This documentation describes the Delta Melody Match application, built with Streamlit, which recommends songs by analyzing a dataset of user preferences and song details. The steps are described below.

1. **Importing Libraries**

The necessary libraries are imported to provide the tools and functions needed for tasks like data handling, building the interface, and generating recommendations.

```
import streamlit as st
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.decomposition import TruncatedSVD
```

Step 1. Load the Dataset

The load_data() function reads the dataset containing user listening records and song play counts. Caching ensures the data is loaded a single time to optimize performance.

```
# Cache data loading
@st.cache_data
def load_data():
    df = pd.read_csv('song_dataset.csv')
    return df
```

2. Setting Up the Streamlit Interface

We configure the page layout and add custom CSS to style the interface.

```
st.set_page_config(layout="wide")

st.markdown("""
<style>
.small-select {
    max-width: 200px !important;
}
.stButton > button {
    background-color: pink;
    color: black !important;
    border: none;
}
.stButton > button:hover {
    background-color: pink !important;
    color: white !important;
    border: 2px solid white !important;
}
.stMultiSelect [data-baseweb="tag"] {
    background-color: pink !important;
    color: black !important;
}
.stMultiSelect [data-baseweb="select"] > div:first-child,
.stSelectbox [data-baseweb="select"] > div:first-child {
    border-color: white !important;
    box-shadow: 0 0 0 1px white !important;
}
</style>
""", unsafe_allow_html=True)
```

Step 2: Collaborative Filtering

This method recommends songs by analyzing the listening preferences of users with similar tastes. The process begins by creating a User-Item Matrix that tracks how many times each user has played each song.

1. **Create a User-Item Matrix**: This matrix shows how many times each user played each song.

```
def compute_matrices(df_songsDB):
    user_item_matrix = df_songsDB.pivot_table(index='user', columns='song', values='play_count', fill_value=0)
```

2. **Apply Truncated SVD**. This helps the system understand which users have similar tastes. For example, if two users both listened to “The Cove” and “Better Together”, the matrix would apply Truncated SVD to simplify the dataset and find patterns in user behavior to allow the system to recommend similar songs to them.

```
svd = TruncatedSVD(n_components=20, random_state=20)
svd_matrix = svd.fit_transform(user_item_matrix)
item_factors = svd.components_
return user_item_matrix, svd_matrix, item_factors
```

Step 3. Content-Based Filtering. This approach recommends songs based on their details, such as artist name, release, and title. In the code, these details are combined into a single feature for each song.

```
def compute_tfidf(df_songsDB):
    df_songsDB['combined_features'] = (
        df_songsDB['artist_name'] + " " +
        df_songsDB['release'] + " " +
        df_songsDB['title']
    )
```

We then used TF-IDF (Term Frequency-Inverse Document Frequency) to turn song details into numbers so the system can compare them. This helps the system calculates the similarity between these values to find songs that are similar to the ones the user has selected. If a user likes “The Cove” by Jack Johnson, the content-based filtering method can suggest other songs by Jack Johnson or songs from similar albums.

```
tfidf = TfidfVectorizer()
tfidf_matrix = tfidf.fit_transform(df_songsDB['combined_features'])
return tfidf, tfidf_matrix
```

Step 4. Hybrid Recommendation System

We improved the recommendations by merging collaborative filtering and content-based filtering, using a weighting factor called alpha. This hybrid method benefits from the strengths of both techniques. The system generates a score for each song by analyzing the user’s listening history (collaborative filtering) and the song’s attributes (content-based filtering). The final recommendation list is sorted by these combined scores.

For example, if the collaborative filtering method gives a high score to “Banana Pancakes” and the content-based filtering method finds it similar to “The Cove”, the system will rank “Banana Pancakes” higher in the final recommendations.

```
def hybridRecommendationEngine(user_id, selected_songs):
    alpha = 0.5

    listened_songs = df_songsDB[df_songsDB['user'] == user_id]['song'].unique()
    all_songs = df_songsDB['song'].unique()
    unlistened_songs = set(all_songs) - set(listened_songs)

    cf_scores = collaborative_score_calculator(user_id, unlistened_songs)
    content_scores = content_score_calculator(selected_songs, unlistened_songs)

    final_scores = {}
    for song_id in unlistened_songs:
        cf_score = cf_scores.get(song_id, 0)
        content_score = content_scores.get(song_id, 0)
        final_scores[song_id] = alpha * cf_score + (1 - alpha) * content_score

    scores = list(final_scores.values())
    min_score = min(scores) if scores else 0
    max_score = max(scores) if scores else 1

    if max_score > min_score:
        normalized_scores = {
            song_id: (score - min_score) / (max_score - min_score)
            for song_id, score in final_scores.items()
        }
    else:
        normalized_scores = {song_id: 0.5 for song_id in final_scores}

    sorted_songs = sorted(normalized_scores.items(), key=lambda x: x[1], reverse=True)
    recommended_song_ids = [song_id for song_id, _ in sorted_songs[:10]]

    recommended_songs = (
        pd.DataFrame(recommended_song_ids, columns=['song'])
        .merge(df_songsDB[['song', 'title', 'release', 'artist_name']].drop_duplicates(), on='song', how='left')
        .assign(recommendation=lambda x: x['title'] + ' by ' + x['artist_name'])
    )
    return recommended_songs['recommendation'].tolist()
```

Step 4. Streamlit Interface

The Streamlit interface makes it easy for users to interact with the recommendation engine. First, the user selects their unique ID from a dropdown menu. Then, they choose songs they have listened to from another dropdown list. When the user clicks the “Get Recommendations” button, the system generates a list of ten recommended songs. Each recommendation displays the song title and the artist’s name. For example, If a user listens to “The Cove” by Jack Johnson, the system might suggest “Better Together” by the same artist or other songs liked by people with similar music tastes.

```
# Streamlit app
st.title("Delta Melody Match 🎶")

# Make columns take more width
col1, col2 = st.columns([2, 4])

with col1:
    with st.container():
        user_id = st.selectbox(
            "👤 Select User ID",
            options=df_songsDB['user'].unique().tolist(),
            key="small_select"
        )
        st.markdown('<style>div[data-testid="stSelectbox"] > div:first-child {max-width: 200px;}</style>', unsafe_allow_html=True)

songs_selectable = df_songsDB[df_songsDB['user'] == user_id]['title'].unique()

with col2:
    song_titles = st.multiselect(
        "🎵 Select Songs You Like",
        options=songs_selectable,
        default=songs_selectable[:1]
    )

# Make the recommendations table wider
if st.button("Get Recommendations"):
    st.subheader("Recommended Songs")
    recommendations = hybridRecommendationEngine(user_id, song_titles)
    for i, rec in enumerate(recommendations, 1):
        # Split the recommendation into title and artist
        title, artist = rec.split(' by ')
        st.write(f"{i}. ***{title}*** by {artist}")
```

Conclusion

The song recommendation engine is an easy-to-use platform that gives personalized song suggestions based on what users have listened to before. Built with Streamlit, the system allows users to easily select their user ID and multiple songs they have listened to using drop-down menus. By combining collaborative filtering and content-based filtering, the engine ensures that recommendations are both relevant to the user's behavior and similar in characteristics to their chosen songs. The platform has a working URL, and a demo will be provided with the report to ensure a smooth and easy user experience. This project shows a good understanding of recommendation techniques and handles real-world challenges in building and deploying an online system. The approach uses smart model selection, efficient data processing, and a design that focuses on the user. Overall, the engine gives accurate and useful recommendations in a way that is simple and effective.