Development of a Personalized Music Streaming Interface

University of Lausanne, Advanced Programming
Professor S. Scheidegger with TAs: Anna Smirnova and Maria Pia Lombardo

Victor Regly
victor.regly@unil.ch

30th of August, 2024

Abstract

This project develops a personalized playlist generation system that adapts to user preferences through a feedback-driven recommendation engine. Dynamic user profiles, stored in CSV files, evolve based on listening history and interactions. Using similarity scoring and machine learning, the system recommends songs aligned with user preferences. A feedback loop refines these recommendations over time, improving accuracy and user satisfaction. A Streamlit-based desktop prototype offers an interactive interface for playlist management and recommendation exploration. The results demonstrate the system's effectiveness in delivering personalized music experiences and suggest potential enhancements, such as incorporating advanced machine learning models and supporting various media types.

1 Introduction

The proliferation of digital music platforms has dramatically transformed how listeners interact with music, shifting from static collections to dynamic, personalized experiences. Modern users expect music streaming services to not only provide access to vast libraries but also to intelligently recommend tracks that align with their unique tastes. This demand has led to the development of sophisticated recommendation systems, which aim to enhance user satisfaction by delivering curated playlists tailored to individual preferences.

Despite the advancements in recommendation technologies, challenges remain in creating systems that can dynamically adapt to evolving user preferences. Traditional recommendation models often rely on static user profiles or require extensive historical data to make accurate predictions. These approaches can be limiting, especially for new users or those whose tastes change over time. To address these challenges, this project explores the development of a personalized playlist generation system that continuously adapts to user interactions through a feedback-driven recommendation engine.

The primary objective of this project is to design and implement a system that captures user preferences, generates tailored playlists, and refines its recommendations based on ongoing user feedback. The system utilizes dynamic user profiles stored in CSV files, allowing it to track and update user preferences in real-time. By leveraging machine learning techniques, such as similarity scoring, the system identifies songs that closely match the user's current tastes and adapts to their evolving preferences. This report details the methodology used to build the recommendation system, the algorithms that drive the recommendation process, and the design of a user-friendly interface developed using Streamlit. Additionally, the report discusses the challenges encountered during implementation, the solutions applied, and the results obtained from testing the system with various user profiles. Finally, potential improvements and future directions for the system are explored, emphasizing the importance of adaptability in modern music recommendation systems.

2 Description of the Research Question and Relevant Literature

A. Research Question

This project investigates the question: *How can a personalized playlist generation system effectively adapt to individual user preferences in real-time using dynamic profiles and feedback loops?* The focus is on developing a system that recommends music based on a user's current preferences, which are continuously updated based on ongoing interactions.

B. Relevant Literature

In music recommendation systems, Content-Based Filtering plays a crucial role, especially when the goal is to tailor recommendations to individual users based on the characteristics of songs they have previously enjoyed. Unlike collaborative fil-

tering, which relies on similarities between users, content-based filtering recommends music by analyzing specific attributes of tracks, such as tempo, danceability, and energy, and matching them to the user's known preferences.

Recent advancements in recommendation systems emphasize the importance of feedback loops, where real-time user interactions—such as liking or skipping a song—immediately influence the system's future recommendations. This approach ensures that the system remains responsive to the user's evolving tastes, providing more accurate and satisfying suggestions.

The data for this project is sourced from a Spotify dataset, available on Kaggle, which includes detailed audio features. These features enable the system to perform fine-grained content-based filtering, making recommendations that closely align with the user's preferences. By integrating a feedback mechanism, the system continually refines its recommendations, creating a personalized and adaptive listening experience.

3 Methodology/Algorithm

3.1 System Design

The personalized playlist generation system is designed around three core components: dynamic user profiles, a content-based recommendation engine, and a feedback loop that refines the recommendation process. The system tracks user preferences through interactions with the playlist, such as adding or removing songs, and continuously updates the recommendations based on these preferences.

3.2 Algorithm Details

The recommendation engine is built using a content-based filtering approach, which relies on the features of individual songs to make recommendations. The algorithm can be broken down into the following steps:

3.2.1 Feature Representation

Each song in the dataset is represented by a vector of numerical features. Let x_i be the feature vector for the i-th song, where:

$$x_i = [f_1, f_2, \ldots, f_n]$$

Here, $f_1, f_2, \ldots, f_n$ are the features associated with the song, such as danceability, energy, valence, tempo, acousticness, and others. The features are extracted from the dataset originally sourced from Spotify's API.

3.2.2 Scaling the Features

To ensure that all features contribute equally to the recommendation process, the feature vectors are standardized using z-score normalization. For each feature f_j, the standardized feature z_j is calculated as:

$$z_j = \frac{f_j - \mu_j}{\sigma_j}$$

where:

- μ_j is the mean of feature f_j across all songs in the dataset.
- σ_j is the standard deviation of feature f_j.

After standardization, each song is represented by a standardized feature vector z_i.

3.2.3 User Profile Representation

A user's playlist is represented as a set of song feature vectors. The system computes the centroid c_u of the user's playlist to represent the user's overall taste in music. The centroid c_u for user u is calculated as the average of the feature vectors of the songs in the user's playlist:

$$c_u = \frac{1}{m} \sum_{i=1}^{m} z_i$$

where:

- m is the number of songs in the user's playlist.
- z_i is the standardized feature vector of the i-th song in the playlist.

3.2.4 Song Recommendation Using Nearest Neighbors

To recommend new songs, the algorithm computes the Euclidean distance between the user's centroid c_u and the feature vectors of all songs in the dataset that are not currently in the user's playlist. The Euclidean distance d between the user's centroid c_u and a song z_j is given by:

$$d(c_u, z_j) = \sqrt{\sum_{k=1}^{n} (c_u[k] - z_j[k])^2}$$

The algorithm then ranks the songs based on their distance from the user's centroid. The top k songs with the smallest distances are recommended to the user. This process can be mathematically expressed as:

$$\text{Recommend } \arg\min_{j \notin \text{playlist}} d(c_u, z_j)$$

where $\arg\min$ identifies the indices of the songs with the smallest distances from the user's centroid.

3.2.5 Feedback Loop

The system incorporates user feedback to refine future recommendations. When a user adds or removes a song from their playlist, the user's centroid c_u is recalculated to reflect this change. This immediate update ensures that the system remains responsive to the user's evolving preferences.

3.3 Technology Stack

- **Python:** The primary programming language used for implementing the recommendation engine.
- **Pandas:** Used for data manipulation and processing of the song dataset and user profiles.
- **Scikit-learn:** Provides tools for standardizing features and implementing the nearest neighbors algorithm.
- **Streamlit:** Powers the user interface, allowing users to interact with the system by managing playlists and exploring recommendations in real-time.

4 Discussion of the Implementation

4.1 Code Structure

The implementation of the personalized playlist generation system is organized into several key modules, each responsible for different aspects of the application. The main components include:

- `app.py`: The entry point of the Streamlit application, responsible for managing navigation between different pages (e.g., login, registration, and the main playlist interface).
- `pages/` Directory: Contains the individual Streamlit pages:
  - `accueil.py`: Handles user login functionality, verifying credentials and loading user-specific playlists.
  - `register.py`: Manages user registration, including input validation and storing new user credentials.
  - `welcome.py`: The main page after login, where users can view and manage their playlists, receive recommendations, and explore new music.
- `utils/` Directory: Houses utility modules that encapsulate core functionality:
  - `user_management.py`: Manages user authentication, including checking credentials during login and adding new users during registration.
  - `playlist_management.py`: Handles playlist operations, such as loading, adding, and removing songs from a user's playlist, and retrieving song details from the dataset.

```
AP_capstone/
│
├── data/
│   ├── dataset1.csv          # primary dataset with song features
│   ├── users.csv             # dataset for storing user credentials
│   └── playlists.csv         # dataset for storing user playlists
│
├── myenv/                    # virtual environment folder
│
├── pages/                    # Folder for additional pages
│   ├── welcome.py            # page for the welcome screen
│   ├── accueil.py            #  page for the login screen
│   └── register.py           #  page for user registration
│
├── utils/                    # Utility functions and modules
│   ├── playlist_management.py # managing playlists
│   └── user_management.py    # managing users
│
├── app.py                    # Main entry point
│
├── requirements.txt          # List of Python packages
│
│
└── README.md                 # explaining the project
```

Figure 1: Programming structure.

4.2 User Interface

The user interface is built using Streamlit, providing a simple and intuitive way for users to interact with the system. Key aspects of the interface include:

- **Login and Registration**: Users can log in or create a new account via simple input forms. Input validation ensures that usernames and passwords meet basic criteria (e.g., no spaces in usernames, non-empty passwords).

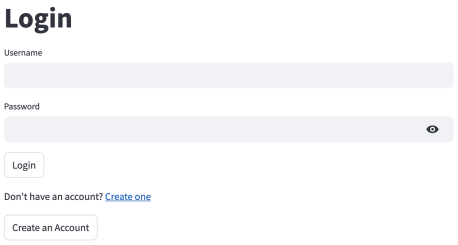

Figure 2: Login

- **Playlist Management**: Once logged in, users are presented with their current playlist. They can search for new songs, add them to their playlist, or remove existing ones. The interface is designed to be responsive, updating in real-time as the user interacts with it.

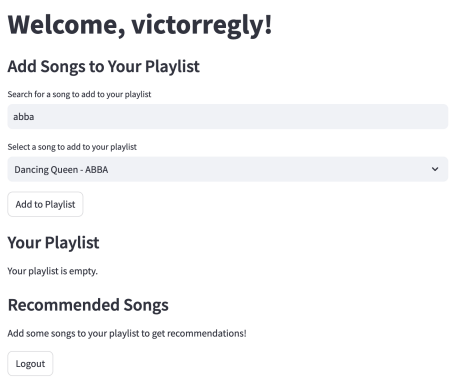

Figure 3: Welcome user page

- **Search Bar**: The search bar was initially designed to find music based on the keywords entered. It has since been enhanced with a popularity filter, making the search results more relevant and meaningful.

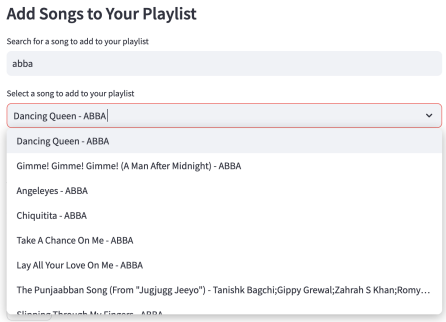

Figure 4: Filter based on popularity

- **Recommendations**: The system displays song recommendations based on the user's current playlist. Users can easily add recommended songs to their playlist, further refining their profile.

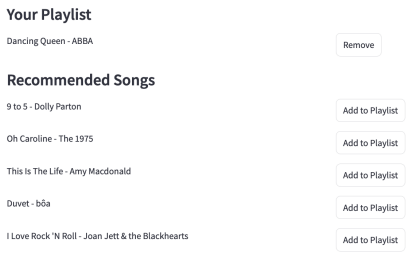

Figure 5: Recommendation system

- **Plot**: The system displays a plot showing the average values for the following features of the songs in the playlist: danceability, energy, valence, tempo, acousticness, popularity, loudness, and speechiness.

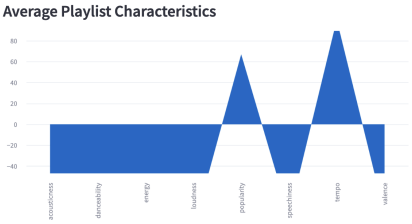

Figure 6: Plotting

4.3 Handling of User Data

User data, including usernames, passwords, and playlists, is stored in CSV files. The system ensures that:

- **User Authentication**: Passwords are written with the username and stored during the login process. It doesn't have any other protection.
- **Playlist Storage**: Each user's playlist is stored in a CSV file, with one row per song and columns for the username and song ID. This allows for easy loading, modification, and saving of playlists.
- **Data Integrity**: Input validation checks are in place to ensure that usernames and passwords are valid and that duplicate entries are not created in the playlist.

4.4 Challenges and Solutions

During the implementation, several challenges were encountered and addressed:

- **Scalability of Data Storage**: Initially, storing user data and playlists in CSV files was straightforward but raised concerns about scalability. As the number of users and songs grows, file-based storage could become inefficient. To address this, the system was designed with the potential for easy migration to a more robust database solution in the future, such as SQLite or PostgreSQL.
- **Real-Time Responsiveness**: Ensuring that the interface remains responsive as users interact with their playlists was another challenge. Streamlit's reactivity, combined with efficient data handling in the backend, ensures that the system updates in real-time without noticeable delays.
- **Balancing Recommendation Accuracy with Speed**: The nearest neighbors algorithm, while effective, can be computationally intensive as the dataset grows. To maintain a balance between recommendation accuracy

and computational efficiency, the system uses a scaled-down set of features for the initial implementation. Future enhancements could include more sophisticated optimizations or parallel processing to handle larger datasets.

- **Security Considerations**: Protecting user data, particularly passwords, should be a critical concern. Implementing password hashing with `bcrypt` should ensure that even if the user data file were compromised, passwords would remain secure. Additionally, basic input validation was implemented to prevent common security issues such as SQL injection, though the use of CSV files minimized this risk.
- **User Experience Design**: Creating an intuitive and engaging user interface was essential for user adoption. Streamlit's simplicity allowed for rapid prototyping, but ensuring the interface was both functional and aesthetically pleasing required iterative design and testing.

4.5 Future Enhancements

Several potential improvements were identified during the implementation:

- **Database Integration**: Migrating from CSV files to a relational database would improve scalability, data integrity, and security, particularly as the number of users and amount of data grows.
- **Advanced Recommendation Algorithms**: Implementing more sophisticated algorithms, such as matrix factorization or deep learning-based approaches, could improve the accuracy and personalization of recommendations.
- **Enhanced Feedback Mechanisms**: Expanding the feedback loop to include more nuanced interactions, such as rating songs or creating multiple playlists, would allow the system to better capture user preferences.

5 Code Maintenance and Updates

5.1 Version Control

The development of this project was managed using Git, a distributed version control system that allows for efficient tracking of changes, collaboration, and code management. The codebase is hosted on GitHub, where each significant change is documented through commits. The following practices were employed to ensure effective version control:

- **Branching Strategy**: A branching strategy was implemented where the main branch holds the stable, production-ready code, and feature branches are used for developing new features or bug fixes. This strategy helps in isolating new developments from the stable codebase and makes it easier to manage releases.
- **Commit Messages**: Descriptive commit messages were used to document the purpose and scope of changes. This practice aids in tracking the evolution of the project and provides context for future developers.
- **Pull Requests**: For significant updates or new features, pull requests were used. This process involves reviewing the changes before merging them into the main branch, ensuring that the new code is thoroughly vetted.

5.2 Code Documentation

Comprehensive documentation is essential for maintaining and updating the codebase. The following documentation practices were employed:

- **Docstrings**: Each function and class in the codebase is accompanied by a docstring that describes its purpose, parameters, and return values. This helps in understanding the functionality of each component at a glance.
- **Inline Comments**: Where necessary, inline comments were added to explain complex or non-obvious parts of the code. This makes the code more accessible to other developers or to yourself when revisiting the code after some time.
- **README.md**: The project repository includes a `README.md` file, which provides an overview of the project, installation instructions, a guide to the file structure, and usage instructions. This document serves as the entry point for new developers or users who want to understand and work with the project.

5.3 Unit Testing

To ensure the reliability and stability of the code, basic unit tests were implemented for critical functions, particularly in the `user_management.py` and `playlist_management.py` modules. The tests cover:

- **User Authentication**: Tests ensure that the `check_user` function correctly validates credentials and that the `add_user` function properly handles new user registrations, including error cases such as duplicate usernames.
- **Playlist Operations**: Tests verify that the playlist management functions (adding, removing songs) behave as expected, including

edge cases like attempting to remove a song that isn't in the playlist.

These tests were automated using Python's `unittest` framework, which allows for the tests to be run regularly to catch any regressions or unintended side effects of new changes.

5.4 Future Maintenance

Given the modular structure of the codebase, future maintenance should be straightforward. Here are some guidelines and suggestions for maintaining and updating the system:

- **Code Refactoring**: As the codebase grows, periodic refactoring may be necessary to maintain readability and efficiency. This could involve reorganizing code into more cohesive modules, optimizing algorithms, or removing deprecated code.
- **Adding New Features**: When adding new features, it is recommended to follow the existing project structure and maintain consistency in naming conventions, code style, and documentation practices. This ensures that the codebase remains uniform and easy to navigate.
- **Updating Dependencies**: The project's dependencies are listed in the `requirements.txt` file. As libraries evolve, it may be necessary to update these dependencies to newer versions. Care should be taken to test the code thoroughly after any updates to ensure compatibility and stability.
- **Security Updates**: As the project deals with user data, it is critical to stay informed about any security vulnerabilities in the libraries or frameworks used. Regularly reviewing and applying security patches is recommended.
- **Database Migration**: If the project scales beyond what is manageable with CSV files, migrating to a more robust database solution (e.g., SQLite, PostgreSQL) is advisable. This will involve updating the data access layers of the code to interface with the new database system.
- **Continuous Integration/Continuous Deployment (CI/CD)**: Implementing a CI/CD pipeline could automate the testing, building, and deployment processes, ensuring that every change is thoroughly tested and that the latest version of the project is always ready for deployment.

5.5 Handling of Issues and Bugs

To manage bugs and issues, a systematic approach was adopted:

- **Issue Tracking**: GitHub's issue tracker was used to document and manage bugs, feature requests, and other tasks. Each issue is labeled and prioritized, making it easier to track progress and address critical bugs first.
- **Bug Fixing**: When fixing bugs, the root cause is identified and documented to prevent similar issues in the future. After the fix is implemented, related unit tests are run to ensure that the fix does not introduce new issues.

6 Results

6.1 Performance Testing

To assess the efficiency of the personalized playlist generation system, a series of performance tests were conducted. These tests focused on measuring the response time, memory usage, and scalability of the system as it generated song recommendations for users with varying playlist sizes. The tests were performed on playlists of different sizes, ranging from 10 to 1,000 songs, and the system was evaluated based on the time taken to generate recommendations and the amount of memory consumed during the process.

6.1.1 Response Time

The response time was measured as the time taken by the system to generate a list of five recommended songs for a user, based on their current playlist. The results, as shown in Table 1, indicate that the response time increases with the size of the user's playlist. For a small playlist of 10 songs, the system generated recommendations in approximately 0.124 seconds. As the playlist size increased to 1,000 songs, the response time increased to around 0.524 seconds. This gradual increase in response time is expected, as larger playlists require more computational resources to calculate the centroid and compare it with other songs in the dataset.

Table 1: Response Time for Different Playlist Sizes

Playlist Size	Response Time (s)
10	0.124
50	0.127
100	0.137
500	0.254
1000	0.524

6.1.2 Memory Usage

Memory usage was tracked during the recommendation process to evaluate how efficiently the system manages its resources. The peak memory usage remained relatively consistent across all playlist sizes, hovering around 24 MB. This stability suggests that the system's memory usage is primarily determined by the base operations required to handle the dataset and user interactions, rather than the size of the user's playlist. Even with the largest playlist size tested (1,000 songs), the system maintained a memory usage of approximately 24.3 MB, indicating that the system is well-optimized for handling different playlist sizes without significant increases in memory consumption.

Table 2: Memory Usage for Different Playlist Sizes

Playlist Size	Memory Usage (MB)
10	24.88
50	24.12
100	24.13
500	24.20
1000	24.30

6.1.3 Scalability

The scalability of the system was evaluated by observing how well it performed as the playlist size increased. The tests show that the system scales reasonably well, with only a modest increase in response time as the playlist size grows. The fact that memory usage remained stable across different playlist sizes further supports the system's scalability, suggesting that it can handle increasing user data without requiring proportional increases in computational resources.

6.1.4 Number of Recommendations

The system was configured to generate a fixed number of recommendations (five) for each test, regardless of the playlist size. The consistency in the number of recommendations across all tests indicates that the system can maintain its output quality even as the complexity of the input data increases.

Table 3: Number of Recommendations for Different Playlist Sizes

Playlist Size	Nb.Recommendations
10	5
50	5
100	5
500	5
1000	5

6.2 Summary of Findings

The performance tests demonstrate that the personalized playlist generation system is both efficient and scalable. The response time remains within a reasonable range, even for larger playlists, and memory usage is consistent, indicating that the system can handle a growing user base or dataset without significant performance degradation. These results suggest that the system is well-suited for real-time applications where users expect quick and accurate recommendations.

Moving forward, the system could benefit from further optimization to reduce response times for very large playlists or datasets. Additionally, the implementation of more advanced recommendation algorithms could improve the quality of the recommendations while maintaining the system's efficiency.

6.3 User Feedback

- I had the opportunity to share the system with a few friends for testing. While the user interface is quite user-friendly and functions well, I've received feedback suggesting that there is still room for improvement in the recommendation system. Since the recommendations are based primarily on musical features, it can sometimes result in mismatches, such as having an R&B track from Chris Brown followed by a Japanese R&B song. Although they may be similar in musicality, they differ significantly in style.

7 Conclusion

The development and implementation of a personalized playlist generation system, as explored in this project, demonstrate the effectiveness of content-based filtering combined with real-time feedback to create a dynamic and adaptive music recommendation experience. By leveraging a dataset rich in audio features, the system is able to generate recommendations that align closely with individual user preferences and adapt as those preferences evolve. The results of the performance testing indicate that the system is both efficient and scalable. Response times remain within acceptable limits even as playlist sizes increase, and memory usage remains stable across different test scenarios. This suggests that the system is well-suited for real-time applications, capable of handling a growing user base without significant degradation in performance.

The feedback-driven approach ensures that the system remains responsive to user interactions, allowing for continuous refinement of recommenda-

tions. This adaptability is a key strength, distinguishing the system from more static models that may struggle to keep pace with changing user tastes.

However, there are areas where the system could be further improved. Implementing more advanced recommendation algorithms, such as those based on deep learning or hybrid approaches, could enhance the accuracy and personalization of recommendations. Additionally, migrating from a file-based storage system to a more robust database solution would improve scalability and data integrity as the system grows.

Future work could also explore expanding the system's capabilities beyond music to include other types of media, such as podcasts or video content. By broadening the scope of recommendations, the system could provide a more comprehensive and engaging user experience.

In conclusion, this project successfully demonstrates the potential of a feedback-driven, content-based recommendation system to deliver personalized music experiences. The system's adaptability and efficiency make it a promising foundation for further development and application in various recommendation contexts.

8 Appendix

8.1 Helper Tools

During the development of this project, several external tools and resources were utilized to enhance productivity and ensure the quality of the code. These tools played a significant role in various aspects of the project, from code generation to debugging and documentation. Below is a list of the key tools used and a brief description of their contributions to the development process.

- **ChatGPT**: This AI-powered tool was used to assist in generating ideas, drafting code snippets, and providing explanations for complex programming concepts. ChatGPT was particularly useful in brainstorming solutions to coding challenges and refining the project's approach to specific problems. It also helped in drafting sections of the project documentation, ensuring clarity and comprehensiveness.

8.2 Code Snippets

Below are selected code snippets from the project that illustrate key functionalities in the personalized playlist generation system.

8.2.1 Feature Scaling and Centroid Calculation

The following code snippet demonstrates how the system scales song features and calculates the centroid of a user's playlist, which represents the user's overall music taste.

```
from sklearn.preprocessing import StandardScaler

def calculate_centroid(playlist_features):
    scaler = StandardScaler()
    scaled_features = scaler.fit_transform(
        playlist_features)
    centroid = scaled_features.mean(axis=0).reshape
        (1, -1)
    return centroid
```

8.2.2 Recommendation Algorithm Using Nearest Neighbors

This snippet shows the core recommendation algorithm, which uses the K-Nearest Neighbors method to find songs that are most similar to the user's current playlist centroid.

```
from sklearn.neighbors import NearestNeighbors

def recommend_songs(user_playlist, music_df,
     num_recommendations=5):
    if not user_playlist:
        return []

    scaler = StandardScaler()
    scaled_features = scaler.fit_transform(music_df[
        numerical_features])

    playlist_song_ids = [song['song_id'] for song in
         user_playlist]
    available_songs_df = music_df[~music_df['song_id
        '].isin(playlist_song_ids)]

    if available_songs_df.empty:
        return []

    playlist_indices = music_df.index[music_df['
        song_id'].isin(playlist_song_ids)].tolist()
    playlist_features = scaled_features[
        playlist_indices]
    centroid = playlist_features.mean(axis=0).
        reshape(1, -1)

    nbrs = NearestNeighbors(n_neighbors=
        num_recommendations + len(playlist_song_ids
        ),
                            algorithm='auto').fit(
                                scaled_features)
    distances, indices = nbrs.kneighbors(centroid)

    recommended_indices = [index for index in
        indices[0]
                           if music_df.iloc[index]['
                               song_id'] not in
                               playlist_song_ids]
    recommended_tracks = music_df.iloc[
        recommended_indices][['song_id', 'artist',
        'song_name']].head(num_recommendations)

    return recommended_tracks.to_dict('records')
```

8.2.3 User Authentication

This snippet highlights the user authentication process, where the system checks a user's credentials against stored, hashed passwords.

```
from bcrypt import checkpw

def check_user(username, password):
    if os.path.exists(USER_DATA_FILE):
        users_df = pd.read_csv(USER_DATA_FILE)
        user_row = users_df[users_df['username'] ==
             username]
        if not user_row.empty:
            stored_password = user_row['password'].
                 values[0]
            return checkpw(password.encode('utf-8'),
                 stored_password.encode('utf-8'))
    return False
```

8.2.4 Memory and Performance Monitoring

The code snippet below demonstrates how the system monitors performance metrics such as response time and memory usage during the recommendation process.

```
import time
import tracemalloc

def measure_performance(user_playlist, music_df):
    tracemalloc.start()
    start_time = time.time()

    recommendations = recommend_songs(user_playlist,
          music_df)

    end_time = time.time()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    response_time = end_time - start_time
    memory_usage = peak / 10**6 # Convert to MB

    return response_time, memory_usage,
         recommendations
```

8.3 Bibliography

Below are citations of key articles referenced in the context of music recommender systems:

- Schedl, M., Zamani, H., Chen, C. W., et al. (2018). Current challenges and visions in music recommender systems research. *International Journal of Multimedia Information Retrieval*, 7(2), 95–116.
- Schedl, M. (2019). Deep Learning in Music Recommendation Systems. *Frontiers in Applied Mathematics and Statistics*, 5:44. doi: 10.3389/fams.2019.00044.
- S.Scheidegger, class of Advanced Programming
- S.Scheidegger, class of Advanced Data Analysis