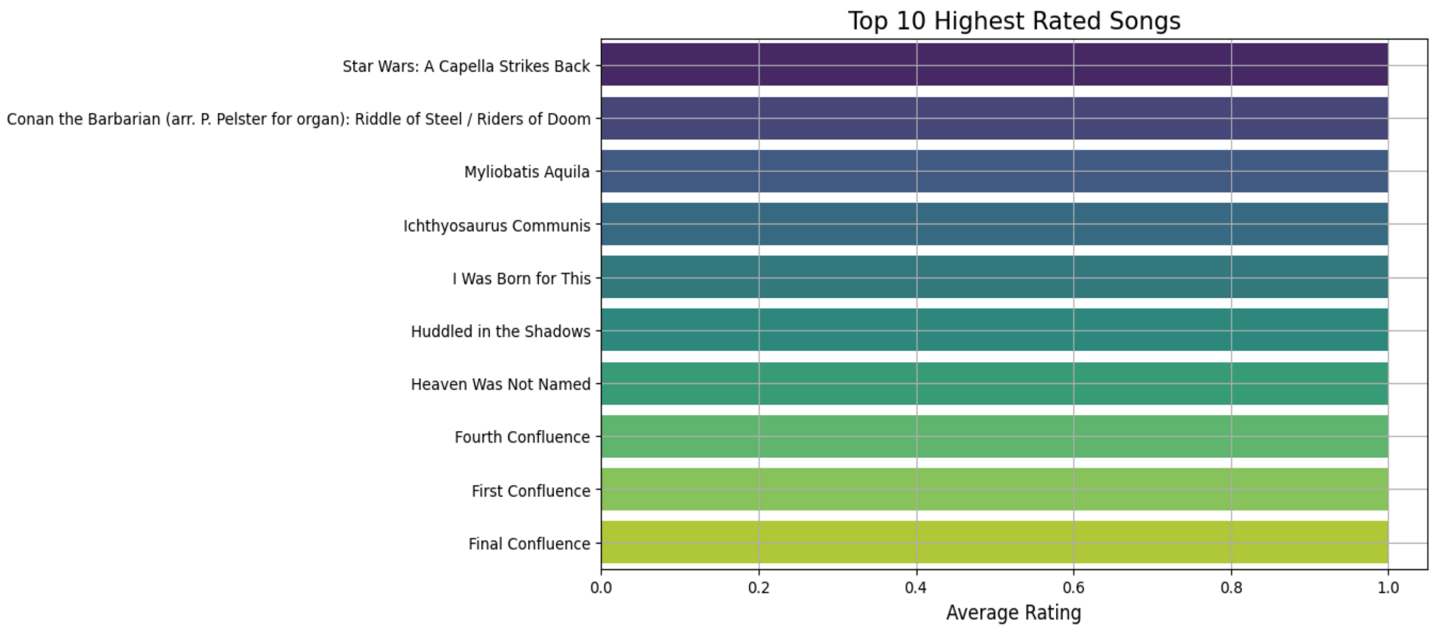
Data Versioning System:

- **Version Control**:
  - Data versioning is crucial for tracking changes in model data and recommendation logic. By using Git for model versioning and implementing dataset versioning with tools like DVC (Data Version Control), we ensure transparency and reproducibility.

Database Choice

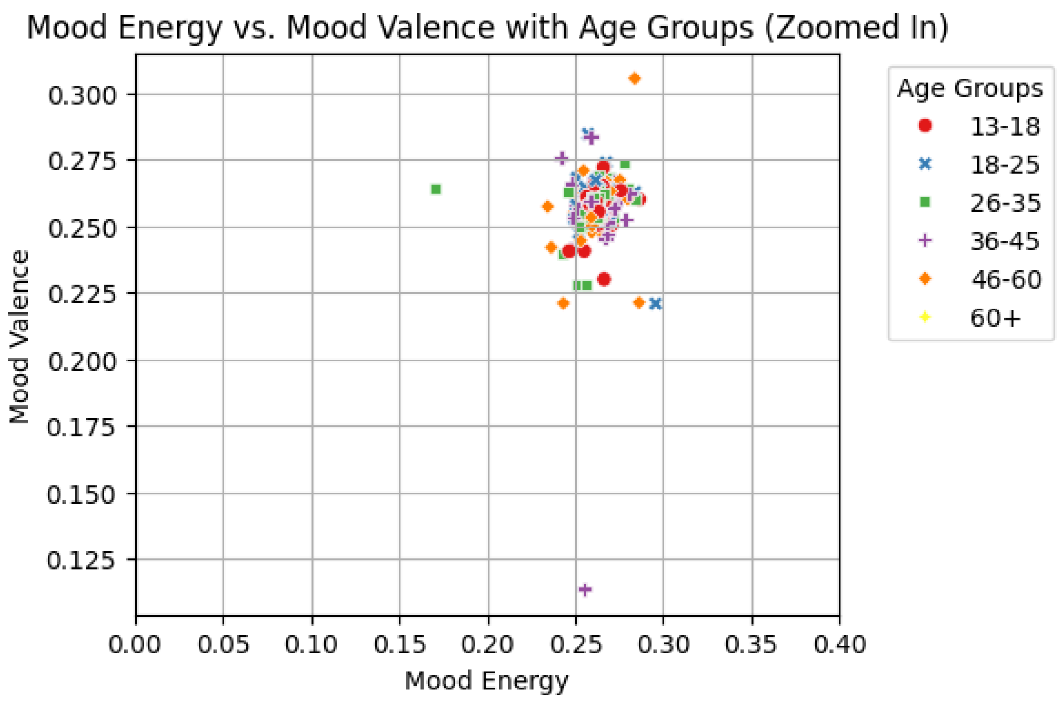
- **MongoDB**:
    - Used for its flexibility in handling unstructured data, MongoDB stores logs, user interactions, and other dynamic information that supports real-time recommendations.

Data Visualization



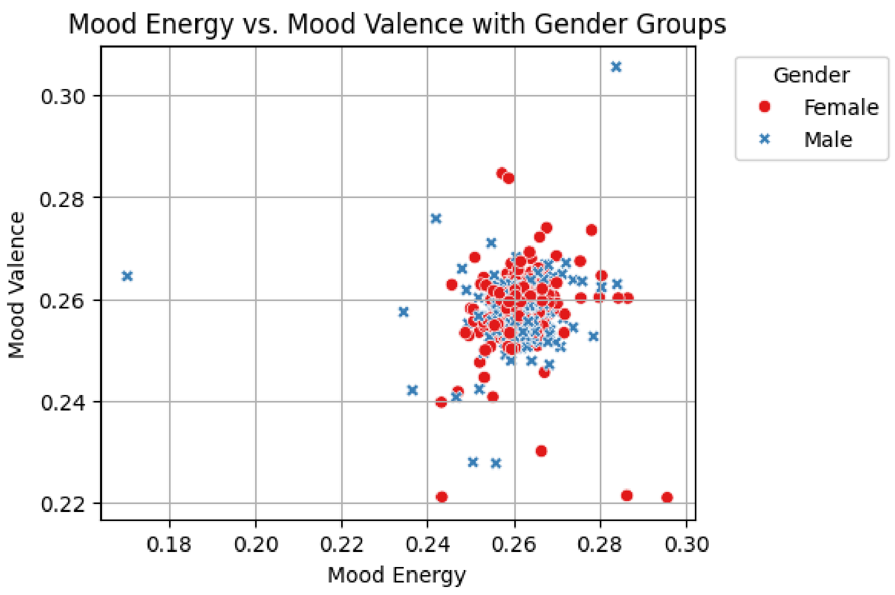
Mood Energy vs. Mood Valence with Age Groups:

- This scatter plot visualizes the relationship between `mood_energy` and `mood_valence` for users, with data points colored by age group across various age ranges, including 13-18, 18-25, 26-35, 36-45, 46-60, and 60+ years. This visualization helps identify how mood energy and valence vary within each age category for users with lower energy levels.



Mood Energy vs. Mood Valence with Gender Groups:

- This scatter plot depicts the relationship between `mood_energy` and `mood_valence` for users, with data points colored and styled according to gender, showcasing any potential gender-specific trends in users' emotional states.



Novel Idea Explored

Content-Based Filtering by Emotions:

- Traditional content-based filtering relies on song attributes like genre, tempo, or artist. Adding emotional context as a feature for songs, such as mood (happy, sad, energetic, etc.), can help refine recommendations.
- You can extract audio features (like energy and valence) to represent the emotional character of a song.
- By considering the user's current emotional state (derived music history), you can filter songs that align with the user's emotional preference at that moment.

How NCF and Emotional Content Filtering Work Together:

- **NCF learns patterns** from the user-song interactions, while **content filtering by emotions** offers more granular control over the emotional tone of the recommendations.
- By adding **emotion as an additional input feature**, you can enrich the user and item embeddings, guiding the NCF model to not only recommend popular or commonly liked songs but also to recommend songs that match the user's emotional context.
- For example, if a user is feeling sad, the model might suggest slower, melancholic songs, even if those songs are not highly rated but match the user's emotional state.

Benefits of Combining NCF and Emotional Content Filtering:

- **Personalization:** The model can offer more personalized recommendations, not only based on user preferences but also taking into account their emotional state, leading to a better user experience.
- **Context-Aware Recommendations:** By integrating emotions, the recommendations become context-aware, meaning they adapt to the user's current mood or life situation.
- **Novelty and Diversity:** This approach helps introduce users to new songs they might not have rated highly but which resonate with their emotions, improving the diversity of recommendations.

Comparison with Existing Technologies:

1. Apache Spark vs. MapReduce

Apache Spark:

- **Advantages:**
  - **Speed and Performance:** Spark is significantly faster than MapReduce, primarily due to its ability to process data in memory, reducing disk I/O operations. This makes Spark highly suited for large-scale data processing tasks like training recommendation models on massive datasets (e.g., music listening history).
  - **Ease of Use:** Spark provides higher-level APIs (in Python, Scala, Java, etc.) compared to MapReduce, making it easier to write and maintain distributed data processing code.
  - **Real-Time Processing:** Unlike MapReduce, which is batch-based, Spark supports both batch and real-time stream processing (e.g., real-time recommendations based on user activity).
  - **Advanced Analytics:** Spark integrates well with machine learning libraries like MLlib, enabling complex data processing and machine learning pipelines for music recommendations.

Why Spark is Better:

- **Recommendation Systems:** Music recommendation systems often need to process large datasets in real-time and perform complex transformations, which Spark is optimized for.
- **Scalability:** Spark's ability to scale across many nodes with ease is important when you have millions of users and tracks, like in a music recommendation system.

2. MongoDB Atlas vs. SQL Databases

MongoDB Atlas:

- **Advantages:**
  - **Schema-less & Flexible Data Model:** MongoDB’s document-based storage allows for a flexible, schema-less design. This is ideal when dealing with diverse data types, such as user activity logs, songs, and metadata, which may not fit well into a rigid SQL schema.

    - **Horizontal Scalability:** MongoDB provides easy horizontal scaling by sharding, which is crucial for handling large volumes of data, like millions of users and songs, in a scalable way.
    - **Aggregation Framework:** MongoDB's aggregation pipeline is powerful for performing complex queries and transformations that are necessary for your recommendation algorithms (e.g., filtering songs based on user preferences, demographics).
    - **JSON-like Documents:** Since MongoDB stores data in a JSON-like format (BSON), it's easier to represent nested data, such as songs with multiple attributes (e.g., genre, artist, mood).

Why MongoDB Atlas is Better:

- **Dynamic Nature of Music Data:** Unlike SQL, MongoDB provides the flexibility to store and evolve complex data schemas, especially with the rapidly changing nature of your data (new songs, user activity, and ratings).
- **Ease of Integration with Elasticsearch:** MongoDB Atlas can seamlessly integrate with **MongoDB Atlas Search**, making it easier to search and analyze large music catalogs for recommendation purposes.

When to Prefer SQL Databases:

- SQL databases are typically preferred when your data is highly structured and relational, such as transactional data (e.g., banking systems). However, for music recommendations where data diversity is high, MongoDB excels.

3. MongoDB Atlas Search vs. Elasticsearch

MongoDB Atlas Search (powered by Elasticsearch):

- **Advantages:**
    - **Integrated with MongoDB:** MongoDB Atlas Search uses Elasticsearch under the hood, but it provides native integration with MongoDB, making it easier to set up and manage. You don't need to maintain a separate Elasticsearch cluster.
    - **Simplified Management:** MongoDB Atlas provides a fully managed platform, so you don't need to handle deployment, scaling, or updates for your search infrastructure.
    - **Optimized for MongoDB Data:** MongoDB Atlas Search is designed to work seamlessly with MongoDB collections, which means you can easily index and search across your music data (e.g., user activity, songs, metadata).

Why MongoDB Atlas Search is Better:

- **Easier to Set Up and Manage:** With MongoDB Atlas Search, you get the power of Elasticsearch without the overhead of managing separate services, making it ideal for a scalable recommendation system that requires real-time search.
- **Efficiency:** MongoDB Atlas Search has been optimized for performance, indexing data in real-time and offering low-latency searches for your application.

When to Prefer Elasticsearch:

- If your data source is not MongoDB, then **Elasticsearch** is a standalone option for full-text search and analytics across a variety of data sources.

4. PyTorch vs. Other Deep Learning Frameworks (e.g., TensorFlow, Keras)

PyTorch:

- **Advantages:**
    - **Dynamic Computation Graphs:** PyTorch uses dynamic computation graphs, which are easier to debug and more flexible for building complex, research-oriented models. This makes it ideal for experimentation in developing custom recommendation algorithms.
    - **Better Debugging:** Due to its "eager execution" model, PyTorch allows you to inspect and debug your model step-by-step, which is crucial when implementing and fine-tuning complex algorithms.
    - **Active Community & Research Focus:** PyTorch is widely used in the research community, and many state-of-the-art models (including for recommendation systems) are implemented in PyTorch.
    - **Support for GPU Acceleration:** PyTorch provides seamless integration with CUDA for fast training on GPUs, allowing you to train large neural networks on large datasets like music interaction history.

Why PyTorch is Better:

- **Customizability:** PyTorch allows for more flexibility when building complex models like **Neural Collaborative Filtering (NCF)**, which is crucial for your recommendation system.
- **Rich Ecosystem:** PyTorch has a comprehensive ecosystem that can assist in building advanced models for content-based filtering and collaborative filtering.

When to Prefer TensorFlow or Keras:

- TensorFlow is highly optimized for production deployment and offers a robust set of tools for model deployment, like TensorFlow Lite, which might be better if you need to deploy your model to mobile devices or edge computing environments.

6. Go vs. JavaScript for Backend Development

Go (Golang):

- **Advantages:**
    - **Performance:** Go is known for its fast performance and minimal memory footprint, which is ideal for building high-performance backend services for recommendation systems.
    - **Concurrency:** Go provides built-in concurrency support via goroutines, which makes it easy to handle multiple concurrent tasks (e.g., serving multiple user recommendations at the same time).
    - **Scalability:** Go is well-suited for building scalable, distributed systems like recommendation engines.
    - **Simple Syntax and Efficiency:** Go's syntax is simple and its compilation speed is fast, which improves developer productivity.

Why Go is Better than JavaScript (Node.js):

- **Performance:** Go is typically faster than JavaScript (Node.js) in terms of raw performance, making it better suited for high-performance backend applications like a recommendation engine.
- **Concurrency:** Go's native concurrency model is more efficient than JavaScript's asynchronous event-loop, making Go a better choice for handling large-scale parallel requests (e.g., serving millions of user requests).

When to Prefer JavaScript (Node.js):

- If you need to build a web server quickly with a large ecosystem of packages and a focus on handling requests, **Node.js** with JavaScript might be more suitable. It's also a good option if you're working in a full-stack JavaScript environment.

7. Hybrid Collaborative Filtering vs. Traditional Collaborative Filtering

Hybrid Collaborative Filtering:

- **Advantages:**
    - **Combines Multiple Approaches:** Hybrid collaborative filtering combines **collaborative filtering** (based on user-item interactions) and **content-based filtering** (based on song attributes like mood or genre). This approach addresses the limitations of pure collaborative filtering (e.g., cold start problem and sparsity).
    - **Improved Personalization:** By leveraging both user behavior and song content, the hybrid model offers a richer, more personalized set of recommendations.

Why Hybrid Collaborative Filtering is Better:

- **Cold Start Problem:** Hybrid models can better handle the cold start problem (new users or items) by using content-based filtering to recommend items even when there isn't enough user interaction data.
- **Diversity:** Hybrid systems can offer a more diverse range of recommendations by incorporating both user preferences and content features.
