Music Recommender QA Deliverables - Backend

- Test Strategy/Approach

Testing will be conducted on back-end functionality with only black box knowledge (testing that is performed with no knowledge of a system's internals). The strategy is to test its external specifications and user requirements. This will ensure that the system will produce the expected output based on decision-based scenarios.

The approach will include a lot of repetition testing. Such as domain testing, equivalence partitioning, boundary value analysis, and scenario testing. Domain testing is testing for robustness by inputting valid and invalid values, making sure the system handles edge cases. Equivalence partitioning is to group similar inputs together and test values that represent each group to ensure consistent behavior. Boundary testing is the test of the upper and lower limits of input parameters. Scenario testing is what it sounds like, mimicking real world scenarios. Verifying the music recommender works as intended by providing accurate songs based off criteria.

- Test Plan

Since we need to implement the black box strategy, the test plan will be revolved around it.

Unit Tests:

- Validate that the recommender system processes various inputs correctly and produces expected outputs.
- Verify the functionality of data structures used (e.g., linked lists, arrays).
- Test the system's behavior at the upper and lower limits of acceptable input ranges.
- Validate different execution paths within the functions of the system.
- Ensure the system handles errors gracefully and provides appropriate error messages.

Integration Testing:

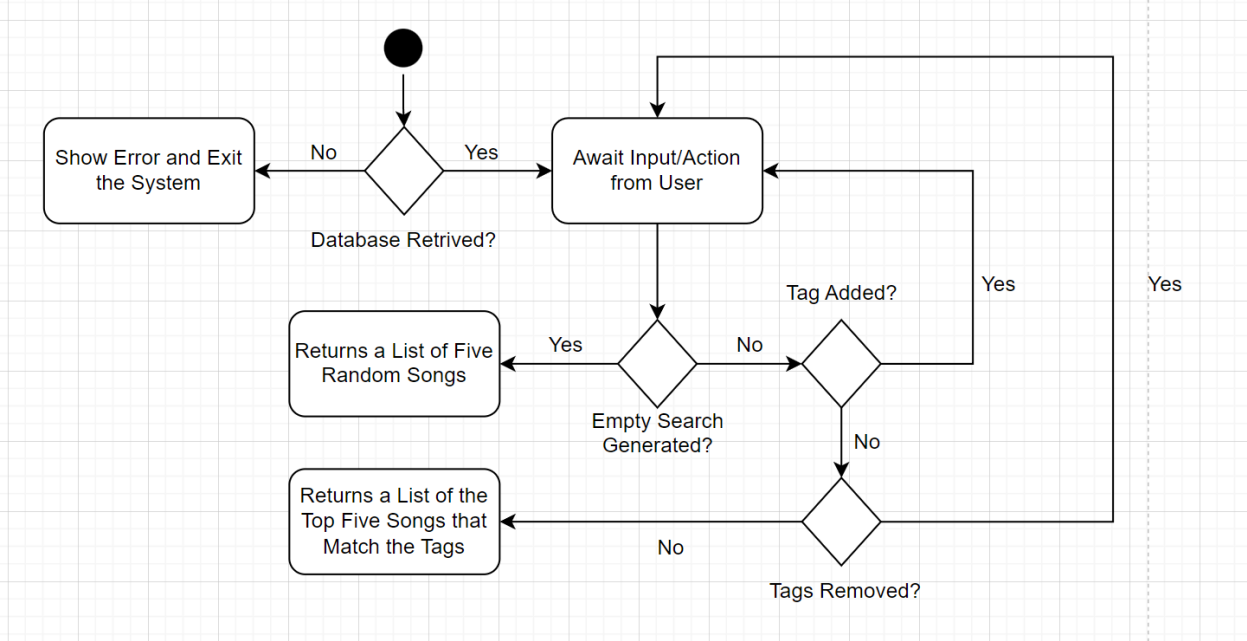
- Manual regression testing via Azure.
- Perform initial testing to ensure basic functionality after integration.

Validation Tests:

- Demonstrate that the recommender system functions according to specified requirements.
- External testing involves a sample of end-users to collect feedback.
- Evaluate the system's ability to recover from failures or crashes.
- Apply stress to the system to evaluate its robustness under extreme conditions.
- Evaluate the system's performance under normal and peak loads.

Project: Music Recommender
Backend QA Deliverables

Test Case Design – Scenario Based Testing



- Traceability Matrix

Requirement Identifiers	Req 1.1	Req 1.2	Req 1.3	Req 1.4	Req 2.1	Req 2.2	Req 2.3	Req 3.1	Req 3.2	Req 3.3	Req 3.4
1	X	X									
1.1		X	X	X							
2		X			X	X					
2.1		X					X		X	X	X
3		X						X	X	X	X

1.1: Connect to Frontend API

1.2: Retrieve Preprocessed Database via backend

1.3: Extract information from database based on tags

1.4: Send recommended songs to frontend

2.1: All tags take in valid inputs and give error upon invalid inputs

2.2: All tags can be combined with one another seamlessly

2.3: Tags give error handling

3.1: Repetitive search checks

3.2: Two or more unlikely tags give a relevant song

3.3: Random song/closest song is represented to characters that are unlikely to be a song

3.4: Random song recommended if no tags are presented

Project: Music Recommender
Backend QA Deliverables

- Test Cases and Results Summary

Connecting to the frontend and sending songs: The frontend will submit a function call to the backend, and the backend will have a return statement for the frontend. With this, the system is able to successfully communicate between frontend and backend.

```
# Return top recommendations
    top_recommendations = df.iloc[related_indices[:5]]
    result = top_recommendations.to_dict(orient='records')

    return result
```

Preprocessed data retrieval and extraction: The code is shown to retrieve and extract the data from preprocessed Spotify data. The only reason this will not work is if the file path for the database file is not in the same directory, otherwise, this is successful.

```
file_path = 'preprocessed_data.csv'

# Load the CSV file into a DataFrame
df = pd.read_csv(file_path)
# df = df.sample(frac=1)  # frac=1 means to use all rows, and the rows will be in
a random order

# app = Flask(__name__)

# Load the pre-trained TfidfVectorizer during initialization
tfidf_vectorizer = joblib.load('tfidf_vectorizer.pkl')

# Load the pre-computed TF-IDF matrix
tfidf_matrix = joblib.load('tfidf_matrix.pkl')
```

Valid and invalid inputs: To test, the values for the json input (tags) were changed to follow certain criteria.

```
json_input = '''
{
    "artists_genres": [],
    "artists_names": [],
    "release_date": "2020",
    "danceability": "null",
    "energy": "null",
    "loudness": "null",
    "speechiness": "null",
    "acousticness": "null",
    "instrumentalness": "null",
    "liveness": "null",
```

```
    "valence": "null",
    "tempo": "null"
}
'''
```

Every single tag was tested with valid inputs and returned valid outputs. To test invalid inputs, special characters were used (^, %, #, *,), (, !, ? \, /, etc). However, only strings characters were accepted so special characters did throw errors and stopped the system from outputting a valid input. It was also tested to see if removing “null” does anything to the valid outputs. It was found that in terms of backend, it does not do anything to the search. However, it was decided to add a value so there is no mixed or overlapping values.

Combining tags: Having two or more tags together prioritized the ones with the most recent addition to the database. For example, when combining Bruno Mars and fast valence, it will prioritize Bruno Mars songs that are fast. If two input tags were Bruno Mars and Olivia Rodrigo, it will prioritize and return the artist with more recent songs. Since both artists have made a song together, the system will prioritize Olivia Rodrigo because she is more recent in the database.

Now the combination makes a difference as there is precedence. From the rigorous regressive testing, it was found the precedence was (highest priority starting from left to right):

release_date -> Artists_names -> loudness -> artists_genres -> instrumentalness -> liveness -> valence -> tempo -> accousticness -> speechiness -> danceability -> energy

However, the way it combines the tags, it still will return to a relevant song depending on which tags are combined and their priority to each other. Therefore, this test was successful.

Song searching: The system was put again through regressive manual testing where it searches a lot of different things. The first test was to not search for anything. And do that repeatedly to see what output it gives me each trial. It was found that there is no errors when searching the same thing multiple times, just the same output will be shown. In addition, empty searches result in the same output as well and is not randomized.

The next test was to put two unlikely tags together to see what is outputted. This was done with a genre of pop and artist Logic. The output was that it showed Logic songs (because artist is prioritized rather than the genre). This shows that no matter if tags don’t necessarily work well together, the system itself will still function and output something related to one of those tags.